Accompanying Text

for

***On Logo* Videotape**

Hurdles 3

Recursion

by Wyn Snow

Table of Contents

1. Introduction

```
TO WOW
HAND.UP
HAND.DOWN
WOW
END
```

Recursion is a simple idea -- which makes it deceptive, too. Like an iceberg, it has depths and subtleties which a beginner cannot see. Sometimes people who are sure they understand WOW become confused when they see a procedure with changing inputs, or one that juggles several different actions.

The problem is that their mental model of what happens in WOW doesn't explain these other procedures, so they throw up their hands in frustration. Since Logo often sounds like English, people expect it to behave like English. But English is a language for people -- who are smart -- and Logo is a language for computers -- who are, as children put it, "completely dumb."

To understand what's happening in recursion, we turn to our old friend -- playing turtle -- in slightly different form. We use it to help you build a model for thinking about what happens inside the computer when a program uses recursion.

Since this mental model is so important for understanding recursion, we present it first in this text. Descriptions of recursive procedures -- and suggestions for writing your own -- will then be easier to follow.

2. The Actor Model

In the videotape, a group of children play turtle and draw a spiral. Since it has 90 degree "corners" like a square, let's call it a SQUIRAL:

```
TO SQUIRAL :SIDE
FD :SIDE
RT 90
SQUIRAL :SIDE + 10
END
```

As they explain to the teacher, Laura draws the first line, turns and calls Alex -- who draws the second line, turns and calls Matt -- and so on. We could write each child's actions as a separate procedure:

```
TO LAURA :SIDE          TO ALEX :SIDE          TO MATT :SIDE
FD :SIDE                FD :SIDE               FD :SIDE
RT 90                   RT 90                  RT 90
ALEX :SIDE + 10         MATT :SIDE + 10        EMILY :SIDE + 10
END                     END                    END
```

Here it is quite clear that ALEX is a sub-procedure of LAURA, and that MATT is a sub-procedure of ALEX. This is the *entire secret of recursion.* When Laura performs SQUIRAL 10, she calls a new actor to perform the instruction SQUIRAL :SIDE + 10. Laura then "goes to sleep" until Alex completes his job and nudges her back awake again.

Remember how Mark's HOUSE procedure called FRAME and then ROOF in the first videotape? Mark didn't know what FRAME and ROOF actually do. Here, too, Laura doesn't know what Alex is going to do when she gives him SQUIRAL :SIDE + 10. She's just following a set of instructions, line by line. The fact that both actors perform a procedure named SQUIRAL is only "coincidence" -- even though we can use that coincidence deliberately for our own purposes.

We encourage everyone to "play turtle" with recursion -- to recruit several friends and act out each instruction. It's an important experience for anyone who is learning recursion, as well as a powerful debugging tool for seeing how a particular program works.

The point to emphasize is that each new actor is performing a subprocedure. And just like any other subprocedure, that new actor has its own set of inputs.

3. Forever Programs

Recursion isn't an isolated mathematics daydream -- it's all around us. Seashells are an especially beautiful example: each tiny chamber is a slightly bigger version of the one before it. Another example is children on a see-saw going up and down and up and down and up and down and up .... They probably *would* go on forever

if they didn't get hungry first!

Many people enjoy turtle geometry, so drawing spirals has already become Logo's "traditional" route into recursion. It's not the only one, however, and we will show you some different ideas in section 5.

Like WOW, these procedures can be called "forever programs" because they never stop. They are like Jack taking his picture of Jill taking her picture of Jack taking his picture of Jill taking her picture ... and so on. To stop them, use the STOPIT! keystroke for your machine. (IBM PC: CTRL-BREAK. IBM PCjr: FCTN-BREAK. Apple: CTRL-G. MacIntosh: "curly-CTRL"-S. Etc.) We will soon show you how to put stop rules into your procedures so they will stop themselves.

4. Spirals

"Forever" programs are easy to write. Just put a recursion line at the end of the procedure.

```
TO SILLY :SIDE
FD :SIDE
RT 90
SILLY :SIDE                    recursion line
END
```

SILLY takes an input and draws a square: one side and another side and another side and another ... and so on. To make a spiral, change SILLY so that each new actor gets a bigger input for :SIDE.

```
TO SPIRAL :SIDE
FD :SIDE
RT 90
SPIRAL :SIDE + 10
END
```

This is what Laura and Alex and Matt and the other children drew on the videotape. SPIRAL is a good starting point for experimentation. Change the 10 to other numbers and see what happens. What do you think will happen if you change SPIRAL :SIDE + 10 to SPIRAL :SIDE - 10?

> For now, we are ignoring the PRINT "WOW! part of their SPIRAL procedure. We will come back to it in section 8.

You can change the shape of the spiral by changing the angle the turtle turns. Instead of RT 90, two familiar possibilities are RT 120 and LT 144. Naturally, if you don't want to write a zillion procedures for these different spirals, or spend half your time changing the procedure, you can give SPIRAL a second input for the angle.

```
TO SPIRAL :SIDE :ANGLE
FD :SIDE
RT :ANGLE
SPIRAL :SIDE + 10 :ANGLE
END
```

This makes it easy to see what happens with many different angles.

You can start with some "obvious" :ANGLE inputs -- numbers like 90 and 120 and 60. One that may surprise you is 180. Several other interesting angles are:

```
SPIRAL 10 72
SPIRAL 10 360 / 7
SPIRAL 10 144
SPIRAL 10 200
```

A completely different strategy is to try "silly" numbers, like 66 or 137 or 189. Another is to give inputs that are close to familiar numbers.

```
SPIRAL 10 93
SPIRAL 10 122
SPIRAL 10 177
```

There are many more ways of "messing around" with spirals, and we have put several of these into appendices:

- A: Smooth Spirals -- making curves.
- B: Beyond Addition -- multiplying instead of adding.
- C: Seashells -- spirals with shapes.
- D: Another Angle on Spirals -- changing the angle.

Some of their results can be quite spectacular, so we encourage you to spend as much time as you like with them.

The next section explores a completely different avenue for learning recursion: using PRINT with words and lists.

5. A Different Route: Recursion with Print

Once upon a time, Logo didn't have turtle graphics. (Yes, Logo is older than the turtle.) And even though many people -- both adults and children -- enjoy making designs with the turtle, some do not. Some prefer using PRINT to "mess around" with Logo words and lists.

The simplest "forever program" with PRINT must surely be BABBLE:

```
TO BABBLE
PR [I LOVE TO TALK]
BABBLE                          recursion line
END
```

How can we write a procedure that will endlessly print things like CATS BARK and DOGS LAUGH and CHILDREN MEOW? We can use RANDOM to select one of the items in a list.[1]

---

[1]For LogoWriter, use INSERT instead of TYPE; for Terrapin, use PRINT1.

```
TO CHAT
TYPE ITEM (1 + RANDOM 3) [CATS DOGS CHILDREN]
TYPE "\
PRINT ITEM (1 + RANDOM 3) [MEOW BARK LAUGH]
CHAT
END
```

Notice that we must add 1 to RANDOM's output. There's no such thing as the zero-th ITEM of a list -- and we would never get the last item since RANDOM always reports a number *smaller* than its input.

Writing a few reporters will help us clear up the clutter. NOUN and VERB will be subprocedures of CHATTER. PICK.ONE will randomly choose one item from a list.

```
TO CHATTER
TYPE NOUN
TYPE CHAR 32
PRINT VERB
CHATTER
END

TO NOUN
OP PICK.ONE [CATS DOGS CHILDREN]
END

TO VERB
OP PICK.ONE [MEOW BARK LAUGH]
END
```

NOUN and VERB make it easy to simplify CHATTER even further.

```
TO TALK
PR SE NOUN VERB
TALK
END
```

> You may want to try writing your own PICK.ONE before seeing how we did it. (Hint: we used RANDOM, ITEM and COUNT. A different strategy would use only RANDOM and ITEM. You could also use RANDOM and COUNT to give an input to a recursive subprocedure.)

ITEM needs two inputs. The first must be a number; the second can be either a word or a list. ITEM reports the *number*-th element of that word or list. So to pick randomly from a list, we need to give ITEM a random number and a list.

The COUNT reporter says how many elements are in a list[2] So COUNT can do the work of giving an input to RANDOM -- and makes it easy to add more things to the list.

```
TO PICK.ONE :LIST
OP ITEM (1 + RANDOM COUNT :LIST) :LIST
END
```

Naturally, you can expand TALK to create all sorts of patterns. For example, you can include an ADVERB or ADJECTIVE or whatever else you like -- even a question to make it interactive!

> Appendices F and G show two different ways of expanding on TALK. Since they don't use inputs, you can explore them now if you want and come back to this next section later.

---

[2]In most versions of Logo, it will also report the number of characters in a word.

6. Printing Inputs

The next step is to look at recursion with inputs. A typical starting place with PRINT is a procedure like SHOWDOWN:

```
TO SHOWDOWN :WORD
PRINT :WORD
SHOWDOWN BF :WORD
END
```

You may already have a vivid idea of what SHOWDOWN "RUMPLESTILTSKIN will do. Try it anyway -- and see if you can predict what sort of error message you will get.

Remember to keep the actor model clearly in mind:

> Laura runs SHOWDOWN "RUMPLESTILTSKIN. Her first instruction is PRINT :WORD, which is simple and straightforward. Her next instruction is SHOWDOWN BF :WORD, so she calls Alex.
>
> > Alex gets his input from BF and runs SHOWDOWN "UMPLESTILTSKIN. His first instruction is to PRINT his input. Then he reaches SHOWDOWN BF :WORD and calls Matt.
> >
> > > Matt runs SHOWDOWN "MPLESTILTSKIN. First, he PRINTs MPLESTILTSKIN. Then he reaches SHOWDOWN BF :WORD and calls a new actor ...
> > >
> > > > and so on, until ...
> > > >
> > > > > Yvette is called to run SHOWDOWN BF "N. BF reports the empty word, so Yvette has her input and PRINTs " . (PRINT will accept both empty words and empty lists). Now she calls Zach.
> > > > >
> > > > > > Zach tries to run SHOWDOWN BF " . He needs an input -- but BF complains: it doesn't like the empty word as input!

So SHOWDOWN isn't really a "forever" program. It's more of an "until you run out of input" program. COUNTDOWN doesn't have this problem. It won't run out of numbers for a very, very, very long time.

```
TO COUNTDOWN :NUM
PRINT :NUM
COUNTDOWN :NUM - 1
END
```

SECRET.CODE is a slightly different example of this same idea:

```
TO SECRET.CODE :WORD
TYPE CHAR (3 + ASCII FIRST :WORD)
SECRET.CODE BF :WORD
END
```

Two of these primitives may be new to you. ASCII and CHAR are reporters. Just as Logo uses a number for each color, each character also has its own number.

> These numbers are called ASCII -- the acronym for American Standard Code for Information Interchange, a system developed many years ago when computers first needed a way of translating back and forth between numbers and the alphabet.

ASCII reports the number of a character, and CHAR reports the character of a number. ASCII "A reports 65, and CHAR 65 reports "A. SECRET.CODE uses these ASCII numbers to type a word that is "three letters up" from its input.

A good project for seeing if you understand how these procedures work is to write a REVERSE procedure that will TYPE a backwards version of its input. (Look in Appendix K if you get stuck.)

There are many ways of expanding upon these ideas. Several of them have been put into appendices:

- E: Graphic Words and Lists -- using lists to flash background colors. This is an especially good project for "dissolving the graphics/word-list barrier."
- F: More Words and Lists -- shows how to make TALK interactive and create its own replies.
- G: Word Games -- builds upon NOUN to make noun predicates and a recursive SUBJECT.

The procedures in these appendices do not "run out of input" -- so they do not need the STOP rules that we will be looking at next.

The error messages we've seen are annoying, but they haven't yet interrupted a procedure before it finished the work we wanted it to do. In order to explore recursion further, however, we need the control that STOP rules provide.

7. STOP Rules

Letting a procedure go on forever can get kind of messy, or boring, or both. Also, it's hard to hit that STOPIT! key at exactly the right time for the effect you want. If you have already tried the SHOWDOWN and SECRET.CODE procedures, you probably wished they didn't complain when they got to the end of their input. Using a stop rule will solve both these kinds of problems.

What's needed in this situation is something that can change the flow of control. ("Flow of control" means the way that Logo normally carries out a procedure: line by line, procedure by procedure.) And that's exactly what a *controller* is designed to do. Perhaps you remember the way we used STOP in Hurdles Tape 1:

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

Naturally, different procedures need different criteria. You might want to stop SPIRAL before it gets too big for the screen.

```
TO SPIRAL :SIDE :ANGLE
IF :SIDE > 100 [STOP]
FD :SIDE
RT :ANGLE
SPIRAL :SIDE + 5 :ANGLE
END
```

You may be wondering why the stop rule is at the beginning. You could put it in other places -- which would give you slightly different effects. Try moving the stop rule and see what happens.

In some cases, however, the change is more drastic. The complaint in SECRET.CODE was from FIRST, so we must STOP the flow of control before it reaches FIRST " . Some stop rules *must* be placed at the beginning.

```
TO BETTER.CODE :WORD
IF EMPTY? :WORD [STOP]
TYPE CHAR (3 + ASCII FIRST :WORD)
BETTER.CODE BF :WORD
END
```

EMPTY? is often a good tester to use in stop rules with lists or words, but it's not the only one. Notice that which tester you use determines where the stop rule goes.

```
TO DIFFERENT.CODE :WORD
TYPE CHAR (3 + ASCII FIRST :WORD)
IF 1 = COUNT :WORD [STOP]
DIFFERENT.CODE BF :WORD
END
```

Which you prefer is really a matter of taste. Many Logo programs do have the STOP rule at the beginning -- a style that develops when EMPTY? is used often. This placement also makes it easy to find the STOP rule. But the essential part is to put STOP where it is *needed.* Look at WANDER again. What would happen if the STOP rule were moved to the top?

> Appendix H shows you how recursion can make "goof-proof" procedures. This is an important strategy for interactive programming.

8. Two-way Movement

Perhaps you've been wondering what happened to PRINT "WOW! in the SPIRAL procedure on the videotape? This instruction shows how the flow of control returns back through the line of waiting actors.

```
TO SPIRAL :SIDE
IF :SIDE > 50 [STOP]
FD :SIDE
RT 90
SPIRAL :SIDE + 10
PRINT "WOW
END
```

Let's see how this works. From Laura through Zach, the process is already pretty familiar. Notice, however, those easily glossed-over words: "and goes to sleep." This is the crucial part of understanding the two-way movement of recursion.

> Laura is given the instruction SPIRAL 10. She runs IF; > reports FALSE so [STOP] is ignored. Then she moves the turtle FORWARD 10 and RIGHT 90. Her next instruction is SPIRAL :SIDE + 10, so she calls Alex and goes to sleep. (The + reporter takes Laura's :SIDE of 10, adds 10, and reports 20.)
>
> > Alex runs SPIRAL 20. IF's first input is again FALSE. Alex moves the turtle FD 20 and RT 90. Then he calls Matt to do SPIRAL :SIDE

+ 10 and goes to sleep. (The + reporter now takes Alex's :SIDE of 20, adds 10, and reports 30.)

Matt runs SPIRAL 30. 30 > 50 reports FALSE to IF, he moves the turtle FD 30 and RT 90, calls Emily for SPIRAL 30 + 10 and goes to sleep.

Emily runs SPIRAL 40. She does IF, FD 40 and RT 90, then calls Yvette for SPIRAL 40 + 10 and goes to sleep.

Yvette runs SPIRAL 50. After doing IF, FD 50 and RT 90, she calls Zach to do SPIRAL 50 + 10 and goes to sleep.

Zach runs SPIRAL 60. Since his :SIDE is 60, > reports TRUE and IF runs the instruction list [STOP]. So Zach nudges Yvette awake and says "I'm done."

Yvette goes to her next instruction and PRINTs "WOW! Her next line is END, so she nudges Emily awake and says "I'm done."

Emily PRINTs "WOW!, then reaches END and nudges Matt awake.

Matt PRINTs "WOW!, reaches END and nudges Alex awake.

Alex PRINTs "WOW!, reaches END and nudges Laura awake.

Laura also PRINTs "WOW!, reaches END and tells Logo that she's done.

All recursion has this two-way movement. Someone appoints you (your boss), and you appoint a subordinate. When the subordinate is done, you are woken up. You proceed from the same point in your script and continue with your instructions. When you are done, you wake up your boss.

Procedures are literal-minded. They don't end until they reach END -- unless a

controller stops them. All the procedures you've seen up to now have been "tail recursive" (the recursion line is just before END) so the return motion was invisible. PRINT "WOW! lets you see it.

Another procedure that shows this two-way motion is DOWNUP.

```
TO DOWNUP :WORD
IF EMPTY? :WORD [STOP]
PRINT :WORD
DOWNUP BF :WORD
PRINT :WORD
END
```

> The turtle can use this two-way flow to create unusual branching designs. See the TREE procedure in Appendix I.

9. Passing the Buck

When you understand this return movement -- that each "not finished" actor is only snoozing and waiting until the subordinate's job is done -- then you understand the way recursion works. We can use this return motion deliberately to solve a problem. Since this strategy gets somebody else to do most of the work, we can call it "passing the buck."

Suppose we want to add up a series of numbers. The easy way is to keep the first number, then get someone else to add up the rest. Adding that first number to this result is a simple task. Here is the ADDUP procedure:

```
TO ADDUP :NUMBERS
IF 1 = COUNT :NUMBERS
   [OP FIRST :NUMBERS]
OP SUM
     FIRST :NUMBERS
     ADDUP BF :NUMBERS
END
```

The return movement is the heart of the stategy for solving the problem. Use the actors to see what happens.

> Laura is given ADDUP [1 2 3]. COUNT [1 2 3] is not 1, so she goes to the next instruction. OP calls SUM which needs two inputs. FIRST reports 1. SUM now calls Alex to do ADDUP BF [1 2 3].
>
> > Alex is given ADDUP [2 3]. = again reports FALSE, so he goes to his next instruction. OP calls SUM which gets a 2 from FIRST [2 3] and calls Zach to do ADDUP BF [2 3].
> >
> > > Zach runs ADDUP [3]. COUNT [3] is 1, so = reports TRUE and Zach runs the instruction list [OP FIRST :NUMBERS]. OP gives FIRST [3] -- the number 3 -- back to Alex's SUM.
> >
> > Alex's SUM now has both inputs, 2 and 3, and reports 5 to OP. OP tells Alex to give the number 5 back to Laura.
>
> Laura's SUM now has both inputs, 1 and 5, and reports 6 to OP -- which tells Laura to give the number 6 to whomever called her.

Notice that each actor needed an OP in order to give a report to its boss. Providing this OP is the most common "hurdle" that people experience when they start writing recursive reporters. This next section shows you a typical bug in using OP with recursion.

10. Using OP in Recursive Reporters

All reporters need OP in order to give a report. And all recursive procedures need a stop rule. "Aha!" many people say at this point, "What I need to do is substitute OP for STOP in the stop rule."

To illustrate what happens next, let's pretend that you don't have a COUNT primitive -- and we'll write one "from scratch." This is a very different kind of exercise from the remodelling we did in Hurdles 1, and is equally valuable for understanding how a Logo primitive works.

We will use the pattern for stop rules from section 7 and substitute OP for STOP. COUNT.EM will use a subprocedure to do the recursive counting. The idea is that with each new recursion call, it will BF :THESE and :NUM will become :NUM + 1. When it reaches the end of :THESE, the subprocedure will report the :NUM it has reached.

```
TO COUNT.EM :THESE
OP BUGGY :THESE 0
END

TO BUGGY :THESE :NUM
IF EMPTY? :THESE [OP :NUM]          stop rule with OP
BUGGY BF :THESE :NUM + 1
END
```

The error message will tell you that BUGGY didn't report to BUGGY. We'll use the actors to help us find the bug, and we'll start with the instruction PR COUNT.EM "OOPS. PR calls COUNT.EM which calls OP which calls Laura to do BUGGY "OOPS 0.

> Laura runs BUGGY "OOPS 0. Since OOPS is not empty, she goes to her next instruction and calls Alex to do BUGGY BF "OOPS 0 + 1.
>
> > Alex runs BUGGY "OPS 1. EMPTY? reports FALSE, so he goes to his next line and calls Matt to do BUGGY BF "OPS 1 + 1.
> >
> > > Matt runs BUGGY "PS 2. :THESE is not empty, so he goes to his next line and calls Yvette to do BUGGY BF "PS 2 + 1.
> > >
> > > > Yvette runs BUGGY "S 3. Since S is not empty, she goes to her next line and calls Zach to do BUGGY BF "S 3 + 1.
> > > >
> > > > > Zach runs BUGGY " 4. EMPTY? now reports TRUE, so IF runs the instruction list [OP :NUM]. OP tells Zach to give 4 to Yvette.

And there's the bug. Yvette has no idea what to do with Zach's report. It has become a "hot potato" -- just like the one we saw with HDG in the Hurdles 1 tape.

Notice that in ADDUP, there were *two* OPs: in the stop rule *and* in the recursion line. The stop rule OP exerts control when the recursive process has "reached the end of the line" -- and starts the flow-back. The recursion line OP allows the "middle actors" to hand their reports "back up the line." Both OPs are needed whenever you write a recursive reporter.

```
TO COUNT.UP :THESE
IF EMPTY? :THESE [OP 0]
OP 1 + COUNT.UP BF :THESE
END
```

The next section uses the BIGGEST procedure shown on the videotape to describe the process of writing a recursive procedure. Since the process is a flexible one, the steps don't need to be followed in an exact, one-two-three sequence.

11. Strategies for Recursion

The "buck-passing" technique was used in the videotape to find the largest of a set of blocks. Our project here is to write a BIGGEST procedure that will find the longest word in a list.

STRATEGY 1: Simplify the task. What's the simplest case -- when does "the buck stop here"? And what action needs to be done recursively by a series of new actors "passing the buck"?

In BIGGEST, the simplest case for people is when there are two. Then it is obvious which is the larger. But for computers, the simplest case is when there is only one (or in some cases, none). So that will become our stop rule, and since BIGGEST will be a reporter, we need to use OUTPUT:

```
TO BIGGEST :LIST
IF 1 = COUNT :LIST [OP FIRST :LIST]
OP recursive-BIGGEST-action-of-some-sort
END
```

The recursive action is a comparison -- which of two inputs is BIGGER. Each actor will output the result of BIGGER, so BIGGER needs to compare the

COUNT of its two inputs, and that's pretty straightforward.

```
TO BIGGER :THIS :THAT
IF (COUNT :THIS) > (COUNT :THAT)
   [OP :THIS]
OP :THAT
END

TO BIGGEST :LIST
IF 1 = COUNT :LIST [OP FIRST :LIST]
OP BIGGEST first-word second-word
END
```

> This time, the parentheses *are* needed around COUNT :THIS. What would happen without them?

Now we use the buck-passing and return-motion to finish the recursion line. What we need to do is create the two inputs for BIGGER.

In the blocks version of BIGGEST, each person took one stick and passed the rest. The last person had only one stick, so that was the biggest and was returned to the previous person -- who now had two sticks, compared them, and returned the BIGGER one ... and so on, back up the line.

So we have the first input for BIGGER -- FIRST :LIST. The second will be whatever comes back. This "whatever comes back" will be the BIGGEST of the BF :LIST. So our recursion line is now complete:

```
TO BIGGEST :LIST
IF 1 = COUNT :LIST [OP FIRST :LIST]
OP BIGGER
     FIRST :LIST
     BIGGEST BF :LIST
END
```

STRATEGY 2: Anthropomorphize! Play turtle. Recruit friends to act out what

you want the procedure to do. "Translate" these actions into a first draft procedure. See what sort of error message you get, and get your actors to be "completely dumb" and literal-minded until you find the bug(s).

STRATEGY 3: Find something similar. Use the procedures here as starting points and blueprints for other procedures. Unless you're writing a "forever" program, you will need a stop rule -- with either STOP or OP. If the procedure is a reporter, it must have OP in both the stop rule(s) and the recursion line(s).

> A nice project for applying these principles is making a PIGLATIN reporter. We show you the result in Appendix L.

Are you still wondering how children on a see-saw is an example of recursion?

```
TO SEE
IF FEET.ON.GROUND? [PUSH.UP]
SAW
END

TO SAW
IF FEET.ON.GROUND? [PUSH.UP]
SEE
END
```

One child does SEE, the other does SAW, then SEE, then SAW, then SEE ... and so on. In this variety of recursion, one event triggers another. Juggling also has this kind of recursive action. Dr. Papert describes this on pages 105-113 of *Mindstorms*, and a TIC TOC project of this type is presented in Hurdles 4.

So, the next time you see a spider web, "think recursion." The next time you get hungry, "think recursion." The next time you watch waves at the beach, "think recursion." It's not just seashells. It's any process of growth or motion that has this "calls another actor to perform the same procedure" quality.

12. Appendix A: Smooth Spirals

Making spirals with smooth curves is an interesting project.

When we wanted the turtle to draw a circle, playing turtle showed that it needed to go forward a little, turn a little, go forward a little, turn a little, and so on. A spiral is sort of like a circle, except the turtle goes forward a little more each time. And in your experiments with SPIRAL, you saw that the smaller turns were closer to a smooth curve. So a first attempt at a smooth spiral might be:

```
TO CURVE :STEP
FD :STEP
RT 1
CURVE :STEP + 1
END
```

But the results of CURVE 1 are a little disappointing. What kinds of changes are needed here?

Two strategies work well for making a more satisfying CURVE. The turtle needs to add a very tiny number to :STEP -- or to turn farther. And you don't have to limit yourself to one possibility or the other. You can try both.

If you want to do several experiments, it's a nuisance to spend half your time changing the procedure.

```
TO FANCY.SPIRAL :STEP :ANGLE :MORE
FD :STEP
RT :ANGLE
FANCY.SPIRAL :STEP + :MORE :ANGLE :MORE
END
```

> Naturally, you can simply change your SPIRAL or CURVE procedure; you don't have to call it FANCY.SPIRAL. We use different names to make it easier to distinguish among them.

"Help!" many people say when they see this recursion line. And at first glance, it does look rather formidable.

FANCY.SPIRAL has three inputs. :STEP is the input for FORWARD, and :ANGLE is the input for RIGHT. These are familiar, even old friends from the work you've already done with POLYGONs. But :MORE is a new idea. (If you haven't worked with POLYGON, you can find four varieties in Appendix J.)

Just as :ANGLE replaced the 90 degrees we used in SQUIRAL, :MORE is replacing the 10 that is added to :STEP for each new actor. Thus, :MORE is an input for the + reporter in the recursion line.

Many people try a recursion line of FANCY.SPIRAL :STEP + :MORE :ANGLE. Since all three names -- :STEP, :MORE and :ANGLE -- are there, they feel convinced that FANCY.SPIRAL has gotten the three inputs it needs. When they get an error message that FANCY.SPIRAL needs more inputs, they become frustrated.

Playing turtle -- or "playing computer" if you prefer -- shows the bug clearly:

> Laura is called and given the instruction FANCY.SPIRAL 1 15 0.1. She needs an input for :STEP and finds the 1. Now she needs an input for :ANGLE and uses the 15. Finally she needs an input for :MORE and takes 0.1. Since she has all her inputs, she can do her work. She goes FORWARD 1, turns RIGHT 15, and calls Alex to do FANCY.SPIRAL 1 + 0.1 15.
>
> > Alex runs FANCY.SPIRAL and starts looking for his inputs. The + reporter gives him 1.1 for :STEP. Then he needs an input for :ANGLE, and there it is: 15. Now he needs an input for :MORE, but nothing is there. So he complains back to Logo: "Hey! I didn't get all my inputs."

People are smart: they can *see* that the third input is :MORE. It doesn't matter to them that it came earlier in the instruction. But computers can only see one thing at a time. They really are "completely dumb." That's why the actor model is so important. It's a tool that helps us be just as literal-minded as a computer so we

can understand what it's doing.

Parentheses make the recursion line easier to read -- FANCY.SPIRAL (:STEP + :MORE) :ANGLE :MORE. Now the three inputs stand out clearly. Even though Logo doesn't need these parentheses, you can go ahead and put them in. It's important to use any tool that helps you read and understand a program.

Which do you think will give you the kind of spiral you want?

```
FANCY.SPIRAL 1 5 0.1
FANCY.SPIRAL 1 5 0.01
FANCY.SPIRAL 1 5 0.001
```

13. Appendix B: Beyond Addition

Addition is only one of the tools available for changing :SIDE. Subtraction is another, of course, but it's not really very different from addition. What about multiplication?

```
TO MULTI.SPI :SIDE :ANGLE :BIGGER
FD :SIDE
RT :ANGLE
MULTI.SPI (:SIDE * :BIGGER) :ANGLE :BIGGER
END
```

MULTI.SPI also takes three inputs. Each new actor's :SIDE input is now created by * instead of by +.

What kinds of numbers do you want to try for :BIGGER? What do you think will happen with these:

```
MULTI.SPI 10 90 5
MULTI.SPI 10 90 2
MULTI.SPI 10 90 1.1
MULTI.SPI 10 90 1.001
MULTI.SPI 10 90 0.5
```

Or you might want to keep :BIGGER the same, and experiment with the angle:

```
MULTI.SPI 5  90 1.1
MULTI.SPI 5 120 1.1
MULTI.SPI 5  72 1.1

MULTI.SPI 5  20 1.01
MULTI.SPI 5  10 1.01
```

14. Appendix C: Seashells

To draw the seashell that you saw on the videotape, you need a triangle with two equal "legs" (also called an isosceles triangle). This one is *state transparent* -- it starts and ends with the same position and heading -- which is always an advantage when you are putting subprocedures together.

```
TO ISOSCELES :SIDE :ANGLE
FD :SIDE NAME POS "POINT        draws one side, names point
BK :SIDE RT :ANGLE              goes back and turns
FD :SIDE                        draws second side
SETPOS :POINT                   draws third side
LT :ANGLE BK :SIDE              returns to original state
END
```

This triangle names a position and uses SETPOS to draw the third side. That makes it easy to draw ISOSCELES triangles with many different angles. You don't need to go through the trial and error of figuring out the size of the third side, or calculate its angle to get the correct turn.

```
TO SHELL :SIDE
ISOSCELES :SIDE 30
RT 30
SHELL :SIDE + 5
END
```

SHELL can also serve as a blueprint for exploring spirals. Again, you can use different angles in the procedure. Or you could use a SQUARE or other POLYGON instead of ISOSCELES, and let them overlap. You can also move the turtle to a different position:

```
TO SURREAL :SIZE
SQUARE :SIZE
RT 90
FD :SIZE / 2
SURREAL :SIZE + 10
END
```

15. Appendix D: Another Angle on Spirals

Changing :SIDE is only one strategy for "messing around" with SPIRALs. Have you considered changing :ANGLE instead?

```
TO ANGLE.SPI :SIDE :ANGLE
FD :SIDE
RT :ANGLE
ANGLE.SPI :SIDE :ANGLE + 10
END
```

Are you ready for a surprise? Try ANGLE.SPI 10 90. Don't even try to guess what it will do. Now CLEARSREEN and try ANGLE.SPI 10 45. Try different numbers for :ANGLE and see how many different shapes you can draw.

> ANGLE.SPI 10 90 gives the same result as ANGLE.SPI 10 10. The interesting numbers for :ANGLE are 0 through 9.

Soon you will want to try adding a different amount to the angle:

```
TO SWIRL :SIDE :ANGLE :MORE
FD :SIDE
RT :ANGLE
SWIRL :SIDE (:ANGLE + :MORE) :MORE
END
```

Some interesting :MORE inputs to try are numbers between 1 and 25. Budding mathematicians in your class will probably be intrigued by seeing what happens with the prime numbers 7, 11, 13, 17 and so on. When :MORE is larger than 13 or

so, you will need to give SWIRL some help, because the inputs to :ANGLE can get very big indeed.

```
TO PRIME.SPIRALS :SIDE :ANGLE :PRIME
IF :ANGLE > 5000
   [PRIME.SPIRALS :SIDE (REMAINDER :ANGLE 360) :PRIME STOP]
FD :SIDE
RT :ANGLE
PRIME.SPIRALS :SIDE (:ANGLE + :PRIME) :PRIME
END
```

16. Appendix E: Graphic Words and Lists

Many people see graphics and "that word and list stuff" as two separate and distinct parts of Logo. DRAW A RED SQUARE (in the Hurdles Tape 1) showed one way of dissolving this barrier. FLASH.BG shows another way.

```
TO FLASH.BG :COLORS
SETBG THING FIRST :COLORS
WAIT 5
FLASH.BG LPUT (FIRST :COLORS) (BF :COLORS)
END
```

At first glance, this procedure may look rather complex. It takes a list of color names, uses the FIRST of that list to create an input for SETBG, then moves that name to the end of the list for the next FLASH.BG actor.

If we FLASH.BG [RED WHITE BLUE] for example, FIRST reports RED to THING, and THING "RED is our old friend :RED.

> THING "THIS and :THIS both report the *thing* named THIS. The main difference is that THING is more versatile than : since it can get its input from reporters like WORD and FIRST.

So SETBG gets the input it needs. WAIT 5 is completely straightforward.

The recursion line is different from the others seen so far. FLASH.BG needs a list of names as input. We could have used BF, but then we would have run out of colors rather quickly. Saying FLASH.BG [RED WHITE BLUE RED WHITE BLUE] only postpones the problem. Instead, LPUT takes the FIRST of :COLORS and puts it into the end of the list of BF :COLORS.

If your version of Logo has sprites (multiple turtles, changable shapes, sometimes SETSPEED), this area is especially fruitful for dissolving the boundary between graphics and "that word/list stuff." For example, one can create lists of turtles and lists of shapes, both of which can be manipulated with FIRST and BF for all sorts of intriguing results.

17. Appendix F: More Words and Lists

The simplest way to expand on TALK is to put more "pieces" into the pattern. One could PRINT (SE "THE NOUN VERB), for example, or include other kinds of words. Another strategy is asking a question to make it interactive, then using READCHAR and RANDOM to generate a reply:

```
TO SILLY.TALK
(PRINT [DO YOU KNOW THE]
       NOUN
       "THAT
       VERB
       "?)
REPLY READCHAR 2
SILLY.TALK
END
```

Don't let yourself get overwhelmed by (PRINT [DO YOU KNOW THE] NOUN "THAT VERB "?). The parentheses here tell PRINT that it's getting more than one input.

If your version of Logo cannot use parentheses to "override" a primitive's usual number of inputs, use

```
PRINT SE [DO YOU KNOW THE]
         SE NOUN
            SE "THAT
               SE VERB
                  "?
```

Or if you prefer, you can put the four SE reporters together -- PRINT SE SE SE SE [DO YOU KNOW THE] NOUN "THAT VERB "?

> Eliminating the space in front of the question mark can be a nice challenge. Simply saying WORD VERB "? may not work if VERB reports a list like [HAVE EATEN].

REPLY uses READCHAR (which supplies the input for :ANSWER) together with RANDOM in order to choose among four possible replies. The parentheses around WORD and its inputs are not necessary here. However, they do make it easier to see the two inputs for NAME.

```
TO REPLY :ANSWER :NUM
NAME (WORD :ANSWER RANDOM :NUM) "KEY
IF :KEY = "N1 [PRINT [GEE, THAT'S A SHAME.]]
IF :KEY = "NO [PRINT [GOSH, I THOUGHT YOU DID.]]
IF :KEY = "Y1 [PRINT [WOW, SO DO I.]]
IF :KEY = "YO [PRINT [I WISH I DID.]]
END
```

If you want to create more replies, don't forget to give REPLY a larger input for :NUM in SILLY.TALK, perhaps REPLY READCHAR 12.

18. Appendix G: Word Games

We can use recursion in a slightly different way to make a rich variety of sentences. These examples build upon the NOUN and VERB procedures in section 5.

```
TO NP
OP SE NOUN [OF THE]
END
```

NP (for NounPhrase) is very straightforward.[3] It's used with PRINT SE NP NOUN to create lists like [TREE OF THE CAT].

It also gives us three ways of building a subject for a sentence. SUBJECT picks one of them for us. Since one possibility is recursive, it could indeed go on forever (although the odds are rather slim).

```
TO SUBJECT
OP RUN PICK.ONE [[NOUN] [SE NP NOUN] [SE NP SUBJECT]]
END
```

First let's see why we needed RUN, even though we didn't need it before. Suppose we had said something like OP PICK.ONE [NOUN [SE NP NOUN]] instead.

PICK.ONE would report either NOUN or [SE NP NOUN] to OP, which would tell SUBJECT to give NOUN or [SE NP NOUN] to PRINT. So PRINT would get its input directly: the word NOUN or the list [SE NP NOUN] -- and neither would be run as a procedure or as a list of instructions. To do that, we need RUN -- and RUN needs a list. So the list given to PICK.ONE in SUBJECT must be a list of instruction-lists.

19. Appendix H: Oops!

Recursion can be used to make "goof-proof" procedures. This is a good technique for interactive procedures.

```
TO PLAY.GAME
PR [DO YOU WANT TO PLAY A GAME?]
NAME READCHAR "ANSWER
IF :ANSWER = "Y [ASK.WHICH.GAME]
IF :ANSWER = "N [MORE.TALK]
END
```

---

[3] NPHR on the LogoWriter diskette.

If the user types an M by mistake, PLAY.GAME has no leeway. It ends without a "match" for :ANSWER. If we make it recursive, it will continue to call new actors until it gets an answer it recognizes.

```
TO KEEP.ASKING
PR [DO YOU WANT TO PLAY A GAME?]
NAME READCHAR "ANSWER
IF :ANSWER = "Y [ASK.WHICH.GAME STOP]
IF :ANSWER = "N [MORE.TALK STOP]
PR SE [I DUNNO WHAT YOU MEAN BY] :ANSWER
PR [PLEASE TYPE N OR Y.]
KEEP.ASKING
END
```

A bug that often creeps into this kind of procedure is forgetting to put STOP into the instruction list for IF. Let's suppose the user types an N -- so IF runs MORE.TALK. Use the actor model to see what happens when MORE.TALK is done. (It makes a nice joke, but that's not what you meant.)

20. Appendix I: Making a Tree

The following rather tricky procedure uses recursion to draw a tree, complete with branches. TREE shows very clearly the differences between recursion and iteration.

```
TO TREE :SIZE :NUM
IF :NUM = 0 [STOP]
FD :SIZE
LT 30
TREE (:SIZE * 0.75) (:NUM - 1)
RT 60
TREE (:SIZE * 0.75) (:NUM - 1)
LT 30
BK :SIZE
END
```

Putting a little RANDOMness into this procedure is also an interesting challenge, and one that can generate very realistic-looking trees.

(Believe it or not, a day will come when you can write TREE from scratch!)

21. Appendix J: Polygons

There are different strategies for drawing polygons. For example, you can say the number of sides:

```
TO POLYGON.1 :NUM :SIDE
REPEAT :NUM [FD :SIDE RT 360 / :NUM]
END
```

or you can say the size of the angle:

```
TO POLYGON.2 :SIDE :ANGLE
REPEAT 360 / :ANGLE [FD :SIDE RT :ANGLE]
END
```

Both of these have recursive equivalents:

```
TO POLYGON.3 :NUM :SIDE
FD :SIDE
RT 360 / :NUM
POLYGON.3 :NUM :SIDE
END

TO POLYGON.4 :SIDE :ANGLE
FD :SIDE
RT :ANGLE
POLYGON.4 :SIDE :ANGLE
END
```

22. Appendix K: Reverse

To write a REVERSE command, you can use TYPE, LAST and BUTLAST:

```
TO REVERSE :SOMETHING
TYPE LAST :SOMETHING
REVERSE BL :SOMETHING
END
```

To write a REVERSE reporter, you will need OP and something to "glue" the pieces together: WORD for words, and LPUT for lists.

```
TO REVERSE.1 :WORD
IF EMPTY? :WORD [OP :WORD]
OP WORD LAST :WORD REVERSE.1 BL :WORD
END

TO REVERSE.2 :LIST
IF EMPTY? :LIST [OP :LIST]
OP LPUT LAST :LIST REVERSE.2 BL :LIST
END
```

Naturally, you may also want to write a REV.ANY procedure that would take both words or lists.

23. Appendix L: Piglatin

You will need four procedures:

- PIGLATIN will take different actions for words and lists.
- PL.1 will glue SAY to the end of words starting with a vowel.
- PL.2 will recursively strip consonants from the front of words and glue them to the back until it reaches a vowel.
- VOWEL? will detect vowels at the beginning of a word (or anywhere else, for that matter).

```
TO PIGLATIN :ENGLISH
IF EMPTY? :ENGLISH [OP :ENGLISH]
IF WORD? :ENGLISH [OP PL.1 :ENGLISH]
OP FPUT (PIGLATIN FIRST :ENGLISH) (PIGLATIN BF :ENGLISH)
END

TO PL.1 :ENGLISH
IF VOWEL? FIRST :ENGLISH [OP WORD :ENGLISH "SAY]
OP PL.2 WORD (BF :ENGLISH) (FIRST :ENGLISH)
END

TO PL.2 :ENGLISH
IF VOWEL? FIRST :ENGLISH [OP WORD :ENGLISH "AY]
OP PL.2 WORD (BF :ENGLISH) (FIRST :ENGLISH)
END

TO VOWEL? :LETTER
OP MEMBER? :LETTER [A E I O U Y]
END
```