Accompanying Text

for

***On Logo* Videotape**

Hurdles 2

Naming and Variables

by Wyn Snow

Table of Contents

The tape uses muffins as a metaphor for the ways Logo procedures can get their inputs. If you are hungry for some muffins, there are three Logo-like things I can do: I can give you a muffin, I can tell you the name MUFFINS so you can find the MUFFINS jar, or I can give you the recipe so you can make your own.

We can think of Logo procedures as being "hungry for" their inputs. (Some procedures, like FORWARD and REPEAT, are very finicky about what sorts of inputs they will accept, just as some people are choosy about the flavors of muffins they will eat. Other procedures, like PRINT and SE, will take anything you give them.) There are three different ways you can provide these inputs.

Directly: This is like taking a freshly-baked muffin and handing it to the procedure with your fingers. So we can say that the quote marks are like two fingers that take a word and give it directly to a procedure. PRINT "MUFFIN.

When there's more than one word, two fingers aren't enough. Then the words must be put into a list. That's what the brackets are for: they package a list for direct input. PRINT [MUFFINS BAGELS COOKIES]. If you like, you can call the [ ] brackets an *envelope* for the list.

There are two ways to give an input indirectly:

Using a name: This is like saying, "They're called MUFFINS." When you know the name, you can find the muffin jar by reading the labels. PRINT :MUFFINS. Even though computers and Logo don't have "real muffin jars" inside them, the process of finding a real input by using its name is very similar to using a pair of eyes to read the labels on the jars.

Using a procedure:

This is like giving a recipe for making muffins, something like

```
TO MUFFINS
COMBINE [FLOUR MILK EGGS BUTTER [BAKING
        SODA]]
BAKE 350 30
IF DONE? [OP "MUFFINS]
BAKE 350 5
OP "MUFFINS
END
```

Notice that this recipe is a reporter. PRINT MUFFINS. Any procedure that provides an input for another *must be a reporter.*

A procedure like MUFFINS that mimics a "real recipe" is also an excellent starting point for student projects and class discussions. We will come back to this idea in Appendix F, where we describe several kinds of next steps one can take.

1. Direct and Indirect Inputs

The underlying structure of inputs is quite simple. There are only 4 ways that a word can appear in Logo:

`PRINT "SQUARE`	`SQUARE` is quoted
`PRINT [A SQUARE]`	`A SQUARE` is packaged in a list
`PRINT :SQUARE`	`SQUARE` is dotted
`PRINT SQUARE`	`SQUARE` is unadorned

PRINT gets its input directly in the first two instructions. In PRINT "SQUARE, the quotes give the word SQUARE directly to PRINT. In PRINT [A SQUARE], the brackets give a list of the words A and SQUARE directly to PRINT. PRINT "SQUARE and PRINT [A SQUARE] are predictable. You can walk up to any version of Logo, type either instruction, and know exactly what will happen.

The other two are *indirect* inputs -- neither a specific word nor a specific list are given. Thus, we cannot predict what will happen. If you walk up to any Logo and say PRINT :SQUARE or PRINT SQUARE, the result depends on what the user has been doing.

An unadorned word is always treated as a procedure. If no SQUARE procedure has been defined, Logo will say something like I DON'T KNOW HOW TO

SQUARE. Or SQUARE itself might need some input: NOT ENOUGH INPUTS TO SQUARE. If SQUARE simply draws REPEAT 4 [FD 50 RT 90], it's not a reporter; the square will be drawn, but then: SQUARE DIDN'T REPORT TO PRINT. Or, as you saw in the DRAW A RED SQUARE sequence, SQUARE might be a reporter and the instruction would work.

The fourth possibility is PRINT :SQUARE. A dotted word is always treated as a name, but it might be the name of anything -- or of nothing. So again, we cannot predict what will happen.

2. The Inputs to NAME

Just as a clergyman "needs two inputs" for a christening -- a baby and a name -- NAME needs two inputs: a thing and a name.

As the "high priest of naming" on the Hurdles 2 videotape, Dr. Papert illustrated how to use these two terms when he named the dog FIDO. The dog is the *thing* of the name FIDO, and FIDO is the *name* of the thing dog. These two terms *thing* and *name* are important keys to understanding names in Logo.

Logo doesn't have dogs or babies or ships or people. Instead, there are words and lists. Nothing else. Every Logo-thing is either a word or a list, so the things we can NAME in Logo are words and lists.

Did you just ask, "What about numbers?" Numbers are words. They are "special" and slightly unusual words, but words nonetheless. PRINT WORD? 2 will confirm this. Appendix E gives more information about this.

3. Using and Teaching NAME

One of the earliest opportunities for using NAME is with colors. Some people don't like to bother remembering that green is 2 and red is -- whatever. Searching for the color number chart becomes a nuisance. Luckily, computers are *very* good at this. All we need to do is give names to the numbers.

> In fact, the computer's use of numbers and human use of names can be a good starting point for discussion about the differences between human and computational thinking.

```
NAME 2 "GREEN
```

NAME is a command. Its first input is any Logo-thing. Its second input -- the name -- must be a word. In this case, the Logo-thing is the number 2, and the name is the word GREEN. Both inputs are given directly.

Now that 2 has been given a name, we use : (called "dots") to say, "Go get the *thing* of this name."

```
PRINT :GREEN
```

That's why : is like two eyes. It goes and looks through Logo's workspace for the thing with that name, and reports this thing to the procedure that asked : for an input. In fact, you can call : a retriever since it behaves just like a dog that is told, "Go fetch."

NAME is highly syntonic, rich in connections to everyday life situations. We christen babies, launch ships, and give individual names to pets and rivers and countries and towns. In fact, whenever we need a way of saying "that particular *whatzit* over there," we give it a name.

* * *

NAME is not the only Logo primitive that gives names to things, however. You can also say:

```
MAKE "GREEN 2
```

MAKE feels more like algebra -- but if you don't know or didn't like algebra, that can sound as foreign as Latin. NAME "LION "SEYMOUR is very much like English: "We name this lion *Seymour.*" There is the lion, and we give it the name *Seymour.*

People who are comfortable with the MAKE idiom find it more natural in some situations. Which of the following do *you* find most natural?

```
TO CIRCLE :DIAMETER
MAKE "STEP PI * :DIAMETER / 360
REPEAT 360 [FD :STEP RT 1]
END

TO CIRCLE :DIAMETER
NAME PI * :DIAMETER / 360 "STEP
REPEAT 360 [FD :STEP RT 1]
END

TO CIRCLE :DIAMETER
REPEAT 360 [FD PI * :DIAMETER / 360 RT 1]
END
```

Use whichever form you prefer in your own work. But when you are trying to help children, NAME is usually better -- it is usually more syntonic to their own thinking and experience.

4. Reasons for Using Names

There are many reasons and opportunities for using names in Logo. Here are a few of them. You and your students can probably come up with even more.

Color-numbers	So Logo will remember and we can forget.
Positions	So we can move the turtle to them.

Headings	So we can control the direction the turtle faces.
Turtles	So we can talk to particular turtles or groups of them.
Long lists	So we don't have to type them over and over and over again.
RL and RC	So Logo will remember what the user types at the keyboard.

You now have many ideas for thinking about names. The next section describes different kinds of projects that use names, including the kinds of ideas listed above.

5. Name Calling

To use an idea, you must understand it -- and to understand an idea, you must use it. So learning is a spiral art that builds upon a learner's existing skills and understandings. The key for teaching something new is to find understandings that are simple enough to be accessible -- yet rich enough to support real uses. From real uses, fuller understanding will grow.

Naming is a good example of this. In Logo, the only things that can be named are words and lists. But a beginner doesn't have to know this in order to start using names and naming. When you use the instruction

```
NAME POS "HERE
```

you can think of this as saying "name the turtle's position HERE." You don't really need to know that POS is a reporter that reports a list.

Some Logo-things have that "touchable" quality that makes them real and concrete, and thus easy to name. We can point to a position on the screen. We can point a finger in the same direction as a heading. Colors are so important to our experience that we "see red" and have "green thumbs." And our entire language of nouns is built upon the principle of naming shapes. So giving names to things is a natural step, even for beginners.

Names for POS:

The videotape showed how naming can be used for drawing. One names the turtle's position, then uses this name later on as the input to SETPOS.

```
TO CHURCH
FRAME
STEPS
WINDOWS
TOWER
SPIRE
END

TO SPIRE
NAME POS "LBASE           The place where the turtle happens
RT 90 FD 20                 to be is given a name.
NAME POS "RBASE           This new place is also given a name.
BK 10 LT 90 PU FD 65
NAME POS "PEAK            And so is this last place.
PD
SETPOS :RBASE
SETPOS :LBASE
SETPOS :PEAK
HT
END
```

Notice that with SETPOS, the turtle's heading does not change. It doesn't turn to face the new position before it moves. It "jumps" -- just like in checkers or chess or hopscotch -- to the new position.

So POS can be used by people who don't know that POS reports a list, or that the two numbers in the list are the the turtle's x and y coordinates. Fuller understanding can come through use much sooner than you think.

The main reason for naming a position is so you can move the turtle to that position later on. It is an excellent way of drawing the last side of an unusual or irregular shape.

```
TO BOAT
NAME POS "HERE
LT 50 FD 25
RT 140 FD 100
RT 140 FD 25
SETPOS :HERE
END

TO ROCK
NAME POS "HERE
FD 25 RT 70
FD 15 RT 60
FD 30 RT 80
FD 20 RT 50
FD 10 RT 40
SETPOS :HERE
END
```

Naming POS can also create a starting place for a procedure. Perhaps you want to draw lots of castles -- or cats or stars or squiggles -- at different points on the screen. You can move the turtle around to different places and name these positions as you go.

```
BK 25 LT 90 FD 125
NAME POS "ONE
BK 75 RT 90 FD 50
NAME POS "TWO
BK 60 RT 90 FD 130
NAME POS "THREE
```

Maybe you decide that you want to put the third castle in a different place. Just move the turtle and NAME this new position "THREE.

```
FD 15
NAME POS "THREE
```

Now you have inputs for SETPOS in CASTLES:

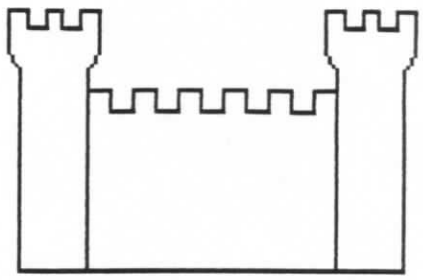
```
TO CASTLES
PU SETPOS :ONE PD CASTLE
PU SETPOS :TWO PD CASTLE
PU SETPOS :THREE PD CASTLE
END
```

A similar strategy will name points for different parts of one picture.

```
NAME POS "CASTLE
LT 90 FD 25
NAME POS "MOAT
```

Then each procedure can "know" where to start.

```
TO CASTLE
PU SETPOS :CASTLE PD
TOWER BATTLEMENTS TOWER
SETPOS :CASTLE
END
```



```
TO MOAT
PU SETPOS :MOAT PD
CIRCLE 150
PU LT 90 FD 25 RT 90 PD
CIRCLE 200
END
```

It's a very short step from NAME POS "HERE and SETPOS :HERE to understanding the list that POS reports.

```
CS SHOW POS
```

We see that the turtle's "starting" position -- the center of the screen -- is [0 0]. Then we can move the turtle and SHOW POS again.

If you say PRINT POS, only the numbers appear. Using SHOW helps demonstrate that POS reports a list because it "shows" the brackets of the list envelope. In fact, whenever you're looking for a bug that is doing something strange with lists, SHOW can be a very useful way of tracing what happens to the lists.

```
FD 50 SHOW POS
BK 70 SHOW POS
```

[0 50] and [0 -20] tell us that the second number in the list is the "up-down" number. Now if we say

```
HOME SHOW POS
```

we see [0 0] again. So HOME is similar SETPOS [0 0].[1] Now we can turn and look at the left-right dimension. A pretty obvious guess is that this will be the first number in the list POS reports.

```
RT 90
FD 50 SHOW POS
```

Bingo! [50 0]. And if we go back ...

```
BK 80 SHOW POS
```

[-30 0]. Going BACK 50 from [0 0] is like going FORWARD -50. So using and understanding points on the screen becomes a strong bridge to ideas like negative numbers and decimals. Moving the turtle to these points on the screen also makes these *x and y coordinates* syntonic and real instead of foreign and abstract.

[1] In some versions of Logo, HOME also sets the turtle's heading to 0.

Names for HEADING:

It's easy to lose track of which way the turtle is facing after doing even a short series of FDs and RTs. Also, adding a new subprocedure may change the way the turtle is facing for the next subprocedure, and everything that follows can wind up tilted.

These problems can be fixed with HEADING and SETHEADING. You can write all your subprocedures to start with SETH 0 (the turtle points straight up), and then simply turn the turtle LEFT or RIGHT to whatever direction you want.

Another strategy is to name the four "compass points" so that you have more than one possible starting direction. Naturally, there's no reason you have to stop at four.

```
TO COMPASS
SETH 0
RT 45 NAME HEADING "NE
RT 45 NAME HEADING "EAST
RT 45 NAME HEADING "SE
RT 45 NAME HEADING "SOUTH
RT 45 NAME HEADING "SW
RT 45 NAME HEADING "WEST
RT 45 NAME HEADING "NW
RT 45 NAME HEADING "NORTH
END
```

Names like SOUTH and NW are fairly easy to understand -- and thus accessible to very young students. Of course, some people prefer numbers like 270 and 135; others prefer names like WEST and SE. The name is then used as an input for SETHEADING:

```
TO TILT.SQUARE :SIZE
SETHEADING :NE
SQUARE :SIZE
SETHEADING :NORTH
END
```

Appendix A describes how you can make *state-transparent* procedures. By giving a name to the turtle's position and heading at the beginning of a procedure, it becomes easy to restore them at the end.

Names for Colors:

Earlier, NAME 2 "GREEN put the name GREEN into the workspace. When you SAVE your workspace, many -- but not all -- versions of Logo will save the names as well as the procedures. If the names do get saved, LOAD will put them back into the workspace together with your procedures.

But there are still two ways you might lose these names. You might NAME something else "GREEN, perhaps the list [TREES GRASS JEALOUSY]. And as Dr. Papert explained in the videotape, two things can't have the same name at the same time. Also, you might erase all the names from the workspace.[2]

The only foolproof way of making sure you have names for the colors is to *write a procedure.* Then you can run this procedure whenever you LOAD (or READ or GP) your file, and you can put it into any file you want.

```
IBM:                      APPLE:

TO NAME.COLORS            TO NAME.COLORS
NAME 0 "BLACK             NAME 0 "BLACK
NAME 1 "BLUE              NAME 1 "WHITE
NAME 2 "GREEN             NAME 2 "GREEN
NAME 3 "CYAN              NAME 3 "VIOLET
NAME 4 "RED               NAME 4 "ORANGE
and so on                 NAME 5 "RED
END                       END
```

[2]Apple & IBM Logo: ERNS, ERALL; Terrapin: ER NAMES, ER ALL; LogoWriter: CLEARNAMES.

You may also want to NAME 0 "INVISIBLE. Naturally, if your machine is neither IBM nor APPLE, use your own machine's color numbers.

A bug that often crops up here is that defining the procedure does not finish the job. One must also *run* the procedure. Just as the turtle doesn't move if you say FD 50 when you're defining a procedure, putting NAME 2 "GREEN into a procedure doesn't create the name. You have to run the procedure, either at top level:

```
NAME.COLORS
```

or by running a STARTUP procedure that has this instruction. When the names have been put into the workspace, then you can use the dotted names as inputs.

```
PRINT :RED
SETBG :GREEN
SETPC :BLUE

TO FLASH
REPEAT 5 [SETBG :GREEN WAIT 5
          SETBG :BLUE WAIT 5]
END
```

Sometimes a student will ask if 2 is "Logo's name" for green, since procedures that use colors must get numbers (or lists of numbers) as inputs. The answer is no. The colors themselves are not Logo-things -- and neither are turtles or shapes. Only words and lists are Logo-things, so only words and lists can be given names in Logo.

Names for Lists:

Another opportunity for introducing naming comes up when students are using long lists. Just as hunting for the chart of color numbers is a nuisance, so is typing

[CAT DOG LION FISH FROG ROBIN] more than once. It's much simpler to name the list.

```
NAME [CAT DOG LION FISH FROG ROBIN] "ANIMALS
```

The *thing* is the list [CAT DOG LION FISH FROG ROBIN]. The *name* is the word ANIMALS. When you say :ANIMALS, : goes and gets the thing named ANIMALS.

Now you can use the list as often as you want, and all you have to type is the dotted name.

```
NAME [MACAVITY FIDO JELLYLORUM SEYMOUR GREEN] "FRIENDS

TO RECOGNIZE? :SOMEONE
OP MEMBER? :SOMEONE :FRIENDS
END
```

Names for READCHAR and READLIST:

Naming is very often necessary when using READCHAR and READLIST. These reporters take input from the keyboard and give it to the procedure that called them. This allows a program to "interact" with the user and take different actions based on what the user types.

> Some versions of Logo also have READWORD, which is useful for reading numbers larger than 9. If yours does not, you can make one with OP FIRST READLIST.

READCHAR makes it simple to give choices. One way is to ask for a yes-or-no answer to each question.

```
TO ASK.YES.NO
PR [DO YOU WANT TO SEE MY FLOWER? Y OR N]
IF READCHAR = "Y  [FLOWER]
PR [DO YOU WANT TO SEE MY BIRDS? Y OR N]
IF READCHAR = "Y [BIRDS]
END
```

After each question, = checks to see if the letter typed at the keyboard is the same as its second input. If the user types Y, ASK.YES.NO will draw the picture. If the user types anything else, ASK.YES.NO simply goes to the next line.

CHOICES tries a different strategy. It wants the user to select one of three possibilities. But CHOICES has a bug.

```
TO CHOICES
PR [WHICH PICTURE DO YOU WANT TO SEE?]
PR [A = FLOWER]
PR [B = BIRDS]
PR [C = CITY]
IF READCHAR = "A [FLOWER]
IF READCHAR = "B [BIRDS]
IF READCHAR = "C [CITY]
END
```

CHOICES carries out the first four instructions without a hitch, then reaches

```
IF READCHAR = "A [FLOWER]
```

Here it is important to use the actor model in visualizing what happens. IF calls = , and = calls READCHAR. READCHAR watches for a letter from the keyboard (let's say the user types a C), then reports it back to = . = now has one input, the letter C. The second input is given directly, the letter A. Since the two are not the same, = reports FALSE to IF.

Now CHOICES goes to its next line

```
IF READCHAR = "B [BIRDS]
```

Just as HOUSE called an actor to run FRAME and then another to run ROOF in the Hurdles 1 Tape, CHOICES now calls a new actor to run this next instruction. This new IF calls a new = actor, which calls a new READCHAR actor, which watches for a letter from the keyboard -- and watches, and watches, and watches. The user's C was "swallowed up" by the first READCHAR.

Flow of control is the key to understanding this bug. Every procedure is "completely dumb" about what it does. It only follows instructions, line by line.

We could call this a Disappearing Bug since the C seems to vanish. As we'll see in the next section, the solution is to name READCHAR's report. Then it can be used as many times as you want.

Chocolate and Chocolate:

In the videotape, the computer sounds "really dumb" when it says, "Let's get both kinds, some chocolate ice cream and some chocolate ice cream." What happened? The procedure started with a greeting.

```
TO GREET
PR [HI, MY NAME IS LOW GO.]
PR [WHAT'S YOURS?]
PR SE
      FIRST READLIST
      [IS A NICE NAME.]
ICE.CREAM
END
```

So far, so good. If someone types `SANDY PITT`, GREET will just say `SANDY IS A NICE NAME.` If it typed the entire name, GREET might sound very stilted and formal. But now the fun begins.

```
TO ICE.CREAM
PR [WHAT KIND OF ICE CREAM DO YOU LIKE?]
( PR [MY FAVORITE IS CHOCOLATE. LET'S GET BOTH KINDS, SOME]
     READLIST
     [ICE CREAM AND SOME CHOCOLATE ICE CREAM.] )
END
```

The parentheses are used because PRINT is getting more than one input.

This is fine if the reply is STRAWBERRY or VANILLA or FUDGE SWIRL. But when Lauren says CHOCOLATE, the ICE.CREAM procedure looks dumb. At first, the solution seems obvious. If the user types CHOCOLATE, we want to do something different -- something like ME.TOO.

```
TO ME.TOO
PR [THAT'S MY FAVORITE TOO!]
PR [LET'S GO GET ENOUGH CHOCOLATE]
PR [ICE CREAM FOR BOTH OF US!]
END
```

A first attempt at fixing the "chocolate and chocolate" bug is often something like BETTER.ICE.CREAM.

```
TO BETTER.ICE.CREAM
PR [WHAT KIND OF ICE CREAM DO YOU LIKE?]
IF READLIST = [CHOCOLATE]
   [ME.TOO]
( PR [MY FAVORITE IS CHOCOLATE. LET'S GET BOTH KINDS, SOME]
     READLIST
     [ICE CREAM AND SOME CHOCOLATE ICE CREAM.] )
END
```

While this is a good step in the right direction, it also creates two bugs. Use the actor model to visualize the flow of control.

First let's see what happens when the user *does* type CHOCOLATE. IF calls = and = calls READLIST. READLIST reports [CHOCOLATE] to = and = reports TRUE, so IF runs the instruction list [ME.TOO]. When ME.TOO is done, ICE.CREAM runs the next line:

```
( PR [MY FAVORITE IS CHOCOLATE. LET'S GET BOTH KINDS, SOME]
    READLIST
    [ICE CREAM AND SOME CHOCOLATE ICE CREAM.] )
```

But that's not what we want it to do. When the user types CHOCOLATE, we want BETTER.ICE.CREAM to do ME.TOO and then stop! So IF's instructionlist needs to be [ME.TOO STOP].

The other bug appears when the user doesn't type CHOCOLATE but something else, say STRAWBERRY. At first, all is well. IF calls = and = calls READLIST. READLIST reports [STRAWBERRY] to = and = reports FALSE to IF. So BETTER.ICE.CREAM runs the next line:

```
( PR [MY FAVORITE IS CHOCOLATE. LET'S GET BOTH KINDS, SOME]
    READLIST
    [ICE CREAM AND SOME CHOCOLATE ICE CREAM.] )
```

Now PR looks for its inputs. [MY FAVORITE ... SOME] is a list given directly. But the parenthesis told PR continue looking for input -- and the next one is a procedure. So PR calls a new actor to do READLIST in the hope of getting an input. Now READLIST waits for input from the keyboard -- and waits, and waits, and waits.

This is our old friend, the Disappearing Bug. [STRAWBERRY] disappeared after the first READLIST reported it to = .

Naming the report from READLIST will fix this bug. So we NAME READLIST "FLAVOR, and then use that name to retrieve [STRAWBERRY].

```
TO BEST.ICE.CREAM
PR [WHAT KIND OF ICE CREAM DO YOU LIKE?]
NAME READLIST "FLAVOR
IF :FLAVOR = [CHOCOLATE]
   [ME.TOO STOP]
( PR [MY FAVORITE IS CHOCOLATE.
        LET'S GET BOTH KINDS, SOME]
     :FLAVOR
     [ICE CREAM AND SOME CHOCOLATE ICE CREAM.] )
END
```

Notice that :FLAVOR is needed in two different places: with the = testor, and where the original ICE.CREAM had its READLIST.

6. Names on Loan

Things with Two Names, Names with Two Things:

Can one thing have two names? People do. They have first names and last names -- sometimes nicknames as well. If I name this cat *Macavity*, you know which thing I mean. If it is also called *Jellylorum*, you still know which thing I mean.

```
NAME 2 "GREEN
NAME 2 "DUO

PRINT :GREEN
PRINT :DUO
```

But two things having the same name is a problem. Suppose the lion wants to be named *Seymour*. How will we know which thing you mean when you say "Seymour"? This me-thing or this lion-thing?

If there are two boys named John in one class, you invent a way of distinguishing between them. One becomes John and the other Johnny -- or a nickname is used or a last name or middle name. Sometimes these "temporary" solutions stick -- and they become a person's name for life.

In a sense, one of these boys has "lost" his name. In Logo talk, the thing of John has become the other person. An event like this will have certain psychological consequences. The boy's identity will shift and expand, like the cat with two names, but in most cases, he will not feel that he no longer exists. In Logo, however, losing one's name is a very serious matter.

```
NAME "LION "SEYMOUR
NAME "HEDGEHOG "SEYMOUR
```

The thing of SEYMOUR is now HEDGEHOG, not LION. LION's name has been given to HEDGEHOG, and it is as if LION no longer exists. Logo gets a lot of its power from being able to protect its Logo-things from losing their names.

It does this by creating two ways for HEDGEHOG to borrow the name SEYMOUR for a little while. One of these is the command LOCAL (which can only be used inside a procedure and is not available in some versions of Logo). The other is through the implicit naming of inputs to procedures. Either of these can be used to protect LION from losing the name SEYMOUR.

Inputs to Procedures:

We have seen how the command NAME attaches a name to a Logo-thing. When you run NAME, it "does a naming." But in Logo, namings are sometimes done automatically. This happens whenever a procedure has inputs. Look at this simple example:

```
TO GREET :SEYMOUR
PRINT SE "HI :SEYMOUR
END
```

The title line says that GREET needs one input, and it should name this input SEYMOUR. So when we say GREET "HEDGEHOG, an implicit naming is done. HEDGEHOG becomes the thing of SEYMOUR. This automatic naming that is done for a procedure's inputs is only a loan. The names are returned to the previous owner when the procedure is done.

We can watch this happen on the computer screen. First we'll make sure LION has his name:

```
NAME "LION "SEYMOUR
```

Then we'll see what happens when we run GREET -- and when we PRINT SE "HI :SEYMOUR, the instruction *inside* GREET.

```
GREET "HEDGEHOG
HI HEDGEHOG

PR SE "HI :SEYMOUR
HI LION
```

The term of the loan is the duration of the procedure. When the procedure gets to END, the names of these inputs revert to their previous owners.

This protective device of names on loan means that you don't need to worry about input names interfering with any other names that may have been set up.

Local Names:

These names on loan are also called local names. This means they are "local to" that particular procedure (and any of its subprocedures).

Here's a little teaser that shows how local names work. Notice the switch that occurs when WATCH.INPUT gives :CAT as input to SHOW.ME. SHOW.ME promptly names this input DOG!

```
NAME [CAT] "CAT
NAME [DOG] "DOG
NAME [LION] "LION

TO WATCH.INPUTS :CAT :DOG
PR [THIS IS WATCH.INPUTS AND MY]
PR SE [:CAT IS] :CAT
PR SE [:DOG IS] :DOG
PR SE [:LION IS] :LION
SHOW.ME :CAT
END

TO SHOW.ME :DOG
PR [THIS IS SHOW.ME AND MY]
PR SE [:CAT IS] :CAT
PR SE [:DOG IS] :DOG
PR SE [:LION IS] :LION
END
```

Use actors to WATCH.INPUTS [MACAVITY] [FIDO]. Have each actor put a sign on the bulletin board when it does its automatic naming -- and then take that sign down when it is finished. Then you can see how the names of the inputs change as they are passed from one actor to another.

7. Appendix A: State Transparent Procedures

A procedure is *state transparent* when the turtle ends up in the same position and facing the same direction as it started. Naming makes this easy to do. One names both the position and the heading.

```
TO FOOT
NAME POS "HERE
NAME HEADING "AHEAD
FD 50 RT 5
REPEAT 4 [TOE FD 2 LT 90]
TOE
RT 95 FD 50
RETURN :HERE :AHEAD
END

TO RETURN :POS :HDG
SETPOS :POS
SETH :HDG
END
```

RETURN uses two inputs: the position and the heading. But we can simplify this. We can combine them into one list -- which will be the input for RESTORE.[3]

[3] RESTORE is a primitive in LogoWriter; use a different name, such as RESTOR.

```
TO RESTORE :STATE
SETH LAST :STATE
SETPOS BL :STATE
END

TO VANE
NAME SE POS HEADING "START
FD 100 NAME POS "VANE
BK 100 RT 25
FD 100
SETPOS :VANE
RESTORE :START
END

TO WINDMILL
REPEAT 4 [VANE RT 90]
TOWER
END
```

But it's easy to forget: "Did I say POS HEADING, or HEADING POS?" So it's useful to have a turtle-state reporter. TST (short for "Turtle STate") lets you NAME TST "START -- just as you NAME POS "HERE -- and now you don't need to worry about whether POS or HEADING came first. TST will remember.

```
TO TST
OP SE POS HEADING
END

TO SQUIGGLE
NAME TST "START
PD
FD 50 RT 33 FD 20 RT 68 FD 20
PU
RESTORE :START
END
```

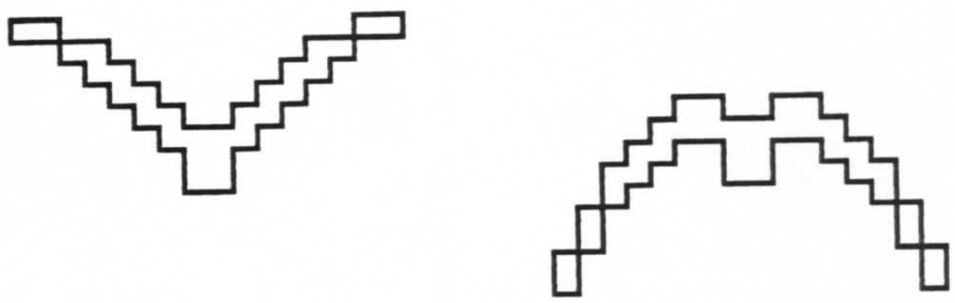
A bug that often creeps into RESTORE is SETPOS FIRST :STATE. It's very easy to make this mistake. One simply thinks, "The :STATE input is the POS and the HEADING, so RESTORE should set the position to FIRST and the heading to LAST of the :STATE."

SHOW :START lets you see the bug. :START is a list of three numbers -- the x coordinate, y coordinate, and heading -- something like [-125 40 355]. In order to SETPOS FIRST :START, the first element of :START would have to be a list instead of a number -- something like [[-125 40] 355]. If you prefer, you can make this kind of list for TST. Just use LIST instead of SE.

8. Appendix B: Names for Turtles and Names for Shapes

Several versions of Logo allow the turtle to take on new shapes, and some also have more than one turtle. Both of these abilities are ripe with opportunities for naming. A project that illustrates both uses is making a flock of birds fly across the screen.

Two shapes are needed: wings up and wings down. Perhaps something like this:



Trying to remember shape numbers can be just as frustrating as remembering color numbers, so we'll name them.

```
NAME 11 "UP
NAME 12 "DOWN

TO FLAP
SETSHAPE :UP
REPEAT 3 [FD 1]
SETSHAPE :DOWN
REPEAT 3 [FD 1]
END
```

Now we need a group of turtles to be the birds. And we can TELL them to fly.

```
NAME [0 1 2] "BIRDS

TO FLY
TELL :BIRDS
SETSHAPE :UP
ST
PU SETPOS [0 50]
RT 90
EACH [FD 20 * WHO]
REPEAT 50 [FLAP]
END
```

9. Appendix C: The Muffin Bug

Sooner or later, everyone who learns Logo meets the Muffin Bug. It crops up when the learner thinks, "Aha! The way to do this is to put a name into a list." This list may be the input for SETPOS or SETPEN or SETTC or FIRST or BL or LPUT or = or MEMBER? or SETCURSOR -- or perhaps a procedure written by the learner. But sooner or later, everyone does something like

```
SETPOS [50 :UP]
```

and then becomes frustrated when Logo complains. "But I did *too* give it the inputs it needs. See? It's right there -- :UP!"

The problem is quite simple. The basic idea is a good one, but the method overlooks one significant detail.

Suppose someone hands you a freshly-baked muffin, butter melting on its top, the aroma wafting towards your nostrils, and you eagerly pop it into your mouth -- only to discover that you are crunching down on this wad of paper that tastes like glue and sawdust and ink. When you spit out this paper in disgust, it says :INGREDIENTS. That's not what the packaging led you to expect, so of course you complain.

The problem with SETPOS [50 :UP] is exactly the same. When " or [ ] are used, the procedure is expecting a real muffin -- not the name of some other Logo-thing, and not a recipe. SETPOS expects a list of numbers, and it did get a list -- but of a number and a name. It doesn't matter that the thing of that name may well be a number. What you gave it was the name.

The solution is to use one of the list-making reporters to create a list from the inputs 50 and :UP.

```
SETPOS SE 50 :UP
```

Notice that SETPOS :MOAT worked just fine because MOAT is the name of the entire list. SETPOS [:MOAT] is the Muffin Bug -- and will also produce a complaint.

But what about REPEAT 4 [FD :SIDE RT 90]? Isn't this a name inside a list? Shouldn't this give us the same complaint?

No, because [FD :SIDE RT 90] is an *instruction list.* The name SIDE is used to give an input to FD, not to REPEAT. So you can put a name inside an instruction list *when it is the input to a procedure inside the instruction list.*

```
REPEAT 4 [FD :SIDE RT 90]        REPEAT gets both inputs directly.

NAME [FD :SIDE RT 90] "SQUARE
REPEAT 4 :SQUARE                 REPEAT gets its list indirectly,
                                 with the dotted word SQUARE.
TO SQUARE
OP [FD :SIDE RT 90]
END

REPEAT 4 SQUARE                  REPEAT gets its list indirectly,
                                 with the unadorned word SQUARE.
```

A procedure can get its input either directly or indirectly. If you give an input directly, use the " or [ ]. The procedure will expect a "real muffin," a word or list that it can use exactly as is. If you don't have a direct input, you must use either a name or a procedure that will "make the muffin."

10. Appendix D: Things of Names

Sooner or later, everyone also encounters another kind of problem. Suppose you can't just put the : in front of the name, but you still want to retrieve the thing of that name.

For example, let's suppose you want to make a quiz to see if your students know the parts of a flower. (This example is based on a teacher's work at the Hennigan School in Boston.) You can name the turtle's position as it draws each part.

```
TO PETAL
LT 30 REPEAT 30 [FD 2 RT 3]
NAME POS "PETAL
RT 90 REPEAT 30 [FD 2 RT 3]
RT 30
END
```

And now you do the same with STEM and LEAF. These names -- PETAL, STEM and LEAF -- can be the inputs to another procedure. The idea is to move the turtle to a particular position, ask what part the turtle is on, and then see if the student's answer is the same as the name of that position. A first attempt would be something like this

```
TO PLANT.QUIZ
DRAW.PLANT
ASK.FOR "PETAL
ASK.FOR "STEM
ASK.FOR "LEAF
END

TO ASK.FOR :PART
PU SETPOS :PART
PRINT [WHAT PART IS THE TURTLE ON?]
IF :PART = READWORD [PR [THAT'S RIGHT.] STOP]
PRINT [SORRY, TRY AGAIN.]
IF :PART = READWORD [PR [THAT'S RIGHT.] STOP]
PRINT SE [THE ANSWER IS] :PART
END
```

This idea almost works. But on the first run-through, SETPOS doesn't like the word PETAL as input. What we need is the *thing* of PETAL -- the position that was named PETAL -- but we can't just put the dots in front and say :PETAL.

There is, however, a primitive that works just like : but doesn't have to be "glued" to the name like dots. This primitive is the reporter THING.

```
PRINT THING "GREEN          PRINT :GREEN
SETCOLOR THING "GREEN       SETCOLOR :GREEN
SETBG THING "GREEN          SETBG :GREEN
```

THING lets us fix the bug in ASK.FOR. What we need is :PETAL and :STEM and :LEAF -- which is the same as THING "PETAL and THING "STEM and THING "LEAF -- and these are the same as THING :PART.

```
TO ASK.IF :PART
PU SETPOS THING :PART
PRINT [WHAT PART IS THE TURTLE ON?]
IF :PART = READWORD [PR [THAT'S RIGHT.] STOP]
PRINT [SORRY, TRY AGAIN.]
IF :PART = READWORD [PR [THAT'S RIGHT.] STOP]
PRINT SE [THE ANSWER IS] :PART
END
```

So you can think of THING as being just like : or you can think of : as a short form of THING. Both do the same job -- they go and get the thing of the name.

THING does have one big advantage over : however. THING can use reporters to get its input, : cannot. PRINT THING WORD "PET "AL. When you use dots, you must also have the name. PRINT :PETAL.

11. Appendix E: About Numbers

We've said that in Logo, a number is a special kind of word. This is not really a strange idea, because in English, a number is also a special kind of word. It can be spelled with the characters 0 through 9 as well as with the alphabet. Nineteen and 19. Three thousand four hundred eighty-seven and 3487. And in both languages, numbers have special properties.

They can of course be added and subtracted, multiplied, divided, and so on -- which words like *hedgehog* and *christening* cannot. Two plus two in English and 2 + 2 in Logo do essentially the same thing. And if we NAME 2 "TWO, we can even get the same result with :TWO + :TWO.

In Logo, numbers are "self-quoting" words. This means that you don't have to put quotes in front of them.

```
PRINT 4
PRINT "CAT

NAME 4 "FOUR
NAME "CAT "FUZZY
```

When a word like CAT or FOUR or FUZZY is given as input, it must be quoted. Any unadorned word is treated as a procedure, so if we said NAME CAT FUZZY, NAME would run CAT to get its first input. But Logo "knows" that any word that is made up only of the digits 0 to 9 (it can also have a decimal point) is a number -- and it acts as if the quote marks were already there.

The reason for this is that instructions like FORWARD "50 and PRINT "3 + "8 would look strange to a beginner's eye. Of course, one could simply explain that the quotes must always be used to give a word directly as input, but then one also has to explain about numbers being words. This would create two extra and unnecessary steps to understanding these otherwise simple instructions.

There are three testers that can distinguish among numbers, words and lists. They are especially helpful in creating procedures like a PIG.LATIN reporter -- something that can take any word or list as input, then test to see what sort of input it got and send it on to different subprocedures that take different actions with words and lists.

```
PRINT NUMBER? 5
PRINT NUMBER? "CAT
PRINT NUMBER? [CAT]

PRINT WORD? 5
PRINT WORD? "CAT
PRINT WORD? [CAT]

PRINT LIST? 5
PRINT LIST? "CAT
PRINT LIST? [CAT]
```

There are only two exceptions to the rule that a word given as direct input must be quoted. The first is numbers, which are self-quoting words. The primitive TO is the second exception. When we say TO SQUARE, TO's first action is an "automatic quoting" of its input, SQUARE.[4]

Of course, a price must be paid for these inconsistencies. They create an opportunity for confusion in learning about quotes. Some people prefer to bypass this potential confusion by using a "pure form" of Logo. So if you want, you can say FORWARD "50 and PRINT "3 + "8 and TO "SQUARE. All of these instructions will work. (In most versions of Logo, however, TO 3 will *not* work. One cannot use numbers as names of procedures.)

In Logo, CAT and 123 are both words. You can also make words in which numerals and letters are mixed. In English, R2D2 and C3PO are familiar and acceptable only because they are names -- and names of futuristic science-fiction robots at that. We can puzzle out the meanings of 2MORROW and DE4MED and L8, but these "words" look very strange. Indeed, the numbers here are only substitutions for the sounds of the names of the numbers, and in fact play havoc with the phonetics of usual English spellings.

[4]In Terrapin Logo, ED and PO and ER also "automatically quote" their input when it is a word.

12. Appendix F: Making Muffins

Writing procedures like MUFFINS can help the learner build bridges between "the real world" and Logo programming -- and such bridges are crucial for syntonic learning. MUFFINS also has three subprocedures -- COMBINE, BAKE, and DONE? -- that suggest a variety of different procedures and can stimulate class discussions.

DONE? is clearly a reporter that tests to see if something is true or false. But it doesn't take an input. How does it know what to test?

One strategy is to re-write MUFFINS so that it gives DONE? an input.

```
TO DONE? :FOOD
OP (TEMPERATURE :FOOD) > 155
END
```

Another strategy is to write DONE? so that it "knows" what to test, just as HEADING has a built-in knowledge of how to find out what direction the turtle is facing.

```
TO DONE?
OP (TEMPERATURE :FOOD) > 155
END
```

Some people prefer to give DONE? an input. Then they can be sure that TEMPERATURE will get the input it needs. If you know that :FOOD will exist when MUFFINS reaches DONE?, then you can use the second version.

Neither strategy is "better" than the other. Both have advantages and disadvantages. The second is useful for minimizing "clutter" in your procedures. It can also get you into trouble. You must make sure that you create :FOOD -- and not :BATTER or :STEW or nothing at all -- before MUFFINS reaches DONE?.

Since COMBINE is already doing something with the ingredients [FLOUR MILK EGGS BUTTER [BAKING SODA]], this would be a good place to create :FOOD.

```
TO COMBINE :INGREDIENTS
PRINT SE [MIX TOGETHER THE] FIRST :INGREDIENTS
PRINT SE [WITH THE] BF BL :INGREDIENTS
PRINT SE [AND THE] LAST :INGREDIENTS
PRINT [UNTIL WELL BLENDED.]
PRINT [POUR INTO MUFFIN TINS.]
NAME :INGREDIENTS "FOOD
END
```

A slightly different strategy is to make COMBINE a reporter. It would still PRINT the instructions for combining the ingredients into a batter and pouring this batter into the muffin tins. But instead of NAME :INGREDIENTS "FOOD, it would then OP :INGREDIENTS, and :FOOD would be created in the MUFFINS procedure.

```
TO MUFFINS
NAME COMBINE [FLOUR MILK EGGS BUTTER [BAKING SODA]]
     "FOOD
BAKE 350 30
IF DONE? [OP "MUFFINS]
BAKE 350 5
OP "MUFFINS
END

TO BAKE :TEMP :TIME
SET.OVEN.TEMPERATURE :TEMP
WAIT :TIME
SET.OVEN.TEMPERATURE 0
END
```

In a sense, this is a first step towards making a robot that would bake real muffins. Many questions can arise from a project like this. Would testing the temperature be enough for a real DONE? procedure, or would a chemical analysis of the crumb be required? What about the COMBINE procedure? This one simply PRINTed the steps, but a real robot would need a BLENDED? tester and a POUR procedure. It would need to know how to MIX its different kinds of ingredients so that the egg shells are not mixed into the batter as well as the yolk and white.

Projects like MUFFINS can start class discussions about how procedures are like real-world recipes -- either for baking muffins or building bridges or researching quasars or learning tennis. There are many steps in writing a procedure that are like the steps in building a new habit. But people forget, and computers remember. So there are important differences too.