Accompanying Text

for

On Logo Videotape

Hurdles 4

Digging Deeper

by Wyn Snow

Table of Contents

1. Introduction

This tape presents two projects, each of which raises a number of issues about Logo. They also have a common thread: how to go about developing a relatively complex Logo problem. These two projects are both recursive, so they also shed additional light upon ways of using recursion.

In approaching any problem, the first step is to try to break it up -- the heuristic device of divide and conquer. Many Logo programs use this approach in a particularly simple form. For example, in the RIVER, HOUSE and CHURCH procedures, the final product can be seen as made up of several parts.[1] The way to simplify the overall task is to make the parts separately, each one as a subprocedure, and then put them together.

The complexity in such procedures is one of structure, of putting together many distinct pieces to create one overall superprocedure. This tape shows a different approach to the divide and conquer strategy, one that focuses on a procedure's own growth and development -- on the process rather than the structure.

In the two examples on this tape, the first step in our divide and conquer strategy is to make a simpler version of the whole. The next step elaborates this into a more complex solution by giving it additional functionality. The final version might well contain "bits and pieces" of the first version -- but not as separate parts that can be isolated and pointed to. Instead, a simple entity has grown into a more complex one. The first version is similar to a sketch that will lead to an oil painting. A simple line drawing captures much of the essence of the final canvas and provides a base upon which color and texture and shadings will grow.

2. Meters

Our first example is a recursive command, a procedure which does not give any output. Our goal is to connect these meters so that going one full circle triggers a TIC in the next dial. In many ways, the problems are the same: how to give a

[1]RIVER is in *New Mindstorms 2 & 3*, HOUSE is in *Hurdles 1* and CHURCH is in *Hurdles 2*.

unique identity to each different running of a procedure. How can we simplify it, find a simpler task that can be elaborated to create a more complex functionality? In this case, there is no return movement. Typically, all recursive procedures have this two-way process -- but we never reach the return motion if the recursive procedure never stops. If we put a STOP rule into this procedure, however, it *would* produce an invisible return or flow-back.

To get the most out of this meters exercise, look at enough of the tape to get started, then come back to the tape when you have completed the project or when you have gone as far as you can. What counts as "enough" depends entirely on you and your style. Some people like to get an overview of a problem before starting in on it. Some people want to try it themselves even before the problem has been started. It is very important to assert your own problem-solving style and to accept yourself well enough to do so without any sense of guilt or regret at not doing it in another style. You have to make a similar decision about when to read this text. If you are not familiar ~~with the~~ idea of multiple turtles, you will probably want to read the following note.

A Note on Multiple Turtles

```
TELL :TURTLE.NUMBER
```

TELL makes a particular turtle active. (In the multiple turtle microworld called TURTLES.EG on your diskette, only one turtle can be active at any one time.) When you say TELL 0, turtle 0 is the one that listens and carries out your commands. If you now say TELL 3, turtle 0 will become inactive and turtle 3 will start listening.

```
WHO
```

WHO is a reporter. It reports which turtle is listening, or as children would say, which turtle is "it". So one can say

```
FD WHO * 10
RT WHO * 30
TELL WHO + 1          and so on.
```

```
ASK :TURTLE.NUMBER :INSTRUCTION.LIST
```

ASK is a temporary TELL. It "remembers" which turtle is WHO, gives the :INSTRUCTION.LIST to :TURTLE.NUMBER, then restores the earlier WHO. While :TURTLE.NUMBER is doing the :INSTRUCTION.LIST, this :TURTLE.NUMBER is WHO, so one can say things like

```
ASK 2 [PRINT WHO]
ASK 1 [RT 360 / WHO]            as well as
ASK 3 [DRAW A RED SQUARE]
```

The exercise has several goals. The narrowest goal is to help you develop a deeper understanding of recursion by working a problem that is rather different from the ones in the recursion segment (Hurdles Tape 3) and the ones usually encountered in Logo books. A broader goal is to develop a sense of strategies in approaching a project of this kind, whether the project has to do with recursion or not. Other goals have to do with taking a syntonic approach to thinking about a project by relating it to other familiar situations and to personal experience.

The tape begins by giving some advice on tackling a project like this one. Here we state some of the advice in slightly different form.

Some General Advice on Thinking Strategies:

1. Get a good sense of what you are trying to do. In this case, you are trying to make a specific thing which is there on the screen for you to see. Immerse yourself in it. In projects that you initiate yourself, you usually do not have a sample of the product you want to make. In that case, you might try to visualize it as a mental game or as a physical model. For example, you could draw four rings of numbers on a sheet of paper and move match sticks to represent the hands of the dials. Maybe you should do that anyway even though you can see the thing on the screen. People often write programs to simulate a "real situation." A good strategy for developing a program is to use "real things" to simulate the computer program.

2. Open your mind to "associations" -- think of other situations that seem somehow similar to this one. The tape mentions a few that are very closely like it. Other, more far-fetched relationships might also be suggestive. They are also more idiosyncratic; one person might not relate to the same ones as another person. For example, I thought of a chain letter and the life cycle of a butterfly.

3. Decide which aspects of the project are unrelated to "the main issue" and put them aside. For example, the numbers around the dials are "decoration" or, at least, a separate project. The tape just ignores them. This does not mean there is nothing interesting there. It means that you should concentrate on one issue at a time.

4. Develop a language for talking about the problem. The words "round" and "action" are part of a language you can use in lots of Logo situations. The names TIC and TOC are special to this problem, part of a "private language". Use this private language as you think about the project.

5. Split off a well-defined sub-problem. The tape suggests writing the procedure

TIC.

Wholes and Parts in Solving Problems:

There is a general heuristic principle that we can call "divide and conquer" -- split a project into parts and do each one separately.

But parts can fit together into wholes in more than one way. In many graphics Logo procedures, the parts can each be made separately as sub-procedures and then put together into one whole in a superprocedure. (For example, the RIVER procedure in New Mindstorms Tapes 2 and 3.) In the meters project, one could think of drawing the numbers as a separate part in this sense. But we have decided to focus on making the dials work. Is each dial a separate part? You could think of it like that and you could make a separate sub-procedure for each dial. In fact, it may be a good idea to do this. Certainly if you are having serious trouble with the whole project you should do so. But the way the project is developed on the tape does not split the project into parts each represented by a subprocedure.

The tape illustrates a different kind of "breaking the problem into parts." Here the parts are different functions within a procedure rather than separate procedures.

Thus TIC is first written as a procedure that does just one simple thing and TOC causes this simple thing to be repeated. This gives a simple procedure for just one dial:

```
TO TOC
TIC
TOC
END
```

But there is another function -- making the next DIAL move. This is achieved by adding a new action:

```
IF HEADING = 0 [TELLNEXT [TOC]]
```

The division of the problem into parts was not like the familiar elementary Logo situation where you make a village by dividing it into houses and make a house by dividing it into its geometrically separate parts. The village program allows you to see the separate parts in the final drawing. Here the TIC is not present in the drawing as a thing. But if you think of the action of the meters as running in time, you could say to yourself "now TIC is happening."

Turtles and Actors as Agents for Recursion:

We do not have a subprocedure for each of the dials. The same procedure is used to activate all of them. The important step in making a mental model is to give individual existence to different "runnings" of the same procedure.

Think back on the SPIRAL procedure acted out by students in Hurdles Tape 3. The same procedure is run several times. When the children act it out, there is a separate child -- a separate actor -- for each running. In the Meters project, we use a slightly different representation for some of the different runnings: we use different turtles.

Think of the way TOC runs from the beginning. We could represent this by having a new actor each time we run TIC, but this seems unnecessary. For the first nine TICs, there is no conceptual difficulty and thus no need to bring in actors to help us think about the way the procedure is running. When the first dial has completed its first cycle of ten TIC movements, then the second dial has to move. This situation would be tricky if we didn't have the second turtle.[2] But the second turtle makes it simple to think about what is happening. We assume that the second turtle has already been placed in the right position and that the procedure TELLNEXT will make sure that it is given the instruction to move the hand on the second dial.

```
TO TELLNEXT :INSTRUCTION.LIST
ASK WHO + 1 :INSTRUCTION.LIST
END
```

[2] We'll talk about that in a while for those who want to try the project without using multiple turtles.

The interesting point to think about here is that the extra turtles provide a simple representation of the process -- just as the actors did in the representation of recursion we used before.

Doing It With One Turtle:

The issue is made clearer by struggling with the project in a single turtle context. The conceptually simplest way to do this with one turtle is probably to start by implementing multiple turtles. A multiple turtle microworld is available on the diskette, and you can examine the procedures to see how we have done this. But suppose you choose not to do that. How can you write these procedures strictly in the spirit of single turtle Logo?

Even here, the best way to think about the project is to use multiple turtles as a metaphor. TOC always uses one turtle -- but you can think of TELLNEXT as giving it another turtle's identity -- one that is appropriate for the dial being moved at the moment.

How can the procedure (or the actor for that running of the procedure) "know" which dial to use -- which turtle identity to adopt? The natural way in recursive programming is to use inputs to carry this information. Let's look at a way to do this.

```
TO TOC :DIALNUMBER
SETPOS PLACE :DIALNUMBER
SETHEADING ANGLE :DIALNUMBER
TIC
IF HEADING = 0 [TOC :DIALNUMBER + 1] [STOP]
TOC :DIALNUMBER
END
```

And to provide a stop rule:

```
TO TOC
IF WHO = 4 [STOP]
TIC
TOC
END
```

3. SWITCH

The next project on the videotape is a typical example of a recursive reporter similar to ADDUP and BIGGEST in *Hurdles 3*. (Perhaps you should review them to be sure they are fresh in your mind.)

Our task here is to write a recursive reporter that will take a list of words (specifically an English sentence) and produce a different list. In particular, it should transform [THE PRINCIPAL IS MAD] into [THE KING IS HAPPY] -- but anything else, such as [THE STUDENTS ARE PLAYING], should stay the same. The first step is to look for something simpler. Very often in list processing, the "something simpler" is a procedure that acts upon individual elements of the list. In this case, those elements are words.

At first glance, SWITCH may seem more complicated than ADDUP or BIGGEST, but in some ways it is actually simpler. SWITCH acts on a list of words, and its effect is simply to collect into one list the results of actions taken on the individual words. The natural first step then is to write SWITCH.WORD -- so we will put aside the whole question of acting on lists and focus exclusively on SWITCH.WORD. When that is done, we will return to SWITCH.

```
TO SWITCH.WORD :WORD
IF :WORD = "PRINCIPAL [OP "KING]
IF :WORD = "MAD [OP "HAPPY]
OP :WORD
END
```

One way of mastering a new procedure is to take it over and change it. We suggest that you do this now. For example, you could expand SWITCH.WORD's vocabulary by adding more words for it to respond to. Or you could turn it into a censor, making it eliminate words it didn't like, perhaps reporting something like

!DELETED!. Or it might change the background color as well as reporting a different word.

The screen in the videotape shows a visualization procedure. Just as some procedures were played out by actors in earlier tapes (HOUSE, HEADING, COMPASS and SPIRAL), here the sequence of instructions are played out on the screen. Our advice is to imagine actors as you watch SWITCH on the tape. Later, you can recruit other people to act it out with you. This will help you form a strong link between actual actors and thinking about these screen representations. Your goal is to internalize the actor model to the point where thinking in these terms becomes second nature.

SWITCH.WORD is a straightforward reporter, so you probably didn't need the analogy with actors to see how it works. When we come to SWITCH, however, the actor model is essential for understanding what is happening. So this "talking through a procedure" is a good habit to get into, even in cases where you feel sure you understand how it works. By going through this process in the cases you *do* understand, you gain mastery of the process itself. You can then use this understanding for cases that you find puzzling. We don't mean for you to be pendantic or to do this all the time. The important thing is to get into the habit of visualizing the actors in your head so that it becomes second nature.

The actor analogy becomes central when we run SWITCH while it is already being run. This is where the actor analogy is most essential: to give a separate identity to each new running of SWITCH. On the screen, we represent this by showing a new copy of the SWITCH procedure. In acting out the same scenario, a new actor would be called.

Points to Emphasize in the Actor Model:

1. Each time that SWITCH is run, it calls out a new actor.

2. Only one SWITCH actor is active at any one time. The others are asleep, waiting for input to some instruction.

3. Remember: the sleeping actor is not done. It is waiting for a subprocecdure to be completed; just by coincidence, this subprocedure is also called SWITCH.

4. All recursion has a two-way movement. We can watch this two-way movement on the screen. The flow of control moves downward as each new instruction "goes green" and seeks its input. In the third running of SWITCH, the return motion begins when EMPTY? reports TRUE to IF. This motion is dramatized by the turtle bringing reports back up the green trail of waiting procedures.

5. Passing the buck is a good model for understanding SWITCH -- as well as other recursive procedures that handle lists. This strategy divides the task into different stages: one that is accomplished "on the way down" and another that gets done "coming back".

Passing the Buck:

We went from a procedure that acts upon individual words (SWITCH.WORD) to a procedure that transforms one list into another (SWITCH). In writing a procedure that produces an output, think of the process that makes it. "Passing the buck" helps us visualize the way in which SWITCH creates this new list. On the way down, each new running of SWITCH runs a SWITCH.WORD on the FIRST of its input. Then it passes the BUTFIRST of its input to a new SWITCH.

Passing the buck means that at any point, you think that somebody else will do the job for you if only it were a little simpler. In other words, if someone gives me a list of four words, I feel confident that I can handle one of them if somebody else would do the other three. When those three are done and the result comes "back up the line", I know how to put my own work into this list and give this new list back to whomever called me. And that's the key step in SWITCH:

```
OUTPUT SENTENCE (SWITCH.WORD FIRST :INPUT) (SWITCH BF :INPUT)
```

SWITCH BF :INPUT is saying to somebody else, "You do the job, but I'll be nice to you and give you one less word. SWITCH.WORD FIRST :INPUT is the part that I'm going to do. SENTENCE takes the work I just did and the report you will soon give me, and turns them into one list. OUTPUT gives that list "back up the line" to whomever called me.