Accompanying Text

for

***On Logo* Videotape**

Hurdles 1

Logo Grammar

by Harry Nelson and Wyn Snow

Table of Contents

1. The Transactional Model

When most people begin to learn Logo, they make use of a linguistic model of Logo. They think about "talking" to the turtle, about "making requests" or "giving instructions" to Logo. One "asks" or "tells" the turtle to go forward 50 steps (FD 50). One asks or tells Logo to clear off the screen in preparation for new work.[1])

This linguistic model for thinking about Logo can take us quite far into Logo, but sooner or later we begin encountering situations -- usually certain bugs -- that require a new model for how we think about Logo.

On the tape, PRINT CS is used to introduce the need for this new transactional model. From the linguistic point of view, this is not a grammatical Logo instruction -- as "ain't ain't English," PRINT CS just isn't Logo. But if that's all there was to say on the matter, how would it be possible for Logo to do anything at all with the instruction? Remember (or even better, test it yourself) that the screen is cleared before the error message appears that PRINT didn't get an input from CS. If PRINT CS is not a proper Logo instruction, how does this happen?

The answer is that Logo does not "look" at the whole instruction to see whether or not it is a grammatically proper Logo instruction. Logo simply starts trying to do what the instruction says and continues until a problem arises.

The linguistic model would lead one to view Logo as saying, "That's nonsense. It can't be done." With the transactional model, one does not think of Logo having such global concerns. Instead, one views Logo as calling up a PRINT agent or actor who needs an input. PRINT sees a procedure name in place of its input and calls up a procedure actor (CS in this case). CS does its job of cleaning any graphics from the screen and returning the turtle to its home position (in most versions of Logo). But CS does not output anything that PRINT can write on the screen. With the transactional model, one views Logo as starting a process which

[1]CLEARSCREEN or CS in IBM Logo and Apple Logo; DRAW in Terrapin Logo; CG in LogoWriter.

goes as far as it can. When CS finishes without giving an input to PRINT, things have gone as far as they can -- and the error message is then displayed.

Try a few different exercises. Enter the instruction PRINT HT and see what happens. Now use the transactional or actor model to think about and explain what took place.

The explanation should go something like this: When Logo got the instruction, it called the PRINT actor to do its job. The PRINT actor looked for its input but found the HT command in its place, so it called the HT actress. She did her job and told the PRINT actor when she had finished -- but she didn't give him any input. PRINT, who can't do anything worthwhile without an input, complains that he didn't get an input from HT or, more precisely, that HT did not output anything to PRINT.

Try to do the same thing with each of the following instructions -- you might try to predict what Logo will do before you enter them:

```
PRINT SETBG 4
PRINT HT + CS
PRINT 1 + CS
```

* * *

The following story about a bug involving the Logo READLIST instruction[2] also shows the difference between the linguistic and the transactional ways of thinking about Logo.

A young programmer wanted her new game program to begin by asking the user if he or she wanted to play. The plan was that if the user entered YES, the PLAY program would be run. If the user entered NO, a GOODBYE procedure would be run. Here is her procedure:

[2]REQUEST in Terrapin, READLISTCC in LogoWriter.

```
TO GAME
PRINT [DO YOU WANT TO PLAY?]
IF RL = [YES] [PLAY]
IF RL = [NO] [GOODBYE]
END
```

The programmer showed off her program by first typing YES and the appropriate thing happened. The computer played the game. She then showed what happened if you answered NO. Nothing happened, so she typed NO again, saying with a smile, "It doesn't want to take no for an answer." You see a bug here that you can almost live with, at least in this case.

But try to imagine what was in the child's mind. She thought about the program linguistically; "If you type YES it will play the game but if you type NO it will say goodbye." The point is that if you read Logo as if it were English, you might read it incorrectly.

The transactional model asks you to imagine a process rather than to translate Logo into English. The GAME procedure will run three separate lines of instructions. The first, the PRINT instruction, contains nothing relevant to the present discussion. GAME then runs the first conditional (IF) instruction. Note that this instruction refers only to YES. READLIST makes the computer wait until the user types something and then the procedure checks to see if what was typed is YES. If it was not YES, the procedure goes on to the next line, the second conditional instruction. This instruction refers to NO. READLIST waits for the user to type something and the instruction checks to see if what was typed is NO.

The reason that the programmer had to type NO twice was not that Logo really didn't want to take no for an answer. Rather, the two READLISTs each got their own responses from the user. The first was used only to check for YES; anything else was ignored. The second READLIST is used to check for NO. Think about it or, even better, act it out.

Here is a version of GAME that eliminates the "double negative" bug. Can you explain why this is so with the help of the transactional model of Logo?

```
TO GAME
PRINT [DO YOU WANT TO PLAY?]
NAME RL "REPLY
IF :REPLY = [YES] [PLAY]
IF :REPLY = [NO] [GOODBYE]
END
```

If you are unsure of the proper use of NAME -- of using Logo naming and names -- don't worry! The second Hurdles tape deals with naming; for now it's enough to know that the line

```
NAME RL "REPLY
```

will give the correct :REPLY (whatever the user types) to the = testor.

* * *

The COMPASS procedure (acted out on the tape to demonstrate the transactional model) could be useful in a "microworld" in which the turtle can only turn 90 degrees at a time. In constructing such a microworld, we might define the following R and L procedures for right and left turns.

```
TO R
RT 90
END

TO L
LT 90
END
```

Try to extend this idea to allow turns that are multiples of 90 degrees -- 0, 90 180, 270, etc. degrees. The R and L procedures could be modified to take inputs. For example, R 3 would make the turtle do the equivalent of three 90-degree right turns. L 2 would turn the turtle left 180 degrees -- two 90-degree left turns. Our procedures could look like this:

```
TO R :NUMBER
RT :NUMBER * 90
END

TO L :NUMBER
LT :NUMBER * 90
END
```

In such a microworld, PRINT COMPASS would show us the turtle's heading as NORTH, SOUTH, EAST or WEST.

```
TO COMPASS
IF HEADING = 0 [OP "NORTH]
IF HEADING = 90 [OP "EAST]
IF HEADING = 180 [OP "SOUTH]
IF HEADING = 270 [OP "WEST]
END
```

Use the transactional model to describe what the following procedure will do before running it, then compare your predictions with what actually happens when you run it. Try to account for any discrepencies.

```
TO SQU
FD 60
R 1
IF COMPASS = "NORTH [STOP]
SQU
END
```

Try the same exercise with this procedure:

```
TO SQUIRAL :SIDE
IF :SIDE > 100 [STOP]
FD :SIDE
R 1
SQUIRAL :SIDE + 10
END
```

Other exercises you might try are extending the microworld to allow turns that are multiples of 30 degrees and writing more procedures in this microworld similar to

SQU and SQUIRAL. What would happen to procedures like SQU if COMPASS had been written with PRINT in place of OUTPUT?

* * *

Describe the transactions that take place when you run:

```
SETBG BG + 1
```

In most cases, the background color of the screen should change to a new color. In some versions of Logo, there are instances in which SETBG BG + 1 will result in an error message instead of a new background color. The following procedure guarantees that this will not happen.

```
TO NEXTCOLOR
IF BG = 15 [OP 0]
OP BG + 1
END
```

NOTE: In this procedure, the assumption has been made that there are 16 background colors numbered 0 through 15. If that is not the case for your Logo, replace the 15 in the procedure with the highest color number used in your Logo.

What will happen when you run these next instructions?

```
SETBG NEXTCOLOR
PR NEXTCOLOR
SETH NEXTCOLOR
PR NEXTCOLOR + 10
```

You might enjoy trying and thinking or acting the following:

```
TO FLASHBG
SETBG NEXTCOLOR
WAIT 10
FLASHBG
END
```

NOTE: In the actor or transactional model, we sometimes make certain assumptions about the actors' abilities. In its purest form, the actor model should result in a new actor appearing every time a procedure call is made -- regardless of whether that procedure is a Logo primitive or a user-defined procedure. However, this easily becomes tedious and often distracts from the points at issue.

In these tapes, we have adopted the view that it is best to be precise on the issues at hand and loose on others. So, for example, we often assume that actors can "make primitives happen" without bothering to call a separate actor for each primitive. Also, in an effort to avoid being overly pedantic, we have given actors a certain liberty or dramatic license. While Logo procedures are "dumb" and follow exactly the text of the procedure definition, important points are often made by allowing the actors the freedom to express their views of given transactions.

2. Remodelling Primitives

When you understand how to remodel a Logo primitive, you understand how that primitive works. You also gain a deeper understanding of how Logo primitives interact with one another. This remodelling -- or reworking -- personalizes Logo as well. It's a way of making Logo uniquely your own.

Your Hurdles 1 diskette has 11 procedures that were illustrated on the videtape: RTT and LTT (TURN), HOP, APPEAR, VANISH, RPT, HDG, QUIT, WANDER, SILLY and SILLIER. Play around with these procedures so you can see how they work, and then make some of your own.

Commands:

There are 4 different types of primitives. Everyone meets commands (or doers) first, so we'll start with those.

```
FORWARD 50
CLEARSCREEN
PRINT [THESE WORDS]
```

Each of these is a complete instruction. When you type them at the keyboard or put them into a procedure, Logo and the turtle know exactly what to do. We call FORWARD and CLEARSCREEN and PRINT *commands* or *doers* because they say what to do.

One of them, CLEARSCREEN, can stand alone. It doesn't need any additional information; it knows how to clear the screen. But FORWARD and PRINT need an input. We have to say *how far* to move the turtle, or *what to print* on the screen. So a command -- together with whatever input(s) it needs -- is a complete Logo instruction.

In remodelling a command, you may want to change the way it handles its input. This is one reason for creating RTT, LTT and HOP. Or you may simply want to use a different name, either an abbreviation or something you prefer. APPEAR, VANISH and RPT were created for these reasons.

Reporters:

VANISH is an excellent model for reworking a command because it is so simple, straightforward, and clear. But when we tackle reporters, the footprints of a distinctive "hot potato" bug quickly appear.

```
TO VANISH                     TO HDG
HT                            HEADING
END                           END
```

When we run VANISH, all is well. The turtle disappears. But if we use VANISH as a pattern for remodelling HEADING, an error message appears when we run HDG -- something that shows that the turtle's heading is a "hot potato". A similar message appears when we try PRINT HDG.

Naturally, we could "fix" this bug by putting PRINT in HDG. Let's use a different title for this modified procedure, say PR.HDG.

```
TO PR.HDG
PR HEADING
END
```

This procedure will run, but it is limited. The *only* thing PR.HDG can do with the heading is print it. HEADING is a reporter. It reports a number back to whomever called it, so we can say:

```
PRINT HEADING                   or
LEFT HEADING * 18               and even
BACK HEADING                    or
SETHEADING RANDOM HEADING
```

So PR.HDG is not as versatile as HEADING. Substituting either HDG or PR.HDG for any of these HEADINGs will result in an error message.

The problem is quite simple. Any Logo instruction must start with a command. A reporter cannot stand alone; it must have someone to give its report to. In HDG, HEADING needs a procedure to report to, and the procedure it needs is OUTPUT, OP for short.

```
TO HDG
OP HEADING
END
```

OP needs one input. Here, HEADING provides that input, so OP HEADING *is* a complete Logo instruction. OP takes the report from HEADING and tells HDG to give that number to whomever called HDG -- perhaps PRINT or LEFT or BACK or SETHEADING.

Reporters are powerful because of this versatility; they can be used in conjunction with many different procedures. Remodelling Logo's reporter primitives and creating one's own reporters are important avenues for understanding how Logo procedures work -- both the built-in primitives and the ones you create. Roughly half of Logo's primitives are reporters.

Controllers:

As you saw on the videotape, however, OP is much more than simply a command. Whenever OP becomes active in a procedure, it takes charge of what happens next. We call it a controller. OP says, "Drop everything else and do what I say." This was illustrated by the actors who did COMPASS in the first section of the videotape.

```
TO COMPASS
IF HEADING = 90 [OP "EAST]
IF HEADING = 180 [OP "SOUTH]
IF HEADING = 270 [OP "WEST]
IF HEADING = 0 [OP "NORTH]
END
```

When the heading is 180, = reports TRUE and IF runs [OP "SOUTH]. This OP tells COMPASS to ignore the rest of its instructions and to hand the word SOUTH to whomever called COMPASS. This changed the usual "flow of control." Normally, a procedure goes through its instructions line by line, instruction by instruction, until it reaches the END. OP and STOP both interrupt this process, telling the procedure they are in to "drop everything else."

```
TO WANDER
FD 10
RT RANDOM 30
IF HEADING = 0 [STOP]
WANDER
END
```

WANDER is recursive. It will go on forever unless the turtle's heading becomes 0. (Recursion and "forever programs" are dealt with in the Hurdles 3 tape.) The flow of control starts with the first line: FD 10. When this instruction has been carried out, WANDER goes on to the next line. When RT RANDOM 30 is done, WANDER goes on to IF HEADING = 0 [STOP]. Note that = is a reporter.

The odds are 359 to 1 that = will report FALSE. Most of the time, IF will not run its instruction list. WANDER will then run the next line -- which is WANDER. Sooner or later, however, = will report TRUE and IF will run [STOP]. STOP will tell WANDER to "Drop everything! You're done." So when STOP becomes active, WANDER doesn't reach that next line and the WANDERing STOPs. It's just like the FD 1000 in SILLIER.

```
TO SILLY                        TO SILLIER
RT 90                           RT 90
OP 90                           STOP
FD 1000                         FD 1000
END                             END
```

Controllers cannot be remodelled. QUIT is a perfectly legitimate procedure, but it cannot stop either WANDER or SILLIER.

```
TO QUIT
STOP
END
```

When QUIT is run, it starts with the first line of instructions and calls STOP. STOP says "Okay, QUIT, drop everything. You are done." QUIT then tells WANDER or SILLIER that it is done, so they proceed to their next lines of instruction: WANDER or FD 1000. STOP can only stop the procedure it is in.

Chameleons:

A few procedures can be called chameleons because they can act either like commands or like reporters, depending on the context. What they do is determined by their instruction list. We have already seen the chameleon flavor of IF. Let's take it one step further.

Look at the following whimsical instruction:

```
IF 1 = 1 [PRINT "HI]
```

Here, IF does not output anything. It is not a reporter. But you can do

```
PRINT IF 1 = 1 [HI]
```

A more serious example is the following:

`PR IF 1 = RANDOM 1 [YES] [NO]`	Apple & IBM Logo
`PR IFELSE 1 = RANDOM 1 [YES] [NO]`	LogoWriter
`PR IF 1 = RANDOM 1 THEN YES ELSE NO`	Terrapin Logo

The other chameleon is RUN. If RUN is an obstacle for your students, you can tell them it is just like REPEAT 1 [*sundry actions*]. RUN also takes an instruction list, but it performs that list only once. And like IF, the instruction list will determine whether RUN behaves like a reporter or a command.

3. DRAW A RED SQUARE

DRAW is a primitive in Terrapin Logo, so Terrapin users must use another word (such as SKETCH) for the following project.

Part III of the tape introduces a way to make Logo understand a few words of everyday English. Developing and extending this project encourages thinking about how the computer handles the Logo language -- and, perhaps, how we handle our own language.

As you have seen, the Logo line DRAW A SQUARE uses the following procedures:

```
TO DRAW :INPUT
RUN :INPUT
END
```

DRAW needs a list of instructions as input which will be run (executed or made to happen). DRAW looks to the next procedure in the Logo line (A) for its input.

```
TO A :INPUT
OP :INPUT
END
```

A needs an input from the procedure to its right. When it gets this input, A outputs it to the procedure to its left (DRAW).

```
TO SQUARE
OP [REPEAT 4 [FD 50 RT 90]]
END
```

SQUARE simply outputs the list of instructions needed to draw a square. It outputs this list to the procedure to its left (A). Using a color adjective, as in the instruction DRAW A RED SQUARE, requires another procedure:

```
TO RED :INPUT
OP SE [SETPC :RED] :INPUT
END
```

RED gets an input from the procedure to its right and it outputs a sentence (list) made up of the SETPC instruction and its input. RED modifies SQUARE's output and hands the modified list to its left (to A).

Different versions of Logo have different commands to change the turtle's pen color: SETPC or SETC or PC. If you are not sure, check to see which your Logo uses. Also, few Logos have built-in color names like :RED; this has been used because the actual color number for red is different in the various Logos. Use your color number for red in place of :RED or, if you know how, create names for your color numbers and use :RED. (The topic of naming is treated in the next tape in this series.)

Creating other shape nouns is not very difficult if you know the instructions needed to make the shapes. For example, CIRCLE could output the list

```
[REPEAT 180 [FD 1 RT 2]]
```

and TRIANGLE could output the list

```
[REPEAT 3 [FD 50 RT 120]]
```

But what about other shapes or figures? What lists of instructions would be needed for a star, a house, or other shapes you may want to draw?

* * *

Creating other color adjectives is even easier. Try making BLUE, GREEN and WHITE procedures that act like RED.

You can tell Logo to draw a red square, then a blue circle. If you next tell it simply to draw a triangle, what color triangle will appear? Blue, of course. But why? To someone who hasn't seen the program, it might look like you taught Logo to remember the last color used. That is just what you did. When you set the turtle's pen color it keeps that color until a new pen color is set.

Because you can use DRAW followed a different number of other procedure names, someone who hasn't seen the entire project could think that you have done the impossible. It might look like you've written a procedure that takes a variable number of inputs, two or three or whatever. But that cannot be done in Logo. (In fact, one reason DRAW was developed was to get around this impossibility.)

On the tape you saw a position or direction specified. In the instruction

```
DRAW A TILTED BLUE SQUARE
```

all the procedures are familiar except TILTED. What does it do?

It should get an input from BLUE and hand its output to A. TILTED should also put a command to turn the turtle into the instruction list that it outputs. The color procedures are the model to use here. TILTED will output a sentence (list) of the command RT 45 and its input.

```
TO TILTED :INPUT
OP SE [RT 45] :INPUT
END
```

If TILTED gets the input list

```
[SETPC :BLUE REPEAT 4 [FD 50 RT 90]]
```

it will output the list

```
[RT 45 SETPC :BLUE REPEAT 4 [FD 50 RT 90]]
```

There is a small bug in here which you may want to find on your own. If so, spend some time playing with the vocabulary that you have already created.

* * *

If you use TILTED for one figure, all of the following figures that you draw will be tilted even though you don't specify it. Having Logo remember the "tilt" may seem all right; a little like its remembering a color. But try using TILTED in two successive instructions -- or in three, four, or five new instructions.

Progressive tilting is probably not what you have in mind if you ask for a tilted square, a tilted star and then a tilted triangle. One way to fix the progressive tilting is simply to replace the RT 45 with SETHEADING 45. This is a sensible change to make and a good opportunity to think about the differences between the primitives RT and SETHEADING.

The other issue, of having everything tilted after using the word once, is also easily fixed. You can change DRAW so that the turtle always returns to its home position and heading after the input list has been run.

```
TO DRAW :INPUT
RUN :INPUT
PENUP HOME PENDOWN      The effect of this new line should be clear.
END
```

This change in DRAW will be helpful in the next phase of extending the project as well as for clearing up the "tilt" bug.

In order to introduce number words like TWO and THREE into the project vocabulary in a meaningful way, it will be necessary for the turtle to change its position before drawing each new figure. If it doesn't move, the turtle will simply retrace its steps two or three times -- only one figure will be seen.

A procedure that sets a random position for the turtle can be a helpful "tool" at this point.

```
TO SET.RANDOM.POS
PENUP
SETPOS SE (150 - RANDOM 300) (100 - RANDOM 200)
PENDOWN
END
```

> This command will set the turtle's position somewhere between 150 and -150 right or left and between 100 and -100 up or down.

One way to use this is to make SET.RANDOM.POS the first command in the instruction list output by each shape procedure. SQUARE, for example, would then output the list [SET.RANDOM.POS REPEAT 4 [FD 50 RT 90]].

It might be more interesting to add RANDOMLY PLACED to the vocabulary, making possible a line like

```
DRAW A RANDOMLY PLACED BLUE CIRCLE
```

PLACED can just output its input like A does:

```
TO PLACED :INPUT
OP :INPUT
END
```

RANDOMLY would add SET.RANDOM.POS to the instruction list:

```
TO RANDOMLY :INPUT
OP SE [SET.RANDOM.POS] :INPUT
END
```

You might try your hand at making a few procedures that have the figure drawn at a specified place. For example, HIGHLY and LOWLY would cause either a highly placed or a lowly placed figure to be drawn. (Hint: HIGHLY could add the following list of commands

```
[PU SETPOS [75 0] PD]
```

to the instruction list.)

After the turtle draws a randomly placed (or any other placed) blue circle, it should almost certainly return to the home position. By fixing the tilt bug, you've already seen to that.

Now for the numbers. What should an instruction like

```
DRAW TWO RANDOMLY PLACED RED SQUARES
```

do? Well, first of all, SQUARES should output the same list as SQUARE. That's easy:

```
TO SQUARES
OP SQUARE
END
```

Plurals for the other shapes are just as easily created.

The next thing that the line should do is make sure that the instruction list is, in effect, run twice. The easiest way to do that is to give DRAW a list that contains two sets of instructions for drawing a randomly placed blue square. So, TWO can look like this:

```
TO TWO :INPUT
OP SE :INPUT :INPUT
END
```

TWO outputs a list that contains its input list twice. THREE, as you may imagine, outputs a list that contains three sets of instructions for drawing the figure.

```
TO THREE :INPUT
OP (SE :INPUT :INPUT :INPUT)
END
```

Remember that SE expects two inputs -- so if you give SE either more or less than two inputs, you must also put parentheses around both SE and its inputs.

Notice that the squares in DRAW TWO SQUARES will be drawn on top of one another. If you say DRAW TWO RANDOMLY PLACED SQUARES, they will be in different places. One way of fixing this bug in TWO and THREE is to use REPEAT instead of SE -- and to insert a RT 180 or RT 120 at the end of REPEAT's instruction list. This strategy was used on the diskette.

Yet another extension of the DRAW project is to add size words so you can say:

```
DRAW THREE RANDOMLY PLACED SMALL GREEN STARS
```

A few changes will accomplish this. First, the input to FD in each shape definition will be changed to the name :SIZE. SQUARE, for example, will look like this:

```
TO SQUARE
OP [REPEAT 4 [FD :SIZE RT 90]]
END
```

(:SIZE replaces the number 50.) Note that it will not be necessary to do anything to the plural shape nouns. Only the singular nouns need be changed.

Where will the value of :SIZE be specified since SQUARE will not be given an input? You probably guessed that this will be done with adjectives like SMALL:

```
TO SMALL :INPUT
OP SE [NAME 20 "SIZE] :INPUT
END
```

SMALL inserts the command that gives 20 the name SIZE into the instruction list. BIG and MIDSIZED could, similarly, assign the name SIZE to 80 and 50.

Now instructions containing size adjectives -- such as the example above -- will work nicely. However, there is a little bug here. You might spend some time trying to find the bug before going on.

* * *

The bug, as you may have found, is a simple one. If the first line you enter does not contain a size adjective, you will see an error message because :SIZE does not have a value. Of course, you could live with this by being sure that you always use a size. But it's not hard to fix the bug so you don't have to worry about it. In fact, you have your choice of two ways to fix it.

The first way to fix the bug is to specify a "default" size in the DRAW procedure that will always take effect if another size is not specified.

```
TO DRAW :INPUT
NAME 50 "SIZE
RUN :INPUT
PENUP HOME PENDOWN
END
```

This will give :SIZE the value 50. But if a size is specified in the instruction list, this value will be overridden. In this way, a size is always specified and the error message will not appear. You will note, however, that whenever a size adjective is not used in the draw line, :SIZE will revert to its default value.

If you want Logo to remember the last size that you specified until you change it, try this way to fix the bug: Specify the default size in a STARTUP list instead of in DRAW. Leave DRAW as it was (without the NAME line) but enter this line, all by itself, before you save your procedures:

```
NAME [NAME 50 "SIZE] "STARTUP
```

Now when you load your DRAW project, this STARTUP list will be run automatically. :SIZE will have the value 50 until you change it with a size adjective. This new value will remain until you use another size adjective. (Remember that names and naming are covered in detail in the next tape.)

Finally, notice that redefining CIRCLE to accept size changes presents some interesting challenges. Using 80, 50 or even 20 as input to FD in CIRCLE will not be acceptable. Try to find a way for your CIRCLE procedure to modify :SIZE in order to produce appropriately sized circles for each value of :SIZE. Here is a sample:

```
TO CIRCLE
OP [REPEAT 180 [FD :SIZE / 50 RT 2]]
END
```

From here, you can continue to extend the DRAW vocabulary, create a whole new project using these ideas in another area, or you may wish to leave the project as is and use it as a basis for thinking about computer language and human language, about computer behavior and human behavior. Or you may just want to leave it for now and return to it some time in the future.