Introduction to Logo

Logo is a programming language which has been designed to be both very powerful and also very easy to use. Atari Logo has special provision for graphics and animation as well as conversational text manipulation.

We have prepared a series of curriculum "units" to get you started with Logo. Each unit is one sheet of paper. Some of the early units will take you only a couple of minutes to go through; some of the later ones might take up to an hour.

There are two different sequences of units. One is about graphics. These units have names like LOGO-AG3. (The AG stands for Atari Graphics.) The other sequence is about English language processing; these units have names like LOGO-AL5. You can start with either sequence. If you get bored, try the other one!

Please note that these units just teach you the minimum information that you need to get started with Logo! The really interesting part comes when you abandon the units and start working on a project of your own invention. Feel free to do that at any time, even if you haven't finished a sequence of units. In fact, after each unit you might want to spend some time just fooling around with what you've learned before going on to another unit.

> >> When you see an indented paragraph starting with arrowheads, like this one, it is an instruction for you to do something with the computer, or think about the answer to a question.

Have fun!

LOGO-AG1: Graphics

This is the first of a series of units in which you will learn to draw pictures on the TV screen. If this were an algebra class, we would pretend the screen was a sheet of graph paper, and you would tell the computer what curve to draw by specifying a mathematical function, like y=3x+7. That kind of graphing is an important real-world application of computer graphics, but if you want to draw pictures instead of graphs (also important in the real world) it is easier to think not in terms of graph paper but in terms of a pen moving over a sheet of paper; a picture is described as a series of steps like "turn 30 degrees to the right" and "draw a line two inches forward, the way you're facing."

>> Type the Logo command CS

(CS stands for Clear Screen.) What you see is that most of the text which was on the screen has disappeared. The top part of the screen is empty except for a small turtle shape near the middle. The turtle is used in honor of the original Logo graphics device, a small robot shaped like a turtle which draws pictures by rolling around the floor with a pen. The turtle now points toward the top of the screen. You'll see that it can change its orientation as well as moving around.

>> Type the command FORWARD 40

Notice that the turtle moves in the direction it was pointing, which in this case is toward the top of the screen.

>> Now type RIGHT 90

Notice that the turtle doesn't change its position, and it doesn't draw any new lines, but it does turn so that it is now facing to the right of the screen.

>> Type FORWARD 40 again.

Remember, "forward" means to move in whatever direction the turtle is pointing.

>> In what direction will the turtle point if you say RIGHT 90 again?

>> Try it and see if you were right.

"Right" means to turn to the turtle's own right, not necessarily to your right or toward the right side of the screen.

>> Using FORWARD, BACK, LEFT, and RIGHT as needed, finish drawing the square which we've started.

>> Now erase the screen with the CS command and see if you can draw a rectangle twice as long as it is tall.

(over)

>> Try the command PENUP and then move the turtle around. This command "lifts the pen up off the paper"; the turtle no longer leaves a track.

>> Now try PENDOWN and move around some more.

Here are abbreviations for the Logo commands you've learned so far:

FORWARD	FD
BACK	BK
LEFT	LT
RIGHT	RT
PENDOWN	PD
PENUP	PU
CS	(Clear Screen)

LOGO-AG2: Procedures

You have learned how to use simple commands like FORWARD and RIGHT to draw pictures. But if you want to draw anything at all complicated, you would need to use hundreds of these commands, and you'd be very likely to make mistakes. For example, a chessboard is simply 64 squares, so you theoretically know enough to be able to draw one, but you'd almost certainly give up out of boredom or frustration.

>> Try to draw a three by three chessboard (9 squares). Count how many commands you needed, and how many mistakes you made.

If you want the computer to do a sequence of things several times, you can create a new command which will mean to do that sequence. For example, if you define a sequence of commands called "SQUARE" which tell the computer how to draw a square, you can then use the word SQUARE as a command just as you use the word FORWARD. Such a sequence of instructions to the computer is called a procedure. The commands which Logo already understands, like FORWARD, are procedures which someone else has already written for you.

>> Type in exactly what you see below.

```
TO SQUARE
```

(Notice that Logo responds with a > instead of ? to prompt for the next line. This is to remind you that the instructions you type are part of a procedure you're defining.)

```
FORWARD 40
RIGHT 90
FORWARD 40
RIGHT 90
FORWARD 40
RIGHT 90
FORWARD 40
END
```

"TO SQUARE" means "I'm going to teach you (the computer) how to square." When you give this command, Logo starts remembering whatever instructions you type as part of the procedure SQUARE. Normally, you'll recall, when you type in an instruction, Logo simply carries it out right away. But when you are defining a procedure, Logo doesn't carry out the instructions until later. When you use the procedure SQUARE, these remembered commands will be executed one by one, in the order in which they appear in the procedure definition. Soon you'll learn how to use the Logo editor to make corrections to the instructions in your procedure.

>> Clear the screen, then type SQUARE

It's important for you to understand that a procedure you define, like SQUARE, is used in exactly the same way as a procedure which is built into Logo, like FORWARD. In either case, you just say the name of the command.

>> Without clearing the screen, type SQUARE again. Did it do what you expected? If you type SQUARE once more, where will the square go?

Don't go on until you understand why the computer draws the square in a different place each time. It has to do with which way the turtle is facing before and after drawing the square.

(over)

What do you have to know about where the turtle is, to be able to predict where it will draw a square? There are only two things: its position and its heading. That is, you must know where the turtle is, and which way it's pointing. You don't have to know, for example, how many lines it has already drawn to get where it is. Those two important things are called the state of the turtle.

Suppose you'd like the SQUARE procedure to draw the square in the same place every time. Then you have to change the procedure so that the turtle's state is the same after the procedure as it is before.

>> How does SQUARE change the state of the turtle? Is its final position the same as its starting position? Is its final heading the same as its starting heading?

>> Before going on, figure out how to change the SQUARE procedure to make the final state of the turtle the same as its beginning state.

>> Now type the following:

```
EDIT "SQUARE
```

(Notice the quotation mark before the word SQUARE!) This command starts up the Logo editor, with the definition of your procedure SQUARE on the screen. In the editor, when you type an ordinary character (a letter, a space, or a punctuation character) whatever you typed is inserted into your procedure, right where the cursor is. (The cursor is the blinking block which is now at the top left corner of the screen.) To make changes elsewhere in the definition, you must first move the cursor where you want it. You do this with CTRL characters. (That means to hold down the CTRL key while typing the character.)

>> Holding down the CTRL key, use the four arrow keys near the right end of the keyboard to move the cursor. What happens if you try to move right past the end of a line? Down past the end of the procedure?

>> Now move to the beginning of the line saying END. Then type CTRL-INSERT (the INSERT key is on the top row of the keyboard). This command opens a new, blank line for you to type on.

>> Now type RIGHT 90

You have just added a new command to your SQUARE procedure. The change is not "official" until you leave the editor by typing the ESC key (top left corner of the keyboard).

>> Do that now.

>> Clear the screen, then type SQUARE a few times. What happens?

Since the SQUARE procedure is now state-invariant (that is, it doesn't change the turtle's state), it draws the same square in the same place each time.

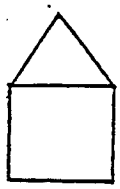
>> Invent a procedure TRIANGLE (you'll start by typing TO TRIANGLE) which draws an equilateral triangle (all sides and angles the same). This is a hard problem!! You'll probably have to EDIT your procedure several times before you get it working right. Ask for help if you need it, but please don't get discouraged if you can't do it right away.

LOGO-AG3: Subprocedures

If everything is going according to plan, you have written procedures named SQUARE and TRIANGLE which draw squares and triangles. (If not, you should do unit LOGO-AG2 before this one.)

>> Make sure your SQUARE and TRIANGLE procedures still work.

In this unit, you are going to write a procedure called HOUSE which will draw this figure:



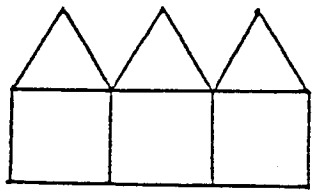
You will not simply use lots of FORWARDs and RIGHTs. The figure contains a square and a triangle, and you should use your existing procedures SQUARE and TRIANGLE as subprocedures of HOUSE. That is, you should start like this:

```
TO HOUSE
SQUARE
  ... and so on.
```

>> Okay, go to it!

Again, don't be discouraged if it doesn't work the first time. Professional programmers don't write perfect programs the first time either. They do just what you're doing: they write a program, they try it out, they find an error, they fix it, and they try again. This is called debugging.

>> Then write procedure SUBURB which draws this:



Don't turn the page until you've written it.

(over)

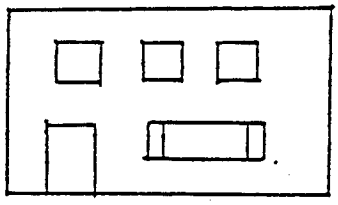
If your procedure SUBURB doesn't use HOUSE as a subprocedure, go tell your teacher to hit you over the head.

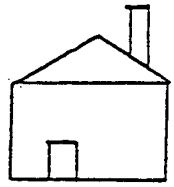
By the way, you've learned how you can edit a procedure to insert a new line, or to change a line, but what if you want to erase an old line altogether? The answer is that you use the CTRL-CLEAR or SHIFT-DELETE command in the editor (this kills the line to the right of the cursor).

If you want to erase a single character, you can either use the DELETE key, which you already know about, or the CTRL-DELETE character. DELETE erases the character to the left of the cursor; CTRL-DELETE deletes the character at the cursor. There are two of them because DELETE is most convenient for fixing a mistake you notice as soon as you type it, while CTRL-DELETE is best for deleting something you've had to move the cursor to get to.

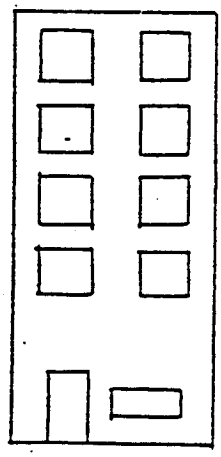
LOGO-AG4: Drawing Pictures

You now know enough about programming to be able to draw some moderately interesting pictures. For example, you could draw a more complicated house:





or an apartment building:



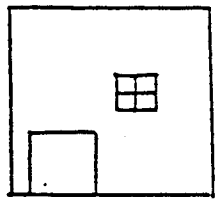
>> Write procedures for one or both of these.

>> Now figure out something else you can draw, and do it.

This would be a good time to show off what you've done to your teacher. If you enjoy drawings like this, and want to do some more before learning more new Logo commands, please feel free to do so.

LOGO-AG5: Procedures with Inputs

We're going to draw this new kind of house:



Here is one possible set of procedures to do it. Don't type this to the computer, just read it:

```
TO NEWHOUSE
FRAME
PENUP
RIGHT 90
FORWARD 10
LEFT 90
PENDOWN
DOOR
PENUP
FORWARD 50
RIGHT 90
FORWARD 50
PENDOWN
WINDOW
RIGHT 90
WINDOW
RIGHT 90
WINDOW
RIGHT 90
WINDOW
END
```

```
TO DOOR
FORWARD 30
RIGHT 90
FORWARD 30
RIGHT 90
FORWARD 30
RIGHT 90
FORWARD 30
RIGHT 90
END

TO WINDOW
FORWARD 10
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 10
RIGHT 90
FORWARD 10
RIGHT 90
END
```

```
TO FRAME
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
FORWARD 100
RIGHT 90
END
```

>> Without using the computer, go through these procedures yourself and convince yourself that they should work.

Notice that FRAME, DOOR, and WINDOW are very similar? They're all the same as the SQUARE procedure, but with different lengths for the sides of the square. It would be nice if we could have one procedure which you could use with a number, like the number used with FORWARD, to say how big the square should be. Here's how:

(over)

```
TO SQ :LENGTH
FORWARD :LENGTH
RIGHT 90
FORWARD :LENGTH
RIGHT 90
FORWARD :LENGTH
RIGHT 90
FORWARD :LENGTH
RIGHT 90
END
```

>> Type that in, then try SQ 100 and SQ 50 and so on.

The number you type after the word SQ is called the input to procedure SQ. Each Logo procedure requires a specific number of inputs. For example, FORWARD takes one input, and PENUP takes no inputs. The word LENGTH in the definition of SQ is the name of a variable. A variable is a sort of container, which has a name and a thing. The name is a way of referring to the variable, like the label on a jar. The thing is whatever is stored in the variable. Right now, we're using the variable LENGTH to store a number, although you'll see later that other things can also be stored in variables. When something like :LENGTH is used in a Logo program, it is an abbreviation for "the thing of the variable named length." When reading a program out loud, pronounce :LENGTH either "dots length" or "thing of length."

The title line of procedure SQ (the line starting with TO) tells Logo that your procedure is named SQ, that it has one input, and that when the procedure is used, whatever it is given as its input should be stored in the variable LENGTH.

Because people often find this business of inputs and variables confusing, please read this carefully and be sure you understand it. If you have trouble, ask for help. When you are defining the procedure SQ, its input has no specific value. You don't know how big a square you'll want to draw. So instead of putting a specific length into the program, as you did in the earlier versions of SQUARE, you use a variable, which will have a different thing stored in it each time you use the procedure. Just as the command FORWARD 100 means "Carry out the procedure FORWARD, and give it the number 100 as its input," the command FORWARD :LENGTH means "Carry out the procedure FORWARD, and give it the thing of variable LENGTH as its input."

When you want to use procedure SQ, you don't refer to the variable LENGTH by name. You just say SQ 100 or whatever number you want to use, just as you would say FORWARD 100. You don't have to know anything about the internal workings of SQ (such as the name it uses for its input) in order to use it. Once you've written it, you can just pretend it's a built-in Logo command.

>> Write a version of procedure NEWHOUSE which uses SQ instead of FRAME, DOOR, and WINDOW.

LOGO-AG6: Regular Polygons; Repeat

You know that to draw a square you must turn through four angles of 90 degrees each. We hope you know that to draw an equilateral triangle you must turn through three angles of 120 degrees each:

```
TO TRIANGLE :SIDE
FORWARD :SIDE
RIGHT 120
FORWARD :SIDE
RIGHT 120
FORWARD :SIDE
RIGHT 120
END
```

Why 120 degrees? Well, if you draw any polygon, and leave the turtle in the same place where it started, and pointing in the same direction, you must have turned through a complete circle, 360 degrees. In the case of the square, you do it in four turns, so each must be 360/4 or 90 degrees. Similarly, for the triangle, you must turn through a total of 360 degrees in three turns, so each is 360/3 or 120 degrees.

>> Write a program to draw a regular pentagon (a five-sided figure with all sides and angles equal).

>> Write a program to draw a regular hexagon (six sides).

You're probably getting very tired of typing the same thing over and over again. Here is a new technique you can use for the hexagon:

```
TO HEX :LENGTH
REPEAT 6 [FD :LENGTH RT 60]
END
```

To write the procedure HEX we've used a new Logo command, REPEAT. This command has two inputs. The first is a number. The second is a Logo command line enclosed in square brackets. (The square brackets indicate a type of Logo object called a sentence. If you don't know what that means, you'll find out in the LOGO-AL series of units.) What REPEAT does is to carry out the command which is its second input repeatedly, the number of times indicated by its first input.

>> Try typing this command to Logo:

```
REPEAT 5 [PRINT "HELLO]
```

(Don't forget about the quotation mark in the above example.) Try using REPEAT to simplify some of the graphics procedures you've written earlier.

LOGO-AG7: Color

So far, you have been drawing pictures in just a single color. The Atari graphics hardware allows you to draw three colors on the screen at once, plus the background color. There are 128 different possible colors to choose from.

>> Type this command:

```
SETBG 88
```

SETBG stands for "set background." If your TV is adjusted properly, the screen is now green instead of blue.

>> Try other numbers as inputs to SETBG. The allowable numbers range from 0 to 127. Can you get back to a blue background?

>> If you'd like to explore the color possibilities more systematically, try this program:

```
TO RAINBOW :COLOR
SETBG :COLOR
PRINT :COLOR
WAIT 20
RAINBOW :COLOR+1
END

RAINBOW 0
```

So far we have changed the background color, but not the color with which the turtle draws. To do that, you must understand that the turtle actually has three different "pens" to draw with. Each pen is a different color. The command SETPN (set pen number) tells the turtle which pen to use. The pens are numbered 0, 1, and 2.

>> Clear the screen and type these commands:

```
SETPN 0
REPEAT 30 [FD 80 RT 178]
SETPN 1
REPEAT 30 [FD 80 RT 178]
SETPN 2
REPEAT 30 [FD 80 RT 178]
```

You should now have on the screen little wedges in three different colors.

(over)

You can change the color of each of the three pens. Unlike real pens, though, when you change the pen color, the lines which you've already drawn with that pen change color also.

>> Type this command:

SETPC 0 7

The SETPC (set pen color) command changes the color of one of the three pens. This command requires two inputs. The first (0 in the example we used) is the pen number whose color you want to change. The second (7 in our example) is the number of the color you want to change it to. So we said to change pen 0 to white (which is what color 7 is).

LOGO-AG8: Multiple Turtles

In Atari Logo you can give commands to four different turtles, not just the one we've been using. The turtles are named by numbers, from 0 to 3. (Try not to get confused about the use of turtle numbers, pen numbers, and color numbers!)

>> Clear the screen.

>> Type a command like FD 80 to move turtle 0 somewhere.

>> Now type this command:

TELL 1

The TELL command tells Logo which turtle to "talk to" when you use graphics commands like FORWARD or SETPN. So you've just said that graphics commands from now on should go to turtle number 1.

If you haven't been using multiple turtles before, turtle 1 just appeared on your TV screen. Each turtle appears automatically the first time you TELL it to become active. If you've already used a turtle and then removed it (you don't know how to do that yet, but you will in two sentences!) then it doesn't automatically appear when you TELL it. Instead, you have to use the command ST (for Show Turtle) to mke it appear. To get rid of a turtle, you give it the command HT (Hide Turtle).

>> Draw something with turtle 1. You might want to turn it before moving it forward, so it won't follow the same path as turtle 0.

>> Now give the command CLEAN which will erase the lines you've drawn but leave the turtles where they are.

>> Now give this command:

TELL [0 1]

(Those are square brackets, not parentheses.)

>> Now try drawing something.

When you give the TELL command a list of numbers as input instead of a single number, it talks to more than one turtle at a time.

(over)

>> Type the following commands. Notice what happens after each command you type.

```
TELL [0 1 2 3]
CS
EACH [FD 30*WHO]
RT 90
FD 50
RT 45
EACH [FD 20*WHO]
```

The EACH command is used to make the turtles do slightly different things. The input to EACH is a sentence (in square brackets) which is a Logo command. Each of the currently active turtles does this command, one at a time. (The currently active turtles are the ones you listed in the most recent TELL command. In this example, all four turtles are active.) The procedure WHO has the number of the current turtle as its value. So in the first EACH command above, turtle 0 moves a distance of 30*0, turtle 1 moves 30*1, and so on. (The * symbol represents multiplication.)

>> If you're interested in multiple turtles, spend some time getting acquainted with TELL, EACH, and WHO. Here is a cute trick:

```
TELL [0 1 2 3]
CS
EACH [RT 90*WHO]
PU
FORWARD 60
PD
REPEAT 6 [FD 30 EACH [RT 360/(WHO+3)]]
```

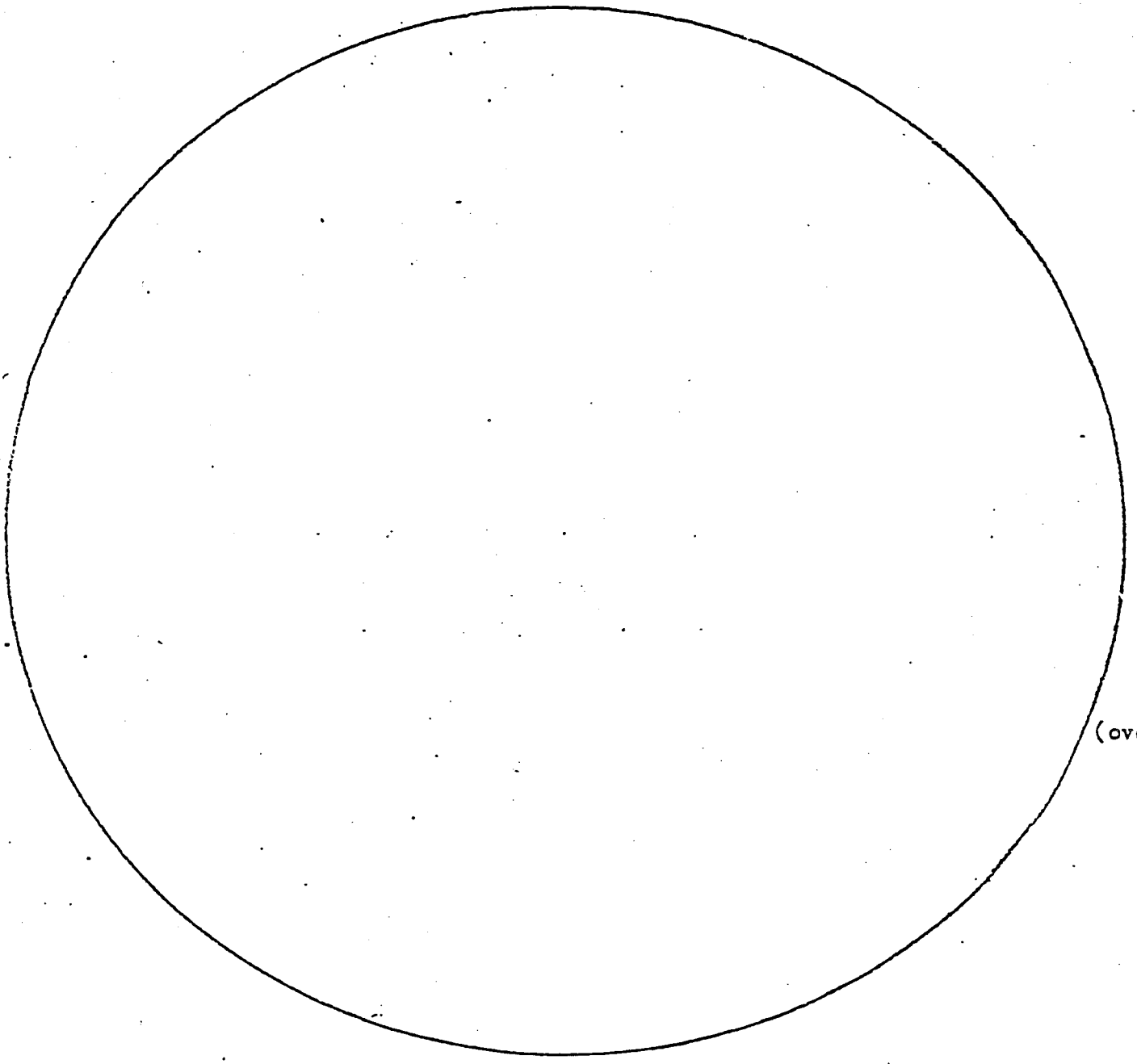
(Be careful of the difference between square brackets and parentheses in that last command!) Again, note what each command does as you type it.

>> To see the missing part of the picture, type CTRL-F (that is, hold down the CTRL key and type F). This stands for Fullscreen. To see your text again, type CTRL-S which stands for Splitscreen.

LOGO-AG9: Circles

>> How can you draw a circle with the turtle? How can you make the turtle follow a curved path when it can only move FORWARD or BACK, and turn LEFT or RIGHT?

>> Pretend you're a turtle and try walking around this circle with a pencil, using only turtle steps:



(over)

A circle isn't a polygon with straight sides; to draw it correctly the turtle would have to know how to turn and move at the same time. But we can draw a good approximation:

>> Write a program to draw a regular icosagon (a 20-sided polygon).

>> Write a program to draw a regular 72-gon.

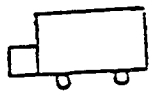
>> What is the closest you can get to a smooth circle? Draw one. How much better is it than the 72-gon? How much longer does it take to draw? Is it worth it?

>> As the number of sides you use gets bigger, the length of each side must get smaller in order to keep the circle the same size. How much smaller? If you double the number of sides, do you divide the length of the sides by two? What about the angle? Does it get bigger or smaller? By how much?

>> How small a circle can you draw and have it look reasonably good?

>> How large a circle can you draw and have it look reasonably good?

>> Write a program to draw a truck:



LOGO-AG10: Shape and Speed

So far, we've used the turtles mostly to make "snapshots": we draw a picture which is static once we've finished it. You can also use the turtles for animation. You can have different shapes moving around the screen and interacting with a "backdrop" you've drawn.

First of all, you have to know how to change the shape of a turtle. You create new shapes using the shape editor. A shape is a rectangular array of dots which can be light or dark. The array is 8 dots wide by 16 dots tall. Logo can remember 15 of your shapes at a time, not counting the regular turtle shapes. The shapes you create are numbered 1 to 15; the turtle shape is number 0.

>> Type this command:

EDSH 1

You are now in the shape editor. On the screen is a big grid of boxes, representing the dots in the shape you're making. The actual shape, of course, will be much smaller than this grid when you use it.

As in the procedure editor, there is a cursor, which is now in the top left corner of the grid. Use the CTRL-arrow key combinations to move the cursor, as in the procedure editor.

To change the shape, use the space bar. This will invert the box where the cursor is: if the box was light, it becomes dark, or the other way around.

>> Use the arrows and the space bar to fill in the boxes on the edges of the grid, making a hollow box.

>> Then type the ESC key to leave the shape editor.

Because you gave the command EDSH 1, the shape you've just defined is shape number 1. The command to tell a turtle what shape to use is called SETSH.

>> If you don't have a turtle on the screen, type CS

>> Type SETSH 1

The turtle is now using your box shape instead of the turtle shape. The command SETSH 0 will return to the turtle shape.

One difference between shapes you define and the standard turtle shape is that yours don't rotate as the turtle turns. That's because "the turtle shape" is actually several shapes, pointing in different directions.

(over)

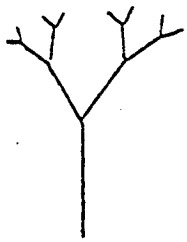
Speed. Turtles aren't limited to jumping around all at once, as they do when you use the FORWARD and BACK commands. They can also move smoothly at a speed you select. The command SETSP sets the speed of the turtle. The speed is in steps per second. The maximum speed is 150.

If the pen is down while the turtle has a speed, it draws as it moves. However, if the speed is greater than 30, it may draw dotted lines rather than solid lines.

To stop the turtle, set its speed to zero.

LOGO-AG11: Tree

We are going to draw this figure:



How many lines are there in this tree? If you wrote a program to draw it line by line, how many steps would it take? But there is a very clever way to solve the problem much more easily. Notice that the complete tree can be dvided into three parts:

the trunk

the left subtree

the right subtree

Drawing the trunk is easy. The secret is to notice that the two subtrees are exactly the same as the complete figure, except that the branches are smaller and there are fewer of them. The procedure to draw the complete tree can use ITSELF as a subprocedure to draw the subtrees. The use of a procedure as a subprocedure of itself is called recursion.

To see how this works, let's start with a simpler figure:

```
TO TREE1 :LENGTH
FORWARD :LENGTH
BACK :LENGTH
END
```

This procedure draws a rather trivial "tree" with no branches, just a trunk. It leaves the turtle in the same place it starts out. That's going to be important, when we use these procedures as subprocedures of more complicated trees. Here is a slightly more interesting tree:

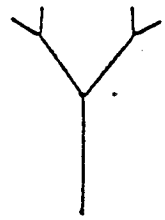
```
TO TREE2 :LENGTH
FORWARD :LENGTH
LEFT 30
TREE1 :LENGTH/2
RIGHT 60
TREE1 :LENGTH/2
LEFT 30
BACK :LENGTH
END
```

Don't go on until you're sure you understand every line of this procedure! Two things to notice are, first, that TREE2 depends on the fact that TREE1 leaves the turtle where it found it, and second, that TREE1 and TREE2 both use a variable named LENGTH, but the two variables are separate. That is, the fact that the variable LENGTH in TREE1 has a smaller value than the variable LENGTH in TREE2 does not change the value of LENGTH in TREE2.

(over)

Now, here's the next step up:

```
TO TREE3 :LENGTH
FORWARD :LENGTH
LEFT 30
TREE2 :LENGTH/2
RIGHT 60
TREE2 :LENGTH/2
LEFT 30
BACK :LENGTH
END
```



Do you see the similarity between TREE3 and TREE2? These two procedures are almost identical; the only difference is that TREE3 uses TREE2 as a subprocedure, whereas TREE2 uses TREE1 as a subprocedure.

>> If you haven't done so already, try all this on the computer.

>> Write TREE4, using TREE3 as a subprocedure.

It seems a shame to have to write all these almost-identical procedures. Instead of a separate procedure for each level of complexity, it would be nice to use a second variable to represent the level:

```
TO TREE :DEPTH :LENGTH
FORWARD :LENGTH
LEFT 30
TREE :DEPTH-1 :LENGTH/2
RIGHT 60
TREE :DEPTH-1 :LENGTH/2
LEFT 30
BACK :LENGTH
END
```

In this approach, instead of saying something like

```
TREE4 60
```

you use the command

```
TREE 4 60
```

>> Type in this TREE procedure and try it.

>> Why doesn't it work?

When you decide to stop the procedure, do it by typing the ESC key. We'll debug this program in the next unit.

LOGO-AG12: More about Tree

Why doesn't the TREE program in the last unit work? The problem is that this procedure doesn't know when to stop using itself as a subprocedure. In the original version, TREE1 was very different from the others--it had no subprocedures to draw subtrees. In the recursive TREE procedure, we need a way to tell that we've reached the smallest level of subtree. To do this, you need to learn about two new Logo procedures, IF and EQUALP.

>> Type these LOGO commands:

```
PRINT EQUALP 3 2+1
PRINT EQUALP 5 2+2
```

The procedure EQUALP is a special kind of Logo procedure called a predicate. A predicate is a procedure which gives a result, called an output, which is either the word TRUE or the word FALSE. In particular, EQUALP is a procedure with two inputs, which outputs TRUE if the two inputs are equal, and FALSE if they're not equal.

>> Now try these commands:

```
IF EQUALP 3 2+1 [PRINT "HUZZAH]
IF EQUALP 5 2+2 [PRINT "GARPLY]
```

The command IF takes two inputs. The first input must be either TRUE or FALSE. (You can see that this requirement fits in nicely with the outputs from predicates!) If the first input is TRUE, then the second input, a list, is executed as a Logo command. (For example, in the first line above the first input to IF is EQUALP 3 2+1 and the second is [PRINT "HUZZAH].) If the input to IF is FALSE, the second input is ignored.

Now we can write a version of TREE which knows when to stop:

```
TO TREE :DEPTH :LENGTH
IF EQUALP :DEPTH 0 [STOP]
FORWARD :LENGTH
LEFT 30
TREE :DEPTH-1 :LENGTH/2
RIGHT 60
TREE :DEPTH-1 :LENGTH/2
LEFT 30
BACK :LENGTH
END
```

We have added the line starting IF to solve the problem of knowing when to stop. What it means is this: "If the value of the variable DEPTH is equal to zero, stop this procedure. Don't go on to the next line. Instead, continue with whatever procedure called this one as a subprocedure."

>> Type this all in and try it out. You may have to move the turtle down below the center of the screen before you start in order to get it all to fit.

(over)

If you have trouble understanding how this all works, you might try this: edit TREE to start with these lines:

```
REPEAT 6-:DEPTH [TYPE ".]
PR (SE [ENTERING TREE, DEPTH] :DEPTH "LENGTH :LENGTH)
WAIT 20
```

You might add a similar message at the end of the procedure. Seeing these messages as the tree is drawn might help you understand how this all works.

LOGO-AL1: Conversational Programs

This is the first of a series of units about computer programming using the Logo language. In these units you'll learn how to write "conversational" programs--ones which manipulate words and sentences to carry on a conversation with you.

>> Type the LOGO statement

PRINT "AARDVARK

(Don't forget the quotation mark!)
(All LOGO commands end with the RETURN key.)

What you see on the TV screen should look like this (what the computer typed is underlined; what you typed isn't):

?PRINT "AARDVARK
AARDVARK
?

The command you typed contained two things: the name of a procedure (PRINT), and a word (AARDVARK) which was used as an input to the procedure. The procedure named PRINT requires one input, which it prints on your TV screen.

>> Use the PRINT command to make the computer print your first name.

>> Now make sure you have a floppy disk containing the LOGO-AL sample programs. Put the disk in the disk drive. Ask for help if you're not sure what to do. Then type this command:

LOAD "D:LOGO.AL

The Logo language includes a number of procedures which the computer "knows" how to do automatically. Writing a computer program means creating new procedures by combining these primitive ones which are provided for you. You are about to see an example of a computer program built out of primitive procedures. You aren't expected to understand all the details of this program right now! As we go on, you'll learn how it works.

>> Type the command

QUIZ1

(No space between QUIZ and 1.)

It will ask you a question, which you should answer. If you get the wrong answer, give the QUIZ1 command again to try again.

>> Now type the command

PO "QUIZ1

(over)

The computer should show you the QUIZ1 program. It consists of a title line (the one starting with TO), several statements, and finally the word END.

The title line tells Logo that what follows is the definition of a program. It starts with TO to suggest the idea "here is how to ..." In other words, writing a program is like teaching the computer how to do something.

The following lines are the steps in the program. When you use the program, these statements are executed by the computer, just as if you typed them in one by one the way you typed the PRINT command earlier.

>> Just to give you some practice in typing in a program, type exactly what you see here. If you make a mistake, use the DELETE key to erase what you've typed.

```
TO GREET
```

(Notice that Logo responds with a > instead of ? to prompt for the next line. This is to remind you that the instructions you type are part of a procedure you're defining.)

```
PRINT "HELLO!
PRINT [THIS IS A FRIENDLY PROGRAM.]
END
```

>> Now tell the computer to carry out your program by typing its name (GREET).

LOGO-AL2: Inputs and Outputs

In the last unit, the first thing you typed to Logo was this:

```
PRINT "AARDVARK
```

There are two things in that command line:

(1) PRINT is the name of a procedure. In other words, it is something the computer knows how to do. The particular procedure named PRINT tells the computer to display some information on your TV screen. Other procedures might tell the computer to multiply two numbers, or to draw a line on the screen.

(2) AARDVARK is a word which is used as the input to the procedure. We said that PRINT displays "some information" on your screen; it needs an input to tell it exactly what to display.

(The quotation mark in front of AARDVARK indicates that it is the word "AARDVARK" itself which is the input--it is not the name of something else, the way PRINT is the name of a procedure. Later you'll learn how to use words as names for other objects.)

Complicated programs are also built out of procedures and their inputs. For example, here is the definition of the QUIZ1 program you used in the last unit:

```
[1] TO QUIZ1
[2] TYPE [WHO IS THE GREATEST MUSICIAN OF ALL TIME?]
[3] IF EQUALP RL [JOHN LENNON] [PRINT [YOU'RE RIGHT!] STOP]
[4] PRINT [NO, SILLY, IT'S JOHN LENNON!]
[5] END
```

(The numbers in [brackets] at the left aren't part of the procedure definition; we've just printed them to make it easier to refer to individual lines in the following description.) In line [2], the procedure called TYPE is used. TYPE is just like PRINT except that it does not move to the beginning of a new line after displaying the information.

The input to TYPE is the sentence inside the brackets. A sentence is a group of words taken together as a unit. In line [2], the input to TYPE is not just one word, like AARDVARK, but a group of eight words.

>> In line [4], what procedure is used? _____

>> What is the input to that procedure? _____

>> Is the input a word or a sentence? _____

An input to a procedure can be a specific word or sentence, as in line [2], or it can be the output from another procedure.

(over)

>> Type these Logo commands:

```
PRINT "HELLO
PRINT FIRST "HELLO
PRINT BUTFIRST "HELLO
```

>> In the first of these three commands (PRINT "HELLO), what procedure is used? _____

>> What is the input to the procedure? _____

In the second command (PRINT FIRST "HELLO), the input to PRINT is not the word HELLO. The word HELLO is used as the input to a procedure called FIRST. What this procedure FIRST does is to take a word (in this case HELLO) as input, and pick out the first letter of the word (H) as its output. The output from FIRST is used as the input to PRINT, so what is printed is the letter H.

>> What procedures are used in the second command line? _____

>> What is the input to FIRST? _____

>> What is the output from FIRST? _____

Now consider the third command (PRINT BUTFIRST "HELLO).

>> What did the computer print? _____

>> What is the input to the procedure PRINT? _____

>> What is the input to the procedure BUTFIRST? _____

>> What is the output from the procedure BUTFIRST? _____

>> What does the procedure called BUTFIRST do? ("It takes a word as input, and gives _____ as output.")

>> What do you guess the procedures LAST and BUTLAST do? Try using them with PRINT commands to find out.

It is possible to have a "chain" of procedures, so that the output from one becomes the input to another. Think of it as a kind of bucket brigade, with each procedure passing some information along to another.

>> What do you think this command will do? Try it.

```
PRINT FIRST BUTFIRST "HELLO
```

Did you guess right? The word HELLO is the input to the procedure BUTFIRST. That procedure outputs the word ELLO (all but the first letter), which becomes the input to the procedure FIRST. The first letter of ELLO is E, and that letter is the input to PRINT.

>> What will this do? Decide before you try it.

```
PRINT FIRST BUTFIRST BUTFIRST "HELLO
```

If you're not sure about your answers, show them to a teacher before continuing.

LOGO-AL3: Editing a Program

In this unit we aren't going to discuss sophisticated concepts, but instead we'll go into the details of typing a program into the computer.

Normally Logo indicates that it is ready for you to type a command by printing a question mark at the beginning of a line. When you type a command like the PRINT commands you used in the last unit, the computer executes (carries out) your command and then prints another question mark.

When, in Unit LOGO-AL1, you typed the command TO GREET, the computer responded by allowing you to type in Logo commands, not to be carried out immediately, but to be remembered for later, as part of the procedure GREET. Later, when you use the procedure GREET, these remembered commands will be executed one by one, in the order in which they appear in the procedure definition.

If you make a mistake while typing in a procedure using TO, and you don't notice it until you're on the next line, there is no way to correct it. Also, the TO command can't be used to change a program you defined earlier. To allow for these needs, there is another way to enter or modify a procedure definition: the editor.

The editor has facilities to allow you to change any line in your procedure, or to delete lines completely, or to add lines.

>> We are assuming that you have already created a procedure named GREET in the first unit, and that that program is still available to you. If not, ask for help.

>> Type the following:

```
EDIT "GREET
```

This command starts up the editor, with the definition of your procedure GREET already on the screen. In the editor, when you type an ordinary character (a letter, a space, or a punctuation character) whatever you typed is inserted into your procedure, right where the cursor is. (The cursor is the block which is now at the top left corner of the screen.) To make changes elsewhere in the definition, you must first move the cursor where you want it. You do this with CTRL characters. (That means to hold down CTRL while typing the character.)

>> Hold down the CTRL key and type the down-arrow key.

The control character you typed moved the cursor down to the next line. You can use the four arrow keys (while holding down CTRL) to move up or down by lines, or horizontally. (Try that now.)

A line on the screen only has room for 37 characters. A Logo command line can be much longer than that. The editor indicates this by showing an arrow at the end of a screen line which is not the end of a Logo line. The up and down arrows work in Logo lines, not screen lines.

(over)

>> Use the arrow keys to move to the beginning of the line containing the word FRIENDLY. Then type CTRL-INSERT (the INSERT key is near the top right corner of the keyboard). This command opens a new, blank line for you to type on.

>> Now type PRINT [HOW ARE YOU]

You have just added a new command to your GREET procedure. The change is not "official" until you leave the editor by typing the ESC key (top left corner of the keyboard).

>> Do that now.

>> Tell the computer to execute the GREET procedure. Compare the results with what happened before you modified it.

You've learned how you can edit a procedure to insert a new line, but what if you want to erase an old line? The answer is that you use the CTRL-CLEAR command in the editor (this kills the line to the right of the cursor).

If you want to erase a single character, you can either use the DELETE key, which you already know about, or the CTRL-DELETE combination. DELETE erases the character to the left of the cursor; CTRL-DELETE deletes the character at the cursor. There are two of them because DELETE is most convenient for fixing a mistake you notice as soon as you type it, while CTRL-DELETE is best for deleting something you've had to move the cursor to get to.

>> EDIT your GREET procedure again, removing the line you just added.

Remember that you may have more than one procedure; so far you have used procedures named GREET and QUIZ1. Therefore, when you want to change something in a program, you must first tell the computer which procedure you want to change (by using the EDIT command) before you type in the corrections. You can continue to make corrections until you type ESC to end the editing.

LOGO-AL4: Predicates

We're going to look at the QUIZ1 program again:

```
[1] TO QUIZ1
[2] TYPE [WHO IS THE GREATEST MUSICIAN OF ALL TIME?]
[3] IF EQUALP RL [JOHN LENNON] [PRINT [YOU'RE RIGHT!] STOP]
[4] PRINT [NO, SILLY, IT'S JOHN LENNON!]
[5] END
```

In each of lines [2] and [4], just one procedure is used.

>> What procedure is used in line [2]? _____

>> What is its input? _____

Line [3] is more complicated; it refers to three procedures you haven't learned about yet: IF, EQUALP, and RL. The reason it is complicated is that it has to have a different effect depending on whether or not the person using the program gives the right answer.

In this unit you will learn about the procedure EQUALP. Later you'll see how it fits in with the others in line [3] to accomplish the desired effect. The letter "P" in EQUALP stands for predicate, which means "a procedure whose output is either the word TRUE or the word FALSE". The procedure EQUALP takes two inputs, and its output is TRUE if the two are identical.

>> Type this command to Logo and see what happens:

```
PRINT EQUALP "H "H
```

Both inputs to EQUALP are the letter H, so the output from EQUALP is the word TRUE.

>> Now type this command:

```
PRINT EQUALP "O "Y
```

>> What are the inputs to EQUALP? ____ ____

>> Are the inputs equal?

>> What is the output from EQUALP? _____

(over)

Usually EQUALP is given inputs which are themselves the output from some other procedure:

>> Type this command to Logo and see what happens:

```
PRINT EQUALP "H FIRST "HELLO
```

The first input to EQUALP is the letter H; the second input is the output from FIRST, which is also the letter H. Therefore, the two inputs are equal, and the output from EQUALP is TRUE.

>> Now type this command:

```
PRINT EQUALP "O FIRST "HELLO
```

The first input to EQUALP is the letter O; since that is not the same as the letter H (which is again the output from FIRST of "HELLO), the output from EQUALP is FALSE.

>> Try to predict what each of these lines will do before you type them to the computer:

```
PRINT EQUALP FIRST "HELLO FIRST "HORRIBLE
PRINT EQUALP LAST "HELLO FIRST "OGRE
PRINT EQUALP FIRST "HELLO FIRST "OGRE
PRINT EQUALP "HELLO "HELLO
PRINT EQUALP "HELLO "HI
PRINT EQUALP FIRST BUTFIRST "HELLO LAST "OGRE
```

LOGO-AL5: If

In the last unit you learned about EQUALP, one of several predicates which output either the word TRUE or the word FALSE. In that unit we simply printed the output with commands like

```
PRINT EQUALP "H FIRST "HELLO
```

In general, though, the use of predicates isn't simply to print their output, but rather to allow a program to do two different things depending on whether some condition is met. For example, in the QUIZ1 program, one of two different messages is printed depending on whether or not the user types in "JOHN LENNON". This conditional execution of a command is accomplished by the procedure IF.

The procedure IF takes two inputs. The first is of a special kind; it must be either the word TRUE or the word FALSE. If the input is TRUE then the second input, a list of Logo commands, will be executed; if it's FALSE, the second input is ignored.

>> Type these commands to Logo:

```
IF "TRUE [PRINT "XYLOPHONE]
IF "FALSE [PRINT "FIDUCIARY]
```

>> Notice what happens if you try to give IF an input other than TRUE or FALSE:

```
IF "PHILOSOPHY [PRINT "EXISTENTIALISM]
```

Of course, it isn't very interesting to use IF with an input which is the same every time you use the program. Usually, the input to IF in an actual program is the output from some predicate function. For example, here is the QUIZ1 program again:

```
[1] TO QUIZ1
[2] TYPE [WHO IS THE GREATEST MUSICIAN OF ALL TIME?]
[3] IF EQUALP RL [JOHN LENNON] [PRINT [YOU'RE RIGHT!] STOP]
[4] PRINT [NO, SILLY, IT'S JOHN LENNON!]
[5] END
```

The input to the IF on line [3] is the output from EQUALP, which you studied in Unit LOGO-AL4. We'll look at some simpler examples of using IF with EQUALP.

>> Type this command to Logo and see what happens:

```
IF EQUALP "H FIRST "HELLO [PRINT "YES.]
```

The first input to EQUALP is the letter H; the second input is the output from FIRST, which is also the letter H. Therefore, the two inputs are equal, so the output from EQUALP is the word TRUE, and the PRINT command is executed.

(over)

>> Now type this command:

```
IF EQUALP "O FIRST "HELLO [PRINT "NOPE.]
```

This time, the two inputs to EQUALP are unequal, so the PRINT command is not executed.

>> The two inputs to EQUALP are the letters ____ and ____.

>> The output from EQUALP is the word _____.

>> Try to predict what each of these lines will do before you type them to the computer:

```
IF EQUALP FIRST "COMPUTER FIRST "CRAYON [PRINT "YES.]
IF EQUALP LAST "HELLO FIRST "OGRE [PRINT "YES.]
IF EQUALP FIRST "HELLO FIRST "OGRE [PRINT "YES.]
IF EQUALP "POTSTICKER "POTSTICKER [PRINT "YES.]
IF EQUALP "HELLO "HI [PRINT "YES.]
IF "TRUE [PRINT "YES.]
IF "FALSE [PRINT "YES.]
IF "HELLO [PRINT "YES.]
IF EQUALP FIRST BUTFIRST "HELLO LAST "OGRE [PRINT "YES.]
IF EQUALP "FALSE EQUALP "A "B [PRINT [THIS IS A HARD ONE!]]
```

LOGO-AL6: User Typein

In the QUIZ1 procedure, EQUALP is given two inputs: the first is the output from the procedure RL, and the second is the sentence [JOHN LENNON]. The procedure RL has no inputs. (Recall that EQUALP has two inputs, while PRINT, FIRST, BUTFIRST, and several others have one input.) When your program uses the RL procedure, the computer waits for you to type a line. (RL stands for Read Line.) Whatever you type becomes the output from RL.

>> What will this command do?

```
PRINT RL
```

>> Write your own procedure which uses RL. Here is a simple example to encourage you:

```
TO FLATTER
TYPE [WHAT IS YOUR NAME?]
PRINT RL
PRINT [IS A WONDERFUL PERSON.]
END
```

This might be a good time to find a teacher and show him or her what you've written so far.

Punctuation. When you want to use a specific sentence in your program, you type it inside brackets:

```
[JOHN LENNON]
```

The brackets aren't actually part of the sentence, but instead serve to indicate which words are part of the sentence. When you are typing a sentence to reply to a RL procedure, you don't use the brackets. Similarly, the quotation mark is used to indicate that what follows is a specific word to be used in your program. For example, consider this command:

```
PRINT FIRST "FIRST
```

The first FIRST, without the quotation mark, is the name of a procedure which you are telling the computer to execute. The second FIRST, because of the quotation mark in front of it, is not considered a procedure to execute, but rather a word to be used just as itself. The following commands would be wrong:

```
PRINT FIRST
```
(Procedure FIRST used without an input.)

```
PRINT "FIRST "FIRST
```
(Two inputs to PRINT instead of one.)

LOGO-AL7: Commands and Operations

There are two kinds of procedures in Logo: commands and operations. An operation is a procedure which has an output, which can be used as the input to another procedure. A command is a procedure which does not have an output, but does something by itself. For example, FIRST and EQUALP are operations. (There is a special name for operations which have the kind of output EQUALP does. What is it?) The procedures PRINT and IF are commands.

>> Circle the operations in the following list. Then show it to a teacher to see if you're right.

```
LAST
RL
EDIT
BUTFIRST
QUIZ1
TYPE
BUTLAST
```

Please don't forget the distinction between the words output and print. In Logo, the "output" from a procedure is a word or sentence which is passed from one procedure to another, becoming an input to the other procedure. This all happens within the computer, and has nothing to do with what is printed, which is what the computer displays on your TV screen.

One last thing we haven't explained in the QUIZ1 procedure is the STOP command used in line [3]. This command tells the computer to stop the execution of the procedure it is doing, rather than continuing with the next line. Also, note that you can put more than one command in the second input to IF.

Now for a few more operations to add to your collection. You already know about the operations FIRST, BUTFIRST, LAST, and BUTLAST as applied to words. But each of these procedures can also take a sentence as input.

>> Try these:

```
PRINT FIRST [ONTOGENY RECAPITULATES PHYLOGENY.]
PRINT BUTFIRST [COMPUTER TEACHERS ARE WARM-HEARTED.]
PRINT LAST [COMPUTER STUDENTS ARE ROWDY.]
PRINT BUTLAST [THIS IS INTERESTING. RIGHT?]
```

These operations split words into letters, but they split sentences into words.

(over)

You can find out how big an object is:

```
PRINT COUNT "HELLO
PRINT COUNT [THIS IS A SENTENCE.]
PRINT COUNT [HELLO]
```

Notice the difference between a word with five letters and a sentence with one word! You can combine objects to make a bigger object:

```
PRINT WORD "NOW "HERE
PRINT SE "NOW "HERE
PRINT SE [THIS IS] [A TEST.]
PRINT WORD [THIS IS] [NOT ALLOWED!]
```

(SE stands for SEntence.) You can combine all these operations to extract, for example, the third word of a sentence, and to combine the results of such extractions.

>> Write down what you think each of these will do before you try them on the computer:

```
PRINT FIRST BUTFIRST [BUD'S ICE CREAM]
PRINT FIRST FIRST BUTFIRST [THE COMPUTER IS FUN TO USE.]
PRINT BUTFIRST BUTLAST "OBLATE
PRINT COUNT FIRST [TRICK QUESTION]
PRINT COUNT BUTFIRST [TRICK QUESTION]
```

(BUTFIRST of a sentence is a sentence!)

>> This is a toughie. Don't go on until you can explain the result of this command:

```
PRINT WORD COUNT "SUN COUNT "WHERE
```

Here is a silly program which shows off these operations:

```
[1] TO SILLY
[2] TYPE [WHAT IS YOUR FULL NAME?]
[3] PRINT SE FIRST RL [IS YOUR FIRST NAME.]
[4] END
```

>> In line [3], what operations are used?

>> How many inputs does RL have?

>> What is the input to FIRST?

>> How many inputs does SE have?

>> What are the inputs to SE?

LOGO-AL8: Writing Procedures with Inputs

You must be pretty bored with the QUIZ1 program by now. Since it only knows about one question and one answer, it can't keep you entertained very long. It would be better if we could write a quiz program with several questions and answers, like this:

```
TO QUIZ
QUEST [WHO IS THE GREATEST MUSICIAN OF ALL TIME?] [JOHN LENNON]
QUEST [WHAT IS THE BEST CITY IN THE WORLD?] [SAN FRANCISCO]
QUEST [WHAT COLOR WAS GEORGE WASHINGTON'S WHITE HORSE?] [WHITE]
QUEST [HOW MUCH IS 2+2?] [5]
QUEST [WHO WROTE "COMPULSORY MISEDUCATION"?] [PAUL GOODMAN]
END
```

In order to make this work, though, we need a procedure called QUEST which will be very much like QUIZ1 except that instead of asking a constant question (the same one every time), it asks a variable question, depending on what you use as its input. Until now, all the procedures we've written haven't had inputs. (The primitive procedures, which are built into Logo, have had inputs.) On the floppy disk of sample programs we have provided QUEST. Here is what it looks like:

```
[1] TO QUEST :QUESTION :ANSWER
[2] TYPE :QUESTION
[3] IF EQUALP RL :ANSWER [PRINT [YOU'RE RIGHT!] STOP]
[4] PRINT SE [NO, SILLY, IT'S] :ANSWER
[5] END
```

>> If you have not already loaded a workspace file into the computer, get the LOGO-AL program diskette and do so. Ask for help if you need it.

>> Use QUEST to write your own quiz program, making up your own questions. Try it out on someone.

In case you've forgotten, here is what QUIZ1 looks like. Compare it to QUEST.

```
[1] TO QUIZ1
[2] TYPE [WHO IS THE GREATEST MUSICIAN OF ALL TIME?]
[3] IF EQUALP RL [JOHN LENNON] [PRINT [YOU'RE RIGHT!] STOP]
[4] PRINT [NO, SILLY, IT'S JOHN LENNON!]
[5] END
```

The differences have to do with the use of :QUESTION and :ANSWER instead of the specific question and answer built into QUIZ1. QUESTION and ANSWER are variables.

(over)

Think of a variable as a kind of box associated with a procedure. The box has a name, which is a sort of label written on the side of the box. It also has a thing or a value (the words mean the same thing), which is whatever is stored inside the box. Logo variables can contain words or sentences as their values. (Later you'll learn about other things which can be in a variable.)

The notation :QUESTION (note the colon before the name) means "the value of the variable whose name is QUESTION". So in procedure QUEST, the meaning of line [2] is "type the value of the variable whose name is QUESTION", just as line [2] of procedure QUIZ1 means "type the constant sentence 'WHO IS THE GREATEST MUSICIAN OF ALL TIME?'".

>> Type in this procedure:

```
TO ECHO :WHATEVER
PRINT SE [THE INPUT IS] :WHATEVER
END
```

>> Then try it with different inputs:

```
ECHO [WHAT YOU SEE IS WHAT YOU GET.]
ECHO "MAYONNAISE
ECHO FIRST [YOU CAN USE COMPLICATED INPUTS.]
```

>> What did the computer print when you tried ECHO "MAYONNAISE? _____

>> What was the value of the variable WHATEVER? _____

>> How many inputs does the procedure ECHO have? _____

>> How many variables are mentioned in the title line of procedure ECHO? _____

The variables named in the title line of a procedure you write are associated with the inputs to that procedure. Each input becomes the value of one variable.

>> How many inputs does the procedure QUEST have? _____

>> Which variable in QUEST gets the first input as its value? _____

>> Consider this example:

```
QUEST [WHY DID THE CHICKEN CROSS THE ROAD?] [TO KEEP HIS PANTS UP.]
```

>> When this example command is executed, what is the value of the variable QUESTION in procedure QUEST? _____

>> What is the value of the variable ANSWER? _____

LOGO-AL9: More Procedures with Inputs

The use of variables to allow your procedures to have inputs is a very powerful technique. Here are a few examples you can try out:

```
TO NAME
TYPE [WHAT IS YOUR FULL NAME?]
IDENTIFY RL
END

TO IDENTIFY :SOMEONE
PRINT SE [YOUR FIRST NAME IS] FIRST :SOMEONE
PRINT SE [YOUR LAST NAME IS] LAST :SOMEONE
IF EQUALP 2 COUNT :SOMEONE [PRINT [NO MIDDLE NAME!] STOP]
PRINT SE [YOUR MIDDLE NAME IS] BUTFIRST BUTLAST :SOMEONE
END

NAME
```

```
TO TRIPLE :HE :SHE :IT
PRINT (SE :HE [IS IN LOVE WITH] :SHE)
PRINT (SE :SHE "PREFERS :IT "HOWEVER.)
PRINT (SE "THEREFORE, :HE [IS MAD AT] :IT)
END

TRIPLE [JONATHAN SWIFT] [LOUISA MAY ALCOTT] [BEN JOHNSON]
```

The parentheses (...) can be used with certain procedures, most notably SE and WORD, to allow them to handle more than two inputs.

>> Write some procedures of your own, using everything you've learned so far. Take your time! You should practice with these ideas enough to feel familiar with them. Show someone what you write.

LOGO-AL10: Lists

One of the problems with our QUIZ1 and QUEST programs is that they are very fussy about getting precisely the answer they expect. For example, if you say that the greatest musician of all time is "LENNON" instead of "JOHN LENNON" you'll be told you have the wrong answer. For that matter, the program should probably also accept "THE BEATLES" as a correct answer. In this unit we'll teach QUEST to recognize several correct answers to a question instead of just one.

The version of QUEST we've used so far has two inputs, the question and the answer, each of which is a Logo sentence. What we would like to do is have the second input be not just one sentence, but an arbitrary number of sentences, each of which is an acceptable answer to the question.

A sentence in Logo is simply a list of words, considered as one large object. It turns out that you can also use more complicated lists in which the individual pieces of the list are themselves lists. For example, try these:

```
PRINT [THIS IS JUST A SENTENCE.]
PRINT FIRST [THIS IS JUST A SENTENCE.]
PRINT COUNT [THIS IS JUST A SENTENCE.]
PRINT [[THIS IS ONE SENTENCE.] [THIS IS A SECOND SENTENCE.]]
PRINT FIRST [[THIS IS ONE SENTENCE.] [THIS IS A SECOND SENTENCE.]]
PRINT COUNT [[THIS IS ONE SENTENCE.] [THIS IS A SECOND SENTENCE.]]
```

>> How many words are in the sentence printed in the first example? _____

>> How many sentences are in the list in the last example? _____

The plan is to modify QUEST so that its second input is a list of sentences, any of which is a correct answer. So a sample use of the new (not yet written) QUEST will look like this:

```
QUEST [WHO IS THE GREATEST MUSICIAN OF ALL TIME] [ [JOHN LENNON]
                                    [LENNON] [THE BEATLES] [BEATLES] ]
```

(Although the example is printed on two lines on this sheet, you must type long lists like this without typing the RETURN key in the middle. What you type will be shown on more than one line of the TV screen, if necessary, but is considered one long line as far as Logo is concerned.) We have listed four possible correct answers to the question.

>> What are they? _____ _____

_____ _____

(over)

Before, we used the predicate EQUALP to compare the answer actually typed in by the person using the program against the single correct answer contained in the variable ANSWER. Now, we want to ask whether the typed-in answer is the same as any member of the list of possible answers. The predicate MEMBERP will do that:

```
PRINT MEMBERP "HELLO [I AM THE EGGMAN]
PRINT MEMBERP "HELLO [YOU SAY GOODBYE, I SAY HELLO]
PRINT MEMBERP [RUBY TUESDAY] [ [HELLO, GOODBYE]
                               [LADY MADONNA] [HELP!] ]
PRINT MEMBERP [LADY MADONNA] [ [HELLO, GOODBYE]
                               [LADY MADONNA] [HELP!] ]
```

>> Predict the results of these commands before trying them:

```
PRINT MEMBERP "LOGO [BASIC LOGO FORTRAN APL PASCAL]
PRINT MEMBERP [JOHN LENNON] [ [JOHN LENNON] [PAUL MCCARTNEY]
                              [GEORGE HARRISON] [RINGO STARR] ]
PRINT MEMBERP [JOHN LENNON] [I AM A JOHN LENNON FAN.]
```

>> Edit procedure QUEST to change line [3] to this:

```
IF MEMBERP REQUEST :ANSWER [PRINT [YOU'RE RIGHT!] STOP]
```

>> What is changed from the old version of line [3]?

>> Try your new QUEST with questions and multiple answers of your choice. Give it both correct and incorrect answers. What happens? Is there anything you'd like to change about this program's behavior? If so, do it!

LOGO-AL11: Subprocedures

A few units ago, we used a procedure named QUIZ which consisted of several command lines each of which used the procedure QUEST. When one procedure uses another like that, we say that the second is a subprocedure of the first. In this example, QUEST is a subprocedure of QUIZ.

Actually, you've been using subprocedures all along, because things like PRINT which are built into Logo are also procedures, and you've used them as subprocedures of things you've written.

>> What are all the subprocedures of QUIZ1?

_____ _____ _____

_____ _____ _____

Local variables. When you use subprocedures with inputs, the computer may be "in the middle of" several of your procedures at once. That is, when you use QUIZ, and QUIZ uses QUEST, at a given moment the computer may be up to the line of QUIZ containing SAN FRANCISCO; that line tells the computer to carry out procedure QUEST, and the computer is also at some line of that procedure. Each of these procedures has its own variables, which are independent of the other procedures. Even if two procedures use variables with the same name, the variables are separate. These variables are called local to a particular procedure. Consider this example:

```
TO SEPARATE :WORD
JOIN FIRST :WORD BUTFIRST :WORD
PRINT SE [YOU GAVE ME THE WORD] :WORD
END

TO JOIN :LETTER :WORD
PRINT SE [THE FIRST LETTER WAS] :LETTER
PRINT SE [THE REST OF THE WORD WAS] :WORD
END
```

>> After defining the procedures SEPARATE and JOIN, suppose you type the command

SEPARATE "CHECKS

>> What is the value of the variable WORD in procedure SEPARATE? _____

>> What is the value of the variable WORD in procedure JOIN? _____

The value of the variable WORD in procedure SEPARATE is not affected by the fact that the subprocedure JOIN happens to use a variable with the same name.

LOGO-AL12: Writing Operations

>> What does this command do?

```
PRINT FIRST BUTFIRST [THIS IS A TEST.]
```

If you often need to extract the second word of a sentence, it would be convenient to have an operation SECOND which you could use like FIRST. Regrettably, Logo does not have a built-in SECOND operation. But it's easy to write one:

```
TO SECOND :OBJECT
OUTPUT FIRST BUTFIRST :OBJECT
END
```

Until now, the procedures you've written have all been commands. Recall that a command is a procedure which does not have an output. When it is used, it does something which is complete in itself, for example, printing something on your terminal or moving a turtle around. An operation, on the other hand, produces an output which can be used as the input to another procedure.

>> What operations are used in the following?

```
PRINT FIRST BUTFIRST [THIS IS A TEST.]
```

>> What commands are used?

When you write an operation, you must indicate what object (word or list) you would like to be the output from your procedure. You do this by using the command OUTPUT, as in the procedure SECOND shown above. You must give OUTPUT an input (sorry if that sounds weird!) which is the thing you want your procedure to have as its output.

>> Experiment with the following procedure which we've included in the LOGO.AL file:

```
TO HOWIS :WHATEVER
IF EQUALP FIRST :WHATEVER "W [OUTPUT "WONDERFUL]
IF EQUALP FIRST :WHATEVER "T [OUTPUT "TERRIFIC]
OUTPUT "ORDINARY
END
```

Remember, this is an operation, so you have to do something with its output, like this:

```
PRINT HOWIS "SCHOOL
```

>> Write an operation THIRD which will output the third letter of a word, or the third word of a sentence.

(over)

You can also write your own predicates. For example, here is one which decides whether or not a letter is a vowel:

```
TO VOWELP :LETTER
IF MEMBERP :LETTER [A E I O U] [OUTPUT "TRUE]
OUTPUT "FALSE
END
```

Remember that any operation you write must output something or other no matter what it gets as input; if it's a predicate, the output must be either the word TRUE or the word FALSE.

>> Is the letter Y a vowel, according to VOWELP? Should it be? What can you do about this?

>> In Pig Latin, words starting with vowels are translated differently from words starting with consonants. Write a predicate FIRSTVP which takes a word as input and outputs TRUE if that word starts with a vowel, FALSE otherwise.

>> Write a predicate SAMELENP which takes two words as inputs, and outputs TRUE if and only if the two words have the same number of letters.

By the way, here is another way we could have written VOWELP:

```
TO VOWELP :LETTER
OUTPUT MEMBERP :LETTER [A E I O U]
END
```

>> Do you agree that this definition is equivalent to the other one?

LOGO-AL13: Recursion

We are going to write a procedure called DOWNUP, which will do this:

```
?DOWNUP "HELLO
HELLO
HELL
HEL
HE
H
HE
HEL
HELL
HELLO
```

That is, the procedure will take a word, drop letters one by one until it works its way down to a single letter, then replace the letters one by one, working up to the full original word. What makes this difficult is that the program must "remember" the original word while it is chipping away at the letters.

There is a very clever, but somewhat hard to understand, way to do this problem. To see how it works, let's start by writing a ridiculously simple special case of the program, which only works when the word is exactly one letter long:

```
[1] TO DOWNUP1 :WORD
[2] PRINT :WORD
[3] END

?DOWNUP1 "H
H
```

No doubt, this seems like a silly, roundabout way to carry out a PRINT command. But if you stay with us, the reason for it will become clear. The next step, as you might have guessed, is to write a version which works for two-letter words. We could write it this way:

```
[1] TO COULD2 :WORD
[2] PRINT :WORD
[3] PRINT FIRST :WORD
[4] PRINT :WORD
[5] END
```

(Convince yourself that that would work.) But instead, we propose to write it like this:

```
[1] TO DOWNUP2 :WORD
[2] PRINT :WORD
[3] DOWNUP1 BUTLAST :WORD
[4] PRINT :WORD
[5] END
```

(Again, convince yourself that this procedure will do what's desired for a two-letter word as input.) Once more, this may seem like a complicated way to do something simple. But compare these two approaches to the problem of three-letter words:

(over)

```
[1]  TO COULD3 :WORD
[2]  PRINT :WORD
[3]  PRINT BUTLAST :WORD
[4]  PRINT FIRST :WORD
[5]  PRINT BUTLAST :WORD
[6]  PRINT :WORD
[7]  END
```

```
[1]  TO DOWNUP3 :WORD
[2]  PRINT :WORD
[3]  DOWNUP2 BUTLAST :WORD
[4]  PRINT :WORD
[5]  END
```

See what's happened? We could explicitly extract the first two letters, and then the first letter, of the three-letter input. But the second way gives a shorter program, because it uses the two-letter version as a subprocedure. (If you aren't completely comfortable with the idea of subprocedures, review Units LOGO-AL8 and LOGO-AL11 before continuing here.)

>> What happens if you give DOWNUP3 an input with more than three letters? Less than three letters?

>> Write procedures COULD4 and DOWNUP4 to handle a four-letter word in a manner analogous to COULD3 and DOWNUP3. The COULD4 procedure should have nine lines, and the DOWNUP4 procedure five lines. Get them both to work before going on. It might be a good idea to show them to a teacher before continuing.

You could now solve the original problem by writing DOWNUP5 and applying it to the input "HELLO. But what if we want to do

```
DOWNUP "ANTIDISESTABLISHMENTARIANISM
```

? You'll get very tired of typing before you have all those subprocedures defined. You'll need through DOWNUP28, not to mention having to count the length of the word by hand to know which procedure to use! But there should be a way to take advantage of the fact that all those DOWNUP procedures are almost identical, differing only in the name of the procedure called in line [3]. Here's the secret: We can write just one procedure and let it use ITSELF as a subprocedure! A procedure which uses itself as a subprocedure is called a recursive procedure, and that technique of programming is called recursion. Writing a recursive version of DOWNUP is almost, but not quite, as simple as this:

```
[1] TO DOWNUP :WORD
[2] PRINT :WORD
[3] DOWNUP BUTFIRST :WORD
[4] PRINT :WORD
[5] END
```

This is the same form as all the previous special-case versions like DOWNUP3, except that there is only one procedure, and it uses itself (rather than the next lower numbered version) in line [3]. If you try applying this procedure to "HELLO, though, you'll find that there is a slight problem.

>> Can you analyze why it doesn't work, before reading further?

LOGO-AL14: DOWNUP continued

In the version with several numbered DOWNUP procedures, the procedure named DOWNUP1 was special. It didn't call any subprocedures, because once you're down to a single letter, there is no more cutting down to do. In the recursive version, we must have a way to tell that the input is only a single letter, and to stop the recursion at that point. Edit DOWNUP to add line [3] below:

```
[1] TO DOWNUP :WORD
[2] PRINT :WORD
[3] IF EMPTYP BUTLAST :WORD [STOP]
[4] DOWNUP BUTLAST :WORD
[5] PRINT :WORD
[6] END
```

The IF command has as its first input the predicate expression

```
EMPTYP BUTLAST :WORD
```

and uses that condition to decide whether or not to execute the STOP command. If EMPTYP outputs FALSE, the STOP command is ignored. So line [3] in DOWNUP says this:

> Take the butlast of :word (that is, all but the last letter).
> See if that's empty (that is, see if :WORD has only one letter).
> If so, stop.

The command STOP tells Logo to stop doing whatever procedure it is currently executing. If this procedure was started up by another procedure, Logo stops the subprocedure and resumes the outer procedure (the one it was doing first).

Line [3] does nothing if the thing in variable WORD (the input to the procedure) has more than one letter. In that case, EMPTYP has the value FALSE. But if the input has only one letter, then its BUTLAST is empty, so EMPTYP has the value TRUE and the STOP command is executed.

>> If you're not completely satisfied that you understand all about how DOWNUP works, stop right now and ask a teacher for help.

>> Write procedures DOWN and UP which only do half of what DOWNUP does, i.e.,

```
?DOWN "HELLO
HELLO
HELL
HEL
HE
H
?UP "HELLO
H
HE
HEL
HELL
HELLO
```

LOGO-AL15: More About Recursion

One way of looking at the DOWNUP procedure is that it does something to each letter of its input in turn. There are many cases where you might want to do that. For example, suppose you want to take a sentence and print it out, one word per line instead of all on one line. Here's how:

```
[1] TO ONEPER :SENT
[2] IF EMPTYP :SENT [STOP]
[3] PRINT FIRST :SENT
[4] ONEPER BUTFIRST :SENT
[5] END
```

(You should try this out on a few examples. From now on, we won't mention that, although it's always a good idea to try out a new procedure to make sure you understand it.) As in DOWNUP, there must be an end test (line [2]) so that the procedure won't try to keep taking the BUTFIRST after it's gone through the entire sentence. Line [3] is the one which really tells what the procedure is supposed to do: print one word of the input on a line by itself. Line [4] is the recursion step, which applies the same procedure to a smaller input. Many other procedures could be written which would have the same form as this one, but would do something different because line [3] would be different:

```
TO BACKTYPE :SENT
IF EMPTYP :SENT [STOP]
PRINT LAST :SENT
BACKTYPE BUTLAST :SENT
END

TO VOWLIST :WORD
IF EMPTYP :WORD [STOP]
IF VOWELP FIRST :WORD [PRINT FIRST :WORD]
VOWLIST BUTFIRST :WORD
END
```

>> What do these do?

>> Write a procedure which takes a sentence as input, and prints only those words in the sentence which start with the letter E.

>> Write a procedure which takes two inputs, a sentence and a letter, and prints only those words in the sentence which start with the given letter.

(over)

Another kind of recursive procedure uses a number to count down, instead of using BUTFIRST or BUTLAST to make the input smaller:

```
TO SAYOVER :WORD :NUMBER
IF EQUALP :NUMBER 0 [STOP]
PRINT :WORD
SAYOVER :WORD :NUMBER-1
END

TO LAUNCH :NUMBER
IF EQUALP :NUMBER 0 [PRINT [BLAST OFF!!] STOP]
PRINT :NUMBER
LAUNCH :NUMBER-1
END
```

>> Write a procedure PRINTNTH which takes two inputs, a number and a sentence; it should skip over the first n-1 (if n is the number) words of the sentence and then print the nth word. (Hint: notice that LAUNCH prints something special on its last invocation, by putting the PRINT command after the end test.)

LOGO-AL16: Recursive Operations

So far, you have only written recursive commands, which print something but have no output to pass on to another procedure. You can also write recursive operations. These programs are complicated by the fact that each recursive use of the procedure must pass an output to the next higher version of the same procedure; the recursion step is typically in an OUTPUT statement. For example, just as FIRST outputs the first letter (or word) of a word (or sentence), and in Unit LOGO-AL12 we wrote an operation SECOND which outputs the second letter or word, here is an operation ITEM which takes a number and a word and outputs the letter selected by the number input:

```
[1] TO ITEM :NUMBER :OBJECT
[2] IF EQUALP :NUMBER 1 [OUTPUT FIRST :OBJECT]
[3] OUTPUT ITEM :NUMBER-1 BUTFIRST :OBJECT
[4] END
```

The ITEM procedure uses the output from the lowest-level use as the output from all the higher-level uses, because line [3] simply says to output the result of the next lower use of ITEM. Some procedures use the output of lower levels as only part of the output from a higher level:

```
TO REVERSE :WORD
IF EMPTYP :WORD [OUTPUT "]
OUTPUT WORD (REVERSE BUTFIRST :WORD) (FIRST :WORD)
END

TO HOWMANY :OBJECT
IF EMPTYP :OBJECT [OUTPUT 0]
OUTPUT 1 + HOWMANY BUTFIRST :OBJECT
END
```

The procedure HOWMANY is simply a rewriting of the Logo primitive procedure COUNT. You won't usually want to rewrite primitives, but the same pattern can be used for counting something other than simply the size of an object. For example, suppose we want to count the number of times the letter W is used in a word:

```
[1] TO WCOUNT :WORD
[2] IF EMPTYP :WORD [OUTPUT 0]
[3] IF EQUALP FIRST :WORD "W [OUTPUT 1 + WCOUNT BUTFIRST :WORD]
[4] OUTPUT WCOUNT BUTFIRST :WORD
[5] END
```

This procedure has two different recursion steps. If the procedure has found a W, line [3] adds one to the W-count of the rest of the word. If not, line [4] passes along the W-count of the rest of the word, unmodified.

(over)

>> Write a procedure to count the number of vowels in a word.

>> Write a procedure to count the number of words in a sentence which contain the letter W one or more times.

>> Write a procedure to count the total number of times the letter W occurs in any word of a sentence. (This count will be different from the previous problem if there are any words in the sentence with more than one W.)

LOGO-AL17: An Example--Transliteration

A while ago, the Chinese government introduced a new system for transliterating Chinese words into our alphabet, so that what used to be Peking is now Beijing. An obvious practical use of a computer is to change an old-style transliterated text into the new system. The rules for this Chinese conversion are very complicated, so instead of writing a program to deal with all of them we'll take just one example, and write a program which takes a sentence as input and changes every letter P into the letter B.

Before dealing with an entire sentence, let's take the simplest part of the problem, and write a procedure which will take a single letter as input, and output the same letter unless it's a P, in which case it'll output B:

```
TO PBLETTER :LETTER
IF EQUALP :LETTER "P [OUTPUT "B]
OUTPUT :LETTER
END
```

This procedure isn't recursive, since it only deals with a single letter. If the input letter isn't P, the output is the same as the input.

Now let's write a recursive procedure to transliterate an entire word:

```
TO PBWORD :WORD
IF EMPTYP :WORD [OUTPUT "]
OUTPUT WORD (PBLETTER FIRST :WORD) (PBWORD BUTFIRST :WORD)
END
```

The parentheses aren't really necessary in this program, but are used to make clear to you what are the inputs to the WORD procedure. The way PBWORD works is to chop up its input into two parts, the first letter, and the rest of the word. The first letter is sent through PBLETTER, and the rest of the word is sent through PBWORD itself, recursively. The results of these two operations are then recombined into a transformed version of the original input word.

The procedure to transform a sentence is based on the procedure to transform a word, in the same way that the procedure for words is based on the procedure for letters:

```
TO PBSENT :SENT
IF EMPTYP :SENT [OUTPUT []]
OUTPUT SE (PBWORD FIRST :SENT) (PBSENT BUTFIRST :SENT)
END
```

>> Write a procedure which transforms a sentence by changing every vowel to a letter X:

```
?PRINT XVOWEL [IT'S TOO HOT IN THE COMPUTER CENTER.]
XT'S TXX HXT XN THX CXMPXTXR CXNTXR.
```