6

Programming Ideas

Adding Numbers

This section is about how to think and talk about the process of making a program. I developed the general approach while introducing elementary school children to computation. But the ideas that are good for children are good for other beginners, and perhaps for some experienced programmers. Variants of the example used here have been used with seventh graders, with college undergraduates, and with teachers. They illustrate a style of programming project, a style of programming language, and a meta-language or style of talking about programming as well as doing it. There is no suggestion that this style is uniquely correct. My message is on a different plane; I mean to assert the importance of paying more attention in the pedagogy of computation to such questions of style.

The problem is very recursive. I want to talk about programming, but I need to invent a way to talk about talking about programming! One way would be to give extracts from real dialog. But this is too cumbersome. Instead I shall condense real dialog into a kind of monolog about developing a program. The monolog gives an impression of one way I know how to think about developing a program. There is nothing very original about this way of thinking. The point I am making is about the technique of getting it out of our heads and into the pedagogy of teaching beginners.

In the discussion I carry with me a computational model in which there are little people, agents, experts in the computer that I can call on to help in thinking about the flow of my program, and, thus, in debugging my program. Keeping this model in mind helps me articulate what jobs need to be done and what procedures I need to get those jobs done. It also helps me figure out how these procedures interact with one another, how they report back what they have found out or constructed. Furthermore, as I debug my program and its individual procedures I talk again to these little people and get them to act out each procedure step by step, instruction by instruction.

The Project

We pretend the computer is ignorant of arithmetic and create an operation that will add two integers. No Logo arithmetic operations may be used. An apparent exception might seem to be `EQUALP` (=), but it is used to compare

---

By Cynthia Solomon.

whether two Logo words or letters are the same. (It is an identity operator.) So +, <, >, COUNT, *, /, and REMAINDER are prohibited. Two implications arise from posing this project. One is that an addition operation can be decomposed into smaller procedures. The other is that numbers are really just words asked to play special roles.

This project generates interesting discussions. It really frees one's thinking about numbers and operations and primitiveness. It is true that arithmetic is a very necessary part of any computer's hardware, but the hardware is made up of "logical units" that are based on the same ideas we will investigate. How do computers really add? It's in their hardware. It's built into the system. It's hardwired. Is addition "hardwired" into *our* system? Are we like computers and so if a wire is loose we can't do it? What about addition among children? Is it really a built-in capacity, or are there pieces of knowledge that are acquired? Maybe we are so familiar with addition that we forget its components. In fact, addition must rely on lots of procedures.

Let's look at this project. Try to situate this particular task into a familiar environment. We have to imagine that there are no arithmetic operators available to us and that there are no arithmetic experts already existing in Logo. We want to make up an addition operation so that we can say

```
PRINT  ADD 16 532
```

and the computer will say

```
548
```

Yes, addition is a familiar operation and it's easy for us to hand-simulate its job. But what if we had to tell a little person in the computer how to add? Where do we start? We might ask ourselves if we know of a similar experience. What we have to do is "teach the computer" to add—just as we might teach a person! Well, now, teachers teach kids to add; we were once those kids. How did we learn—can we give ourselves some tips? (But I thought it was hardwired and teacher just . . .)

At this point in past discussions two suggestions emerge. Teachers say we have to teach the computer the "number facts" and computerists say we have to build a 10×10 table. Great, I say, a beginning. I ask teachers how we teach the number facts and what are they and how many of them there are. I ask computerists if a 10×10 table is large enough and how we organize it. The teachers will face these issues too. After all, making a table is a way of "teaching" number facts.

What kind of table and what are number facts? A table of the sums of the first 100 numbers is very limited, and building a larger table is still very limited. Is that what I have in my head? Isn't there a key idea or two that I could build on without exhausting the computer's memory?

Do children learn "number facts" like $16 + 20 = 36$ as a primitive notion, or is there a more fundamental idea underlying it all? What do kids learn about numbers? They learn their relationship to each other. They learn to order them. *Sesame Street* teaches kids to count from 1 to 20. Kids learn to recognize the digits and their order. They learn that one is the name of 1 and eleven is the name of 11 and one hundred eleven is the name of 111. They learn that 11 is different from 2; they learn that 10 has been added to 1. But there is another way of discussing that change. Let's say 1

is a special word. We can create a new word by putting it together with another. So `WORD 1 1` is `11`, or eleven. Concatenating is a way of changing numbers.

Let's return to learning to recognize digits and ordering them. That indeed is what we have to tell the computer and build upon. You might say we want to teach the computer to count. On the other hand, it does us no good to see the computer spew out numbers from 1 to 500. We want the computer to know *how* to count. Think of what's involved in counting. How many symbols are there? In one sense there are ten, 0 1 2 3 4 5 6 7 8 9; but there are many constructions like 13 or 444; then there are also funny changes such as from 9 to 10, from 19 to 20, from 29 to 30, and so forth.

We want to teach the computer that 7 comes after 6 and 10 follows 9 and so on. Some of it is tricky. But look, the only elements used in a base-10 number system are 0 1 2 3 4 5 6 7 8 9. If kids learn how to use those ten symbols in thousands of different ways, surely we can teach the computer. There must be some rules that specify what to do to produce the "next number in sequence."

That's what we have to do. That is our plan of attack. Tell the computer what the basic elements (our data base) are. Then develop rules of behavior so that we can make the computer give us our number plus 1, that is, the next number. If the computer can do that, it knows how to count.

What is knowing how to count? Here's a computerist model: There is "in the head" a collection of little people, experts capable of doing a whole bunch of things like spewing numbers out, but also capable of conceiving questions like what comes after this or before that. The computer, like children, learns to recognize the digits, how to order them, and then how to use them to make other numbers.

Okay, let's make a procedure that knows about digits. For example, if it receives the input 3, it will output 4. It will add 1 (in some mysterious way) to its input.

```
ADD1 3 ---> 4
ADD1 7 ---> 8
```

We Make ADD1

There are a couple of ways (at least) to do this. People who suggested "teaching number facts" or making tables, of course, had the right idea. There are different ways of constructing tables. For example:

```
TO DIGITTABLE :DIGIT
IF :DIGIT = 0 [OP 1]
IF :DIGIT = 1 [OP 2]
IF :DIGIT = 2 [OP 3]
IF :DIGIT = 3 [OP 4]
IF :DIGIT = 4 [OP 5]
IF :DIGIT = 5 [OP 6]
IF :DIGIT = 6 [OP 7]
IF :DIGIT = 7 [OP 8]
IF :DIGIT = 8 [OP 9]
IF :DIGIT = 9 [OP 10]
END
```

We can also look at the ordered list of digits [0 1 2 3 4 5 6 7 8 9] as another representation that has the same effect if we have a NEXT type of operation.

```
NEXT 0 [0 1 2 3 4 5 6 7 8 9] ---> 1
```

NEXT will output the next element in the list after the one specified.

```
TO ADD1 :DIGIT
OP NEXT :DIGIT [0 1 2 3 4 5 6 7 8 9]
END
```

Why do I suggest this way? It is a more general method. This process will work for any base; all that needs to be changed are the elements of the list!

We Design NEXT

NEXT must supply ADD1 with a word. ADD1 will then send the word out as its answer. From the example of NEXT at work, we see that NEXT is given two inputs, a word like 0 and a list of words. NEXT tells its helpers to look for the word in the list. They send back the word following it in the list.

```
IF :WD = FIRST :LIST [OP FIRST BF :LIST]
```

If :WD doesn't match with :LIST's first word, one of NEXT's helpers just crosses the first word off the list and turns the job over to someone else.

```
OP NEXT :WD BUTFIRST :LIST
```

Check this procedure out.

```
TO NEXT :WD :LIST
IF :WD = FIRST :LIST [OP FIRST BF :LIST]
OP NEXT :WD BF :LIST
END
```

Notice there is a potential bug. What if :WD is not in :LIST? Let's remember the bug, but postpone dealing with it for the moment.

Let's try ADD1 now. Give it a thorough testing. You could exhaustively try each digit because there are only ten. Another strategy is to choose extremes like 0 and 9.

```
PR ADD1 0
1
PR ADD1 1
2
PR ADD1 2
3
PR ADD1 3


PR ADD1 9
FIRST DOESN'T LIKE [] AS INPUT IN NEXT
```

It's logical that there is a bug. After all, 9 is the last element of the list. So there is more work to be done; we have to teach ADD1 that 9 + 1 = 10.

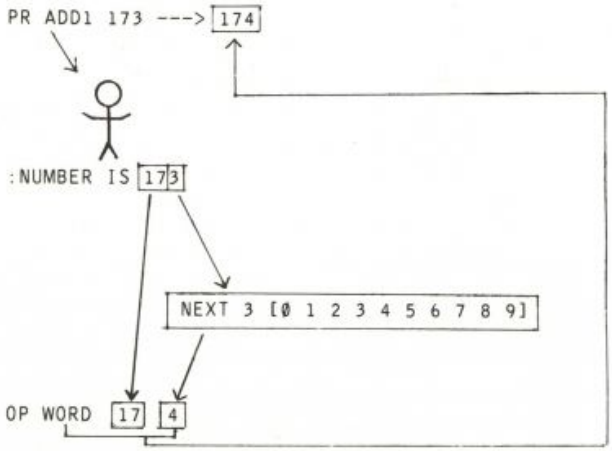
ADD1 will work on any number that doesn't end in 9 if we make one small change! Look, all numbers not ending in 9 behave like digits when you add 1 to them.

```
123 ---> 124
13 ---> 14
```

The only digit that changes is the LAST, so ADD1 merely makes up a new word by replacing LAST :DIGIT with NEXT LAST :DIGIT. Let's be opportunistic—seize the chance, change ADD1 and call its input NUMBER.

```
TO ADD1 :NUMBER

OP WORD BL :NUMBER

          NEXT LAST :NUMBER [0 1 2 3 4 5 6 7 8 9]

END
```

Now we trace through this procedure using the little person metaphor. As a reminder, I draw a stick figure.



(So we thought ADD1 was only good for nine inputs. Suddenly we see it's good for how many—millions? infinitely many? nine-tenths of all the numbers?)

Now ADD1 works on all numbers that don't end in 9. Would it work if we pretend 10 is a digit and add it to the list given NEXT—that is, [0 1 2 3 4 5 6 7 8 9 10]? Then

```
PR ADD1 9
10
```

but

```
PR ADD1 19
11Ø
```

instead of 20!

So putting 10 in the list did not really help. This *nines* bug is not cured so quickly. This issue is really about what to do with the "carry" when adding numbers. If a number is 9 then the answer is 10, but if a number ends in 9 we want to carry one to add it to the next digit of the number. Now, how can wishful thinking help? How can we make use of what we just did? Let's see how we do it. Try 179:

```
179 + 1 ---> 18Ø
```

We turn the 9 into a 0 and add the 1 to the 17. We get 18 . . . and don't forget to glue the 18 and the 0 back together.

Make a special check for LAST :NUMBER being 9. Then replace the 9 by 0 and ADD1 to BL :NUMBER.

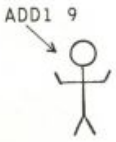
```
IF 9 = LAST :NUMBER [OP WORD ADD1 BL :NUMBER Ø]
```

So

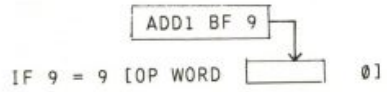
```
TO ADD1 :NUMBER
IF 9 = LAST :NUMBER [OP WORD ADD1 BL :NUMBER Ø]
OP WORD BL :NUMBER NEXT LAST :NUMBER [Ø 1 2 3 4 5 6 7 8 9]
END
```

Now we try ADD1 with "little people."

```
PRINT ADD1 9
```




```
:NUMBER IS 9
```



but BF 9 ---> "

We can fix this bug by making another special test

```
IF 9 = :NUMBER [OP 10]
```

as the first instruction in ADD1. Now

```
PR ADD1 179
180
```

and

```
PR ADD1 9999
10000
```

What luck! Perhaps you thought that the first 9 on the left would give trouble. But we lucked out (or were super smart!).

Adding Two Numbers

Now that we can add 1 to any number, we can really add any number to any other.

It's simple if we think of the kinds of procedures we know about. Some procedures operate on their inputs until they are empty or until a thing has been found. Other procedures do a job for a specified number of times. We can think of the next stage in our project as *adding one* to an input for a declared number of times.

```
6 + 4 is ADD1 ADD1 ADD1 ADD1 6.
```

Typically, counter procedures count down to 0 and then they know the job is done. But they use subtraction, and we are trying to invent addition without using any of Logo's built-in arithmetic operations. We can teach the computer to *subtract one.*

If we had a SUB1 procedure, then

```
TO ADDUP :NUM1 :NUM2
IF :NUM2 = 0 [OP :NUM1]
OP ADD1 ADDUP :NUM1 SUB1 :NUM2
END
```

Making a procedure for subtracting 1 is really easy because we have already thrashed through the difficulties encountered in ADD1. How can we use what we know about ADD1 to describe a SUB1? Let's look at a concrete situation.

```
PR SUB1 2
1
PR SUB1 9
8
PR SUB1 1
0
```

Can SUB1 use NEXT?

If we want NEXT 1 [. . .] to be 0, how should the list be ordered? If we leave the list as [0 1 2. . .9], then NEXT 1 would output 2. It should output 0. Reverse the list. Then NEXT 1 [9 8 7 6 5 4 3 2 1 0] outputs 0.

So

```
TO SUB1 :NUMBER
OP WORD BL :NUMBER NEXT LAST :NUMBER [9 8 7 6 5 4 3 2 1 0]
END
```

Try SUB1.

It works! As long as the numbers don't end in what? Nine is okay. Why? The digit that is the LAST position of the list given to NEXT is the problem digit. That is when a "carry" or a "borrow" takes place. So SUB1 must take special measures when LAST :NUMBER is 0.

```
TO SUB1 :NUMBER
IF 0 = LAST :NUMBER [OP WORD SUB1 BL :NUMBER 9]
OP WORD BL :NUMBER NEXT LAST :NUMBER [9 8 7 6 5 4 3 2 1 0]
END
```

Now ADDUP works but very slowly and sometimes it needs too many people to complete the job. Look, ADDUP 9999 9999 requires 9999 little people.

Is there a shortcut? Yes. Let's treat the numbers as words and add the LAST digit of each number to ADDUP until :N1 and :N2 have been added together.

```
TO ADD :N1 :N2
IF EMPTYP BL :N1 [OP ADDUP :N2 :N1]
IF EMPTYP BL :N2 [OP ADDUP :N1 :N2]
OP WORD ADD BL :N1 BL :N2 ADDUP LAST :N1 LAST :N2
END
```

This is ideal but won't work very often. Do you know when it works?

```
PR ADD 34 21
55
PR ADD 2468 321
2789
```

but

```
PR ADD 19 19
218
```

The *carry* bug has to be dealt with. How can ADD tell if there is a carry? A carry means that ADDUP will send back two digits (1 and something). That makes it easy. ADD needs to test whether the result from ADDUP is one or two digits long. ADD uses ADDIT to help and now looks like:

```
TO ADD :N1 :N2
IF EMPTYP BL :N1 [OP ADDUP :N2 :N1]
IF EMPTYP BL :N2 [OP ADDUP :N1 :N2]
OP ADDIT ADDUP LAST :N1 LAST :N2 BL :N1 BL :N2
END

TO ADDIT :SUM :N1 :N2
IF EMPTYP BF :SUM [OP WORD ADD :N1 :N2 :SUM]
OP WORD ADD :N1 ADD1 :N2 BF :SUM
END
```

In some sense this project is completed. We have constructed an addition operation, and it works on positive integers. There are many extensions we could pursue. For example, handling negative numbers would probably necessitate making a subtract operation.

EXTENSIONS

In discussing setting up the table at the start, I mentioned the possibility of generalizing this scheme so that the operation would add numbers of other bases. What about fractions or decimals? But what about looking at a more general question? There are many arithmetic operations like MULTIPLY, DIVIDE, EXPONENTIATION, REMAINDER, BASE, CONVERSION, FACTORIAL. There are also others, like the Logo operation COUNT that outputs the length of a word or a list, and the predicates > (greater) and < (less). Any of these could be implemented as extensions to this project.

Although we might be able to write procedures to perform many of these operations, the process would probably be uncomfortably slow. This leads to the question: Are there some arithmetic operations that we couldn't define without special hardware or without special software? What operations are primitive? Imagine writing WORD or LIST or FIRST or BUTFIRST. What would be required? Is the derivation too clumsy? The answers to these questions will undoubtedly change as the contexts in which they arise change.

PROGRAM LISTING

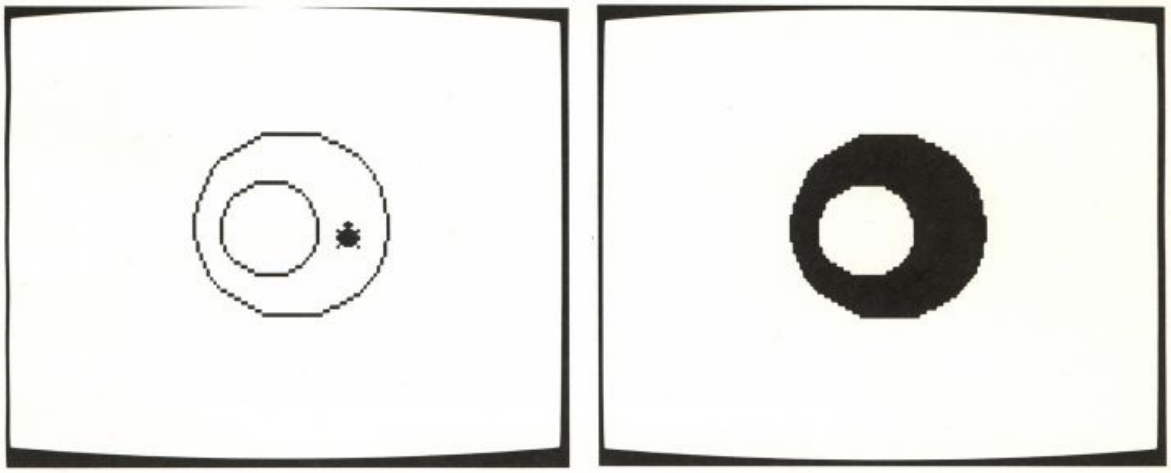
```
TO ADD :N1 :N2
IF EMPTYP BL :N1 [OP ADDUP :N2 :N1]
IF EMPTYP BL :N2 [OP ADDUP :N1 :N2]
OP ADDIT ADDUP LAST :N1 LAST :N2 BL ►
     :N1 BL :N2
END

TO ADDIT :SUM :N1 :N2
IF EMPTYP BF :SUM [OP WORD ADD :N1 :N2 ►
     :SUM]
OP WORD ADD :N1 ADD1 :N2 BF :SUM
END

TO ADDUP :NUM1 :NUM2
IF :NUM2 = Ø [OP :NUM1]
OP ADD1 ADDUP :NUM1 SUB1 :NUM2
END

TO ADD1 :NUMBER
IF 9 = :NUMBER [OP 1Ø]
IF 9 = LAST :NUMBER [OP WORD ADD1 BL ►
     :NUMBER Ø]
OP WORD BL :NUMBER NEXT LAST :NUMBER ►
     [Ø 1 2 3 4 5 6 7 8 9]
END
```

```
TO NEXT :WD :LIST
IF :WD = FIRST :LIST [OP FIRST BF ▸
     :LIST]
OP NEXT :WD BF :LIST
END
```

```
TO SUB1 :NUMBER
IF 0 = LAST :NUMBER [OP WORD SUB1 BL ▸
     :NUMBER 9]
OP WORD BL :NUMBER NEXT LAST :NUMBER ▸
     [9 8 7 6 5 4 3 2 1 0]
END
```

Fill

FILL is a program to fill in solid areas on the graphics screen.



Before After

Figure 1

To use FILL, position the turtle inside the area you want to fill. Then type the command FILL with no inputs. The area the program will fill is bounded by lines drawn with any pen.* For example, try this:

```
CS
REPEAT 4 [FD 80 RT 90]
PU
SETPOS [20 20]
FILL
```

*If the screen dot at the turtle's position was already drawn with one of the pens, then FILL treats that pen as the background color for filling. So if you have a filled-in area on the screen, you can draw a picture within that area and fill the inside of the picture using another color.

By Brian Harvey.

to draw a solid, filled-in square. The SETPOS instruction is necessary to position the turtle inside the square, rather than on its edge, before using FILL.

Note: If you have a 16K Atari computer, you should use the number 8192 instead of 16384 in procedure POSADDR.

How It Works: Overview

Figure 2 shows a sort of eccentric doughnut with the turtle positioned between the two circles, so that the doughnut shape will be filled. The program begins by filling horizontally from the turtle's initial position, in both directions (figure 3). It remembers how far it got, to set left and right limits for what comes later. Then it starts moving up (figure 4), filling horizontally at each level.

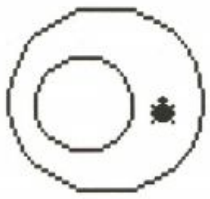


Figure 2

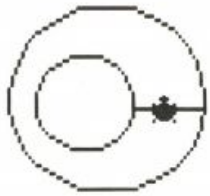


Figure 3

Figure 4

But when a newly filled line extends beyond the previous line (as illustrated by the left edge of the filled area in figure 4), the program also checks for an unfilled space below the new horizontal stretch. If it finds one, it starts filling downward in that new area (figure 5). This search for new areas works from left to right on each line, so (figure 6) the program continues moving downward below the inner hole until it reaches the bottom (figure 7).

Figure 5

Figure 6

Figure 7

Then it starts moving up into the newly discovered area to the right of the hole (figure 8), and when that area is filled, the program continues its interrupted upward filling of the top area (figure 9). The final result is shown in figure 10.

Figure 8

Figure 9

Figure 10

Screen Coordinates and Turtle Steps

The graphics screen consists of about 15,000 small dots, in a rectangular array of 96 rows and 160 columns. Logo draws lines on the screen by "turning on" some of these dots. To fill an area, we must also turn on dots in this array.*

When you use the `FORWARD` command, the distance measured in "turtle steps" is not the same as the number of screen dots (or *pixels*) through which the turtle passes. There are two reasons for this difference. The first reason is that the distance between two vertically adjacent pixels is greater than the distance between two horizontally adjacent pixels. If Logo measured distances in pixels, squares would come out looking like tall rectangles. Instead, Logo uses the *aspect ratio* (the ratio of a horizontal pixel distance to a vertical pixel distance) as a scale factor for vertical turtle steps. The second reason is that both vertical and horizontal turtle steps are scaled by a factor of two, so that 100 turtle steps is a reasonable distance on the screen.

The reason this scaling of distances is important for the `FILL` project is that we're going to have to think in terms of pixels, not in terms of turtle steps. Remember that the overall task of the program is to move along the screen looking for the border of the region we want to fill. In other words, the program must look at a position on the screen to see if that position is in the background color. If so, the program should fill in that position and move on to the next. Suppose we wrote the program in terms of turtle steps. (We'd then use `FORWARD 1` to move from one position to the next.) Since a turtle step is smaller than the distance between pixels, two consecutive turtle positions will often occupy *the same pixel* on the screen! After filling in the first position, we'd move on to the next position and think we'd

*For more details about the screen array, see the Savepict and Loadpict project.

hit the border, because the screen dot would no longer be in the background color.

The approach I took in writing FILL is to think about positions in terms of screen pixel coordinates, rather than turtle coordinates. The top-level procedure FILL computes the pixel coordinates corresponding to the turtle's position, and those pixel coordinates are used as inputs to the lower-level procedures which do the real work. Figure 11 shows the screen coordinate system used in FILL. The origin of this system (the point with horizontal and vertical coordinates zero) is in the top left corner of the screen. XCOR (the horizontal coordinate) gets bigger as you move to the right. YCOR (the vertical coordinate) gets bigger as you move *down* the screen; compare this with Logo's turtle-step YCOR, which gets bigger as you move up the screen.

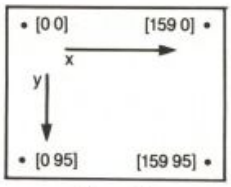


Figure 11

Because FILL uses screen coordinates instead of turtle coordinates, we can't use the usual Logo graphics procedures like FORWARD or XCOR. Instead, we have to write our own tools for examining and modifying screen pixels. Two important procedures in this project are COLOR.AT, which examines the color of a pixel, and DOT, which fills in a pixel.

One final point about the screen array is that each byte of computer memory contains the color information for four pixels. Logo's .EXAMINE procedure lets us look at an entire byte at a time, not just one pixel. Therefore, the program is more efficient if we can design it to examine four pixels at once. You'll see how we do that when we get to the description of the FILL.RAY procedure.

Initialization

Procedures FILL, FILL1, and FILL2 are invoked just once each time you use FILL. They set up certain information that is needed throughout the program. Here are the procedures, followed by a list of their important variables.

```
TO FILL
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH 0.8]
PU
FILL1 79+INT (XCOR/2) 48-(INT (YCOR*:SCRUNCH/2))
        IF (PEN="PE) [0] [PN + 1]
END

TO FILL1 :XCOR :YCOR :PEN
FILL2 COLOR.AT :XCOR :YCOR 0 0
END
```

```
TO FILL2 :BG :BGBYTE :PENBYTE
MAKE "BGBYTE 85*:BG
MAKE "PENBYTE 85*:PEN
FILL.BOTH FILL.LINE 0 0
END
```

SCRUNCH — The aspect ratio. This ratio is 0.8 unless you have changed it by using Logo's .SETSCR command. There is no direct way for FILL to find out the current aspect ratio, so it simply assumes a value of 0.8 unless you provide a different value in the global variable named SCRUNCH before you use FILL. This information is used in the procedure FILL to help convert the current turtle position into screen pixel coordinates.

XCOR — The turtle's current horizontal position, in pixels. Note that the *variable* XCOR is different from the Logo *procedure* named XCOR, which operates in turtle steps. Note also that the name XCOR is used for other variables in several subprocedures to hold local position information.

YCOR — The turtle's current vertical position, in pixels. The same notes apply as for XCOR.

PEN — The pen we should use for filling. Since one of the possibilities is to fill by erasing (setting pixels to the background color), we don't use exactly the same numbers that Logo uses for pens. Instead, Logo's pens 0 to 2 are represented in this variable with the numbers 1 to 3, while the number 0 represents the background color. We use the background color if the turtle is in penerase (PE) when you give the FILL command. Representing the background as 0 and the three pens as 1 to 3 is convenient in this program, because those numbers are the ones that are actually stored in the screen memory in the Atari computer.

BG — The pen number that is the background of the region we should fill. This is not necessarily *the* background color of the screen. When you give the FILL command, FILL1 uses subprocedure COLOR.AT to find out whether the particular pixel at the turtle's position is in the background color or in one of the three pens. Whichever is true of that pixel, the corresponding color is what we look for to determine the region we're supposed to fill. The value of BG is coded like that of PEN: 0 for background, 1 to 3 for the three pens.

BGBYTE — FILL2 sets this variable to the value of BG multiplied by 85. This has the effect of reproducing the value of BG four times in a byte.* A memory byte that contains this number represents four consecutive BG-colored pixels.

PENBYTE — This is PEN reproduced four times in a byte, and it represents four consecutive PEN-colored pixels.

*If you understand how numbers are represented in binary in the computer's memory, you'll want to know that 85 is 01010101 binary. Multiplying a two-bit code (the possible values are 0 to 3) by this number has the desired effect of reproducing it four times in the eight-bit byte. If you don't know about binary representation, don't worry about it.

There is a trick in the way FILL1 calls FILL2. FILL2 has three inputs, named BG, BGBYTE, and PENBYTE. FILL1 provides the real value for the first input (BG), but it uses zero as the values for the others.

```
FILL2 (COLOR.AT :XCOR :YCOR) 0 0
```

FILL2 starts by assigning new values to these input variables. The reason for this trick is to make BGBYTE and PENBYTE *local* variables of FILL2 instead of global variables. Using local variables avoids leaving clutter around when FILL is finished. Actually, the use of local variables isn't terribly important in this particular example, but the same trick is used in some procedures we'll see later (most notably FILL.UP1) where it really is essential.

FILL2 begins the real work of filling an area with the instruction

```
FILL.BOTH FILL.LINE 0 0
```

FILL.LINE fills horizontally, on the line where the turtle is when you give the FILL command. Then FILL.BOTH uses information output by FILL.LINE to handle the vertical part of the filling. We'll discuss these procedures in more detail in the following sections.

Filling a Line

Here is the definition of FILL.LINE.

```
TO FILL.LINE :LEFT :RIGHT
MAKE "LEFT FILL.RAY :XCOR :YCOR (-1)
MAKE "RIGHT FILL.RAY :XCOR+1 :YCOR 1
OP (SE :YCOR :LEFT :RIGHT)
END
```

This procedure uses the same trick as FILL2 to create local variables LEFT and RIGHT. Although they're defined as inputs to FILL.LINE, these variables really get their values within FILL.LINE itself.

Most Logo procedures are either *commands,* which do something visible like move a turtle, or *operations,* which have no visible effect but instead output a value, like the arithmetic operations. FILL.LINE has both an effect and an output. Its effect is to fill the line on which the turtle starts. (Turn back to figure 3 to see FILL.LINE at work.) Its output is a list of coordinates, indicating how far to the left and right it was able to fill.

The turtle starts out somewhere in the middle of the area we want to fill. To fill the line containing the turtle's position, we have to start from that position and fill both to the left and to the right. FILL.LINE invokes FILL.RAY twice, first to fill toward the left and then to fill toward the right. FILL.RAY knows which direction to use because of its third input, which is −1 to fill leftward or 1 to fill rightward.

Filling in One Direction

FILL.RAY does all of the actual filling in of dots in the entire FILL program. The other procedures simply figure out where to tell FILL.RAY to go to work.

Because of the importance of FILL.RAY, I put a lot of effort into trying to make it fast. Unfortunately, the cost of speed is complexity. Let's start by examining a version of FILL.RAY that doesn't yet have all of the efficiency features added.

```
TO FILL.RAY :XCOR :YCOR :DELTA
IF EDGE :XCOR :YCOR [OP :XCOR-:DELTA]
DOT :XCOR :YCOR
OP FILL.RAY :XCOR+:DELTA :YCOR :DELTA
END
```

FILL.RAY has three inputs. The first two are the horizontal (x) and vertical (y) screen coordinates of the pixel at which we want to start filling. The third input tells FILL.RAY the direction in which to fill.*

The strategy of FILL.RAY is this:

1. Look at a pixel to see if it's in our background color.†
2. If it's not in our background color, it is a border for the area we're filling. Output the x coordinate of the last pixel we actually filled—the one before this one.
3. If it is in our background color, fill it and move on to the next pixel in the desired direction, left or right.

To implement this strategy, FILL.RAY uses two subprocedures. The first, EDGE, is a predicate that outputs TRUE if the pixel it examines is in something *other than* the background color. The second subprocedure, DOT, fills in the pixel at the coordinates you give it as inputs. We'll look at those procedures later. For now, the important point is to understand how they're used by FILL.RAY.

Filling Vertically

We have seen how the FILL program fills one horizontal line, the one containing the turtle's position. What remains is to fill more lines, above and below that first one. This task is entrusted to FILL.BOTH.

```
TO FILL.BOTH :RANGE
FILL.UP :RANGE (-1)
FILL.UP :RANGE 1
END
```

*The word *delta* is the name of a Greek letter (Δ) that is often used in mathematics to represent a *change* in something. In this case, :DELTA is added to :XCOR each time a dot is filled in. If :DELTA is positive, the new x coordinate is to the right of the old one. If :DELTA is negative, the new coordinate is to the left.

†As explained earlier, this may or may not be *the* background color of the screen.

The name FILL.BOTH indicates that it must fill both above and below the line we've already filled. Just as FILL.LINE invokes FILL.RAY twice, FILL.BOTH invokes a subprocedure called FILL.UP twice.

FILL.BOTH, you'll remember, is invoked by FILL2. The input to FILL.BOTH is the output from FILL.LINE. This output is a list of three numbers: the vertical (y) coordinate of the line we've filled, and the leftmost and rightmost horizontal (x) coordinates of the line.* See figure 12 for a pictorial representation of this information.

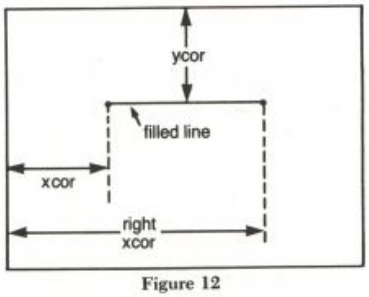


Figure 12

FILL.BOTH gives two inputs to FILL.UP. The first input is the range list. The second input tells FILL.UP the direction (up or down) in which to fill. This second input is either 1 or −1, just like the similar direction input to FILL.RAY.

Here is the definition of FILL.UP.

```
TO FILL.UP :RANGE :DELTA
FILL.UP1 (:DELTA+FIRST :RANGE)
       FIRST BF :RANGE LAST :RANGE :DELTA 0 0
END
```

All it does is to invoke FILL.UP1, with six inputs. The first three inputs are the three members of the range list, except that the vertical coordinate is offset by one. (The reason is this: the range list output by FILL.LINE contains the vertical coordinate of the line it just filled. We now want to fill a new line, just above or just below that line. The first input to FILL.UP1 is the vertical coordinate of the line we should fill next.) The fourth input to FILL.UP1 is the direction indicator, 1 or −1. The fifth and sixth inputs are given as zero. They're really used as local variables within FILL.UP1.

The Smart Procedure

FILL.UP1 really contains all the geometric knowledge of this program. FILL.UP1 has to know how to fill an area above or below a given line. This task would be very easy if areas were always pleasantly shaped. In fact, though, the filling job may have to "double back" because of irregularities in the area we're filling. This complication is illustrated in figures 4 and 5

*If you want to be picky, of course, what we've filled is a line *segment*, not a line.

(reproduced here). In figure 4, we are filling upward. This process continues straightforwardly until we get above the "hole" in the center of the region. At that point, the program is able to extend the filled area farther to the left. It then discovers a new, unfilled region below the new line. Figure 5 shows that the program has reversed its direction; it's filling downward to take care of the area to the left of the central hole.

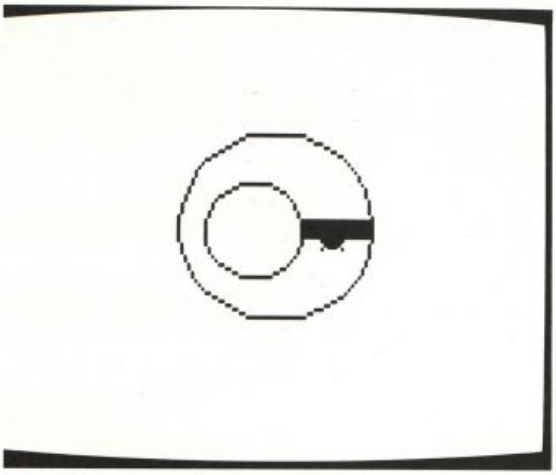


Figure 4

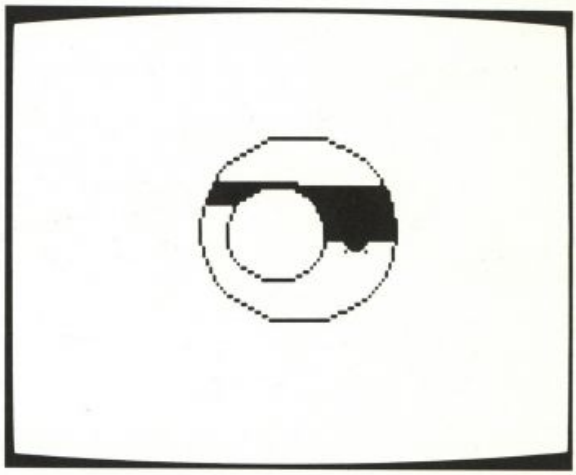


Figure 5

The strategy of FILL.UP1 is quite complicated, but it's made up of two kinds of parts: using FILL.RAY, and using FILL.UP1 recursively.

1. Use FILL.RAY to fill at the current vertical position.
2. Compare the horizontal extent of FILL.RAY's work to the horizontal extent of the previous line.
3. If we've gone farther on this line than on the previous line, invoke FILL.UP1 recursively to deal with the area newly exposed.
4. Also invoke FILL.UP1 recursively to continue with the same region we were already filling.

Since the procedure is complicated, we'll show its definition with the instruction lines numbered. In the discussion that follows we'll refer to particular lines by number.

```
[1]  TO FILL.UP1 :YCOR :LEFT :RIGHT :DELTA :NEWL :NEWR
[2]  MAKE "NEWL FILL.RAY :LEFT :YCOR (-1)
[3]  IF :NEWL<:LEFT
         [FILL.UP1 :YCOR-:DELTA :NEWL :LEFT (-:DELTA) 1 0]
[4]  MAKE "NEWR
         IF :NEWL>:RIGHT [:NEWL-1] [FILL.RAY :LEFT+1 :YCOR 1]
[5]  IF :NEWL<:NEWR+1
         [FILL.UP1 :YCOR+:DELTA :NEWL :NEWR :DELTA 2 0]
[6]  IF :NEWR>:RIGHT
         [FILL.UP1 :YCOR-:DELTA :RIGHT :NEWR (-:DELTA) 3 0]
[7]  MAKE "NEWL FIND.BG :NEWR :YCOR :RIGHT
[8]  IF WORDP :NEWL [FILL.UP1 :YCOR :NEWL :RIGHT :DELTA 4 0]
[9]  END
```

Refer to figure 13 for a picture of what happens in FILL.UP1's work. The solid horizontal line in that picture was filled earlier, either by FILL.LINE or by the previous invocation of FILL.UP1. The dashed horizontal line above is the one that will be filled by the current invocation of FILL.UP1.

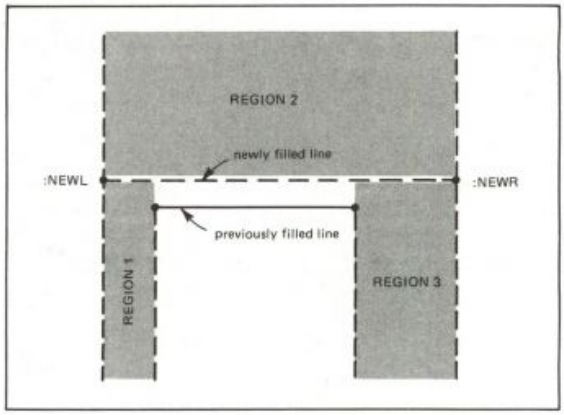


Figure 13

Here is a list of the variables used in FILL.UP1.

YCOR	The vertical coordinate of the dashed line, the one being filled by this invocation of FILL.UP1.
LEFT	The leftmost horizontal coordinate of the *solid* line, the one previously filled.
RIGHT	The rightmost horizontal coordinate of the solid, previously filled line.
DELTA	The direction indicator. Its value will be 1 if the new (dashed) line is above the old (solid) line, or −1 if the new line is below the old line.
NEWL	The leftmost horizontal coordinate of the new (dashed) line.
NEWR	The rightmost horizontal coordinate of the new line.

Each invocation of FILL.UP1 actually fills only one line. This filling is done by using FILL.RAY twice, on lines 2 and 4 of the procedure. Line 2 fills to the left of :LEFT, and line 4 fills to the right of :LEFT. The variables NEWL and NEWR are given as values the x coordinates of the endpoints of the newly filled line.

When we're filling vertically, the most obvious thing is that after filling one line, we must continue filling vertically in the same direction. Referring to figure 13, after filling the dashed line we must continue upward, filling region 2 in the figure. (Of course, we don't know yet what the exact shape of that region will be. In the figure, it's shown as extending straight up, but the edges might really be curved.) This continuation in the same vertical direction is done in line 5 of the procedure.

How do we know when to stop? The answer is that if on *this* level we didn't manage to fill anything (because we ran into borders right away), then we shouldn't continue to the next level up. That's why line 5 compares :NEWL to :NEWR. If they're equal, we didn't fill anything on this level.

There are two possible cases of "doubling back": one if the newly filled line extends farther to the left than the old line, and one if the new line extends farther to the right. In figure 13, both of these situations have arisen.

We know that the new line has extended farther to the left than the old line if :NEWL is less than :LEFT. This is the situation at the transition from figure 4 to figure 5, which we've discussed earlier. Line 3 of the procedure checks for this situation. If the condition is met, then FILL.UP1 is recursively invoked to fill what is labeled region 1 in figure 13.

Similarly, we must double back on the right (into region 3 of figure 13) if :NEWR is greater than :RIGHT. Line 6 of FILL.UP1 takes care of this case. An example of this situation is at the transition between figure 7 and figure 8 (reproduced here). In figures 6 and 7, the program was filling downward. When the lower boundary of the region is reached, in figure 7, the program doubles back and starts filling upward in figure 8.

Figure 6 Figure 7 Figure 8

By the way, the doubling back into region 1 happens *before* the continued filling of region 2. But the doubling back into region 3 happens *after* region 2 is filled. That's because lines 3, 5, and 6 happen to be in the order they are. If line 3 were moved below line 5, the program would always complete one direction of filling before starting in the other direction.

There is one more complication in FILL.UP1. The line that is filled in lines 2 and 4 of the procedure extends to both sides of :LEFT, the leftmost end of the previously filled line. Suppose that a border is reached above the old line, before its rightmost end. This situation is shown in figure 14. Since we want to fill all of the area above the previously filled line, it's not enough to fill the area above the dashed line in the figure. We must also fill what is labeled as region 4.

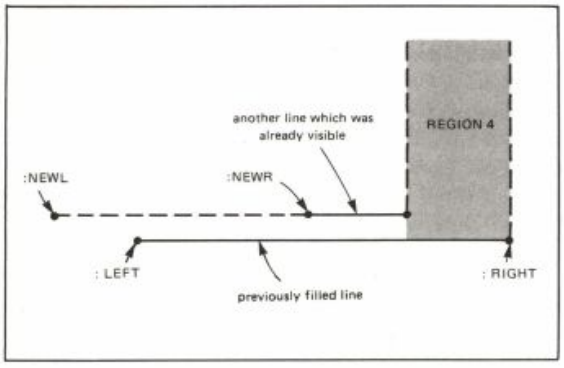


Figure 14

How do we know when this situation arises? First of all, :NEWR must be less than :RIGHT. Second, if we look to the right of :NEWR, we must find another patch of background color before reaching :RIGHT. This search is conducted by FIND.BG, which is used on line 7 of FILL.UP1. FIND.BG outputs the empty list if it does not find a suitable background pixel. If it does find one, FIND.BG outputs the x coordinate of that pixel. This coordinate is the left edge of region 4. Line 8 of FILL.UP1 checks to see if FIND.BG found a background pixel. If so, it invokes FILL.UP1 once more to fill region 4.

Examining a Screen Pixel

The real core of this program is the strategy FILL.UP1 uses to explore the nooks and crannies of irregular shapes. What remains for us to consider are the utility procedures that actually manipulate individual pixels. For example, FILL.RAY relies on EDGE to find out whether a particular pixel is a border of the area.

```
TO EDGE :XCOR :YCOR
OP NOT EQUALP :BG COLOR.AT :XCOR :YCOR
END

TO COLOR.AT :XCOR :YCOR
OP PIXEL (.EXAMINE POSADDR :XCOR :YCOR) REMAINDER :XCOR 4
END

TO POSADDR :XCOR :YCOR
OP 16384 + 40*:YCOR + INT (:XCOR/4)
        Use 8192 instead of 16384 for 16K Atari.
END

TO PIXEL :BYTE :XCOR
IF :XCOR=0 [OP INT (:BYTE/64)]
IF :XCOR=1 [OP REMAINDER INT (:BYTE/16) 4]
IF :XCOR=2 [OP REMAINDER INT (:BYTE/4) 4]
OP REMAINDER :BYTE 4
END
```

EDGE compares the color* of a particular pixel with our background color. It outputs TRUE if the two are different. That is, EDGE outputs TRUE if the pixel it's examining is on an edge of the area we're filling.

COLOR.AT outputs the color status of a pixel. Remember that each byte of screen memory contains this information for four pixels. So COLOR.AT must read a byte of screen memory and extract from that byte the particular pixel we're interested in.

POSADDR translates from the x and y coordinates of a pixel to the byte address in screen memory that contains that pixel. If you want to know about how these addresses are calculated, read the Savepict and Loadpict project.

*Actually, not the color number, but the pen number, in the form discussed earlier in the description of the PEN and BG variables.

PIXEL extracts one pixel from a byte. It takes two inputs. The first input is a byte of screen memory. The second input is a number from 0 to 3, specifying which pixel we want within that byte.

Filling One Pixel

FILL.RAY uses the procedure DOT to fill each pixel. DOT takes the coordinates of the pixel as inputs. Here it is.

```
TO DOT :XCOR :YCOR
DOTA POSADDR :XCOR :YCOR
END

TO DOTA :ADDR
RUN SE (WORD "DOT REMAINDER :XCOR 4) .EXAMINE :ADDR
END

TO DOT0 :BYTE
.DEPOSIT :ADDR SUM (REMAINDER :BYTE 64) 64*:PEN
END

TO DOT1 :BYTE
.DEPOSIT :ADDR (SUM (64*INT (:BYTE/64))
        (16*:PEN) (REMAINDER :BYTE 16))
END

TO DOT2 :BYTE
.DEPOSIT :ADDR (SUM (16*INT (:BYTE/16))
        (4*:PEN) (REMAINDER :BYTE 4))
END

TO DOT3 :BYTE
.DEPOSIT :ADDR SUM (4*INT (:BYTE/4)) :PEN
END
```

DOT must change the color of one pixel in a byte, leaving the other three pixels of that byte unchanged. Since Logo's .DEPOSIT command can only change an entire byte of memory at once, DOT has to combine the new color of one pixel with the old colors of the three other pixels. Precisely how to do this depends on which pixel in the byte we want to change, so DOT has a subprocedure for each possibility. These subprocedures are named DOT0 through DOT3.

Making FILL.RAY *More Efficient*

Earlier we looked at a simplified version of FILL.RAY, which examines and fills one pixel at a time. It's faster if we can examine an entire byte full of pixels at once. Here is the modified FILL.RAY, which does that, along with some new subprocedures.

```
TO FILL.RAY :XCOR :YCOR :DELTA
IF BYTEPOS :XCOR :DELTA
        [IF :BGBYTE=.EXAMINE POSADDR :XCOR :YCOR
            [OP FILL.CHUNK :XCOR :YCOR
                 POSADDR :XCOR :YCOR :DELTA] ]
IF EDGE :XCOR :YCOR [OP :XCOR-:DELTA]
DOT :XCOR :YCOR
OP FILL.RAY :XCOR+:DELTA :YCOR :DELTA
END

TO FILL.CHUNK :XCOR :YCOR :ADDR :DELTA
.DEPOSIT :ADDR :PENBYTE
IF :BGBYTE=.EXAMINE :ADDR+:DELTA
        [OP FILL.CHUNK :XCOR+4*:DELTA :YCOR
             :ADDR+:DELTA :DELTA]
OP FILL.RAY :XCOR+4*:DELTA :YCOR :DELTA
END

TO BYTEPOS :XCOR :DELTA
IF :DELTA>0 [OP 0=REMAINDER :XCOR 4]
OP 3=REMAINDER :XCOR 4
END
```

FILL.RAY can only examine a complete byte of four pixels if the pixel it's ready to examine next is the first one in a byte. The predicate BYTEPOS outputs TRUE if that is the case. If not, FILL.RAY does the same things it did in the simpler version.

If BYTEPOS is TRUE, FILL.RAY examines the entire byte containing the pixel of interest. If that byte contains four pixels all in background color, we can fill all four at once. The variable BGBYTE contains the byte value that represents four background pixels.

If FILL.RAY does find a byte full of background pixels, it uses FILL.CHUNK to fill all four at once. FILL.CHUNK then examines the next byte to see if it, too, contains four background pixels. Once FILL.CHUNK reaches a byte that is not entirely background, it reverts to the use of FILL.RAY to check individual pixels.

Finding Region 4

The procedure FIND.BG, which is used to detect the appearance of a fourth region to fill, is very much like FILL.RAY, with two exceptions. First, FIND.BG passes over nonbackground pixels and stops when it reaches a background pixel. Second, FIND.BG just examines the pixels, whereas FILL.RAY fills them also.

```
TO FIND.BG :XCOR :YCOR :LIMIT
IF :XCOR>:LIMIT [OP []]
IF BYTEPOS :XCOR 1
        [IF :PENBYTE=.EXAMINE POSADDR :XCOR :YCOR
             [OP FIND.BG :XCOR+4 :YCOR :LIMIT] ]
IF NOT EDGE :XCOR :YCOR [OP :XCOR]
OP FIND.BG :XCOR+1 :YCOR :LIMIT
END
```

PROGRAM LISTING

```
TO FILL
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH ►
     0.8]
PU
FILL1 79+INT (XCOR/2) 48-(INT ►
     (YCOR*:SCRUNCH/2)) IF (PEN="PE) ►
     [0] [PN + 1]
END

TO FILL1 :XCOR :YCOR :PEN
FILL2 COLOR.AT :XCOR :YCOR 0 0
END

TO FILL2 :BG :BGBYTE :PENBYTE
MAKE "BGBYTE 85*:BG
MAKE "PENBYTE 85*:PEN
FILL.BOTH FILL.LINE 0 0
END

TO FILL.LINE :LEFT :RIGHT
MAKE "LEFT FILL.RAY :XCOR :YCOR (-1)
MAKE "RIGHT FILL.RAY :XCOR+1 :YCOR 1
OP (SE :YCOR :LEFT :RIGHT)
END

TO FILL.BOTH :RANGE
FILL.UP :RANGE (-1)
FILL.UP :RANGE 1
END

TO FILL.UP :RANGE :DELTA
FILL.UP1 (:DELTA+FIRST :RANGE) FIRST ►
     BF :RANGE LAST :RANGE :DELTA 0 0
END

TO FILL.UP1 :YCOR :LEFT :RIGHT :DELTA ►
     :NEWL :NEWR
MAKE "NEWL FILL.RAY :LEFT :YCOR (-1)
IF :NEWL<:LEFT [FILL.UP1 :YCOR-:DELTA ►
     :NEWL :LEFT (-:DELTA) 1 0]
MAKE "NEWR IF :NEWL>:RIGHT [:NEWL-1] ►
     [FILL.RAY :LEFT+1 :YCOR 1]
IF :NEWL<:NEWR+1 [FILL.UP1 ►
     :YCOR+:DELTA :NEWL :NEWR :DELTA 2 ►
     0]
IF :NEWR>:RIGHT [FILL.UP1 :YCOR-:DELTA ►
     :RIGHT :NEWR (-:DELTA) 3 0]
MAKE "NEWL FIND.BG :NEWR :YCOR :RIGHT
IF WORDP :NEWL [FILL.UP1 :YCOR :NEWL ►
     :RIGHT :DELTA 4 0]
END

TO EDGE :XCOR :YCOR
OP NOT EQUALP :BG COLOR.AT :XCOR :YCOR
END

TO COLOR.AT :XCOR :YCOR
OP PIXEL (.EXAMINE POSADDR :XCOR ►
     :YCOR) REMAINDER :XCOR 4
END

TO POSADDR :XCOR :YCOR
OP 16384 + 40*:YCOR + INT (:XCOR/4)
END

TO PIXEL :BYTE :XCOR
IF :XCOR=0 [OP INT (:BYTE/64)]
IF :XCOR=1 [OP REMAINDER INT ►
     (:BYTE/16) 4]
IF :XCOR=2 [OP REMAINDER INT (:BYTE/4) ►
     4]
OP REMAINDER :BYTE 4
END

TO DOT :XCOR :YCOR
DOTA POSADDR :XCOR :YCOR
END

TO DOTA :ADDR
RUN SE (WORD "DOT REMAINDER :XCOR 4) ►
     .EXAMINE :ADDR
END

TO DOT0 :BYTE
.DEPOSIT :ADDR SUM (REMAINDER :BYTE ►
     64) 64*:PEN
END

TO DOT1 :BYTE
.DEPOSIT :ADDR (SUM (64*INT ►
     (:BYTE/64)) (16*:PEN) (REMAINDER ►
     :BYTE 16))
END

TO DOT2 :BYTE
.DEPOSIT :ADDR (SUM (16*INT ►
     (:BYTE/16)) (4*:PEN) (REMAINDER ►
     :BYTE 4))
END

TO DOT3 :BYTE
.DEPOSIT :ADDR SUM (4*INT (:BYTE/4)) ►
     :PEN
END
```

```
TO FILL.RAY :XCOR :YCOR :DELTA
IF BYTEPOS :XCOR :DELTA [IF ▸
     :BGBYTE=.EXAMINE POSADDR :XCOR ▸
     :YCOR [OP FILL.CHUNK :XCOR :YCOR ▸
     POSADDR :XCOR :YCOR :DELTA] ]
IF EDGE :XCOR :YCOR [OP :XCOR-:DELTA]
DOT :XCOR :YCOR
OP FILL.RAY :XCOR+:DELTA :YCOR :DELTA
END

TO FILL.CHUNK :XCOR :YCOR :ADDR :DELTA
.DEPOSIT :ADDR :PENBYTE
IF :BGBYTE=.EXAMINE :ADDR+:DELTA [OP ▸
     FILL.CHUNK :XCOR+4*:DELTA :YCOR ▸
     :ADDR+:DELTA :DELTA]
OP FILL.RAY :XCOR+4*:DELTA :YCOR ▸
     :DELTA
END

TO BYTEPOS :XCOR :DELTA
IF :DELTA>0 [OP 0=REMAINDER :XCOR 4]
OP 3=REMAINDER :XCOR 4
END

TO FIND.BG :XCOR :YCOR :LIMIT
IF :XCOR>:LIMIT [OP []]
IF BYTEPOS :XCOR 1 [IF ▸
     :PENBYTE=.EXAMINE POSADDR :XCOR ▸
     :YCOR [OP FIND.BG :XCOR+4 :YCOR ▸
     :LIMIT] ]
IF NOT EDGE :XCOR :YCOR [OP :XCOR]
OP FIND.BG :XCOR+1 :YCOR :LIMIT
END
```

Savepict and Loadpict

When you've drawn a complicated picture, it's useful to be able to save the picture itself in a disk file, so that you can later restore it to the screen without going through the procedures that drew the picture again. For example, suppose you're writing a video adventure game in which characters in the story are drawn against a backdrop showing a forest, dungeon, or whatever. The backdrop could be saved as a picture file and then loaded onto the screen for each scene before drawing in the actors.

In this project, you'll see three different sets of Logo programs for saving and loading pictures. The three versions differ in how fast they can load a picture and also differ somewhat in flexibility. The last version, for example, allows a small picture to be "stamped" on the screen in different positions. One thing to learn from this project is how using different *data representations* can affect the efficiency of a program.

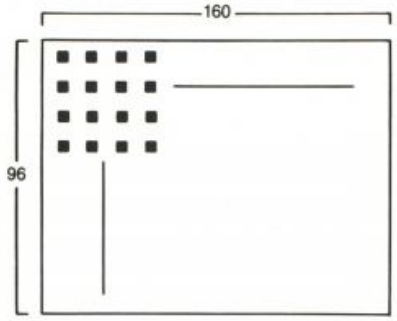
There are two ways to approach this project. If you just want to use these procedures as a tool to save and load pictures for some other project of your own, you don't have to understand some of the details explained here about how pictures are stored. On the other hand, by studying how the project works, you can learn about the important idea of data representation.

Note: If you have a 16K Atari computer, you should use the number `8192` instead of `16384` in procedures `SAVEPICT`, `LOADPICT`, and `PICTLOC`. (`PICTLOC` appears only in the third version of the project.) With a 16K machine, you don't have a disk drive, but you could save pictures on cassette.

By Brian Harvey.

How a Picture Is Stored

In order to save and load pictures, we have to know something about how a picture is represented in the Atari computer. In this project we are concerned only with the pictures drawn with pens, not with the turtle shapes. The lines you draw are represented as a pattern of dots (called *pixels*) on the screen. There are 96 rows and 160 columns of dots on the screen:



Screen pixels

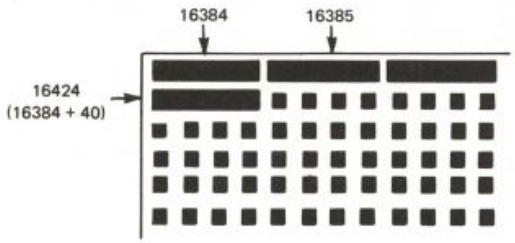
The reason that a diagonal line comes out jagged on the screen is that it isn't actually drawn as a smooth line, but simply by filling in certain dots on the screen. Each pixel can be in one of four conditions: it can be empty (that is, it can be in the background color) or it can be filled in with one of the three possible pens.

By the way, the length of a "turtle step" is not the same as the distance between pixels. That is, when you type the command `FORWARD 100` the turtle does not move 100 pixels on the screen. How many pixels it actually does move depends on the direction. If you're moving horizontally (heading 90, for example), then `FORWARD 100` moves through 50 pixels. If you're moving vertically, the distance depends on the *aspect ratio,* which is controlled by the `.SETSCR` command. The usual aspect ratio is 0.8, in which case `FORWARD 100` moves 40 pixels (50 times 0.8). In this project, since we're interested in saving a picture that is already on the screen rather than drawing a picture with turtle commands, we have to think in terms of pixels, not in terms of turtle steps.

I said that each pixel can be in any of four conditions (background or three pens). Therefore, each pixel can be represented in the computer's memory using two *bits,* or binary digits. Each bit can be either zero or one. The four conditions are represented this way:

0 0 background
0 1 pen 0
1 0 pen 1
1 1 pen 2

Memory is grouped into *bytes* of eight bits. So each byte represents four pixels. There are 96 times 160, or 15,360, pixels altogether on the screen. The memory required is one fourth of that, or 3840 bytes. It happens that the first byte of Logo's screen memory is at memory location number 16384. So the picture memory is arranged something like this:



Picture memory

Characters (letters, digits, spaces, and so on) are represented in the computer's memory by a number that is stored in one byte. For example, the letter *A* is represented by a byte containing the number 65. Most of the time you don't have to worry about this, but if you remember this fact, it'll help you understand the process of storing information in disk files.

Representing the Screen in a Disk File

The most straightforward way to represent a screen picture in a disk file is simply to write each of the 3840 bytes into the file. To find out what is in each byte, we use the .EXAMINE operation, which outputs a number representing the byte at whatever memory location is used as its input. For example:

```
PRINT .EXAMINE 16384
```

will print the number in the first byte of Logo's screen memory. This byte represents the first four pixels in the upper left corner of the screen. (For Atari computers with 16K of RAM, the first byte of screen memory is in location 8192 instead of 16384.)

It would be possible to save a picture in a file, then, with a program like this:

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM
IF :NUM=0 [STOP]
PRINT .EXAMINE :LOC
SAVEPICT1 :LOC+1 :NUM-1
END
```

Each byte of the picture memory would be represented in the file by a line containing the digits in the number in that byte. That is, if a particular byte happened to contain the number 125, that byte would be stored in the file as the three digits 1, 2, 5, just as it is typed on the screen by a PRINT command. Each digit takes up one byte in the file. Therefore, using this scheme, it takes three bytes in the file to represent one byte in the picture!

(Actually, another byte is used to represent the end-of-line code.) This leads to very large files.

Instead, it would be better to use only one byte in the file to represent each byte in the picture. This can be done by using the operation CHAR. This procedure takes a number as its input and outputs the single character that corresponds to that number. For example, CHAR 65 outputs the letter A. Using this procedure, we can write the program as follows:

Savepict/Loadpict, Version 1

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM
IF :NUM=0 [STOP]
TYPE CHAR .EXAMINE :LOC
SAVEPICT1 :LOC+1 :NUM-1
END

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 3840
SETREAD []
END

TO LOADPICT1 :LOC :NUM
IF :NUM=0 [STOP]
.DEPOSIT :LOC ASCII RC
LOADPICT1 :LOC+1 :NUM-1
END
```

To use the SAVEPICT procedure, you first draw a picture on the screen using the usual turtle commands. Then you say

```
SAVEPICT "D:PICTFILE
```

or whatever you want to name the file. The program writes 3840 bytes into this file. Later, you can restore the picture to the screen by typing

```
LOADPICT "D:PICTFILE
```

The operation ASCII, which is used in LOADPICT1, is the inverse of CHAR. It takes a single character as input and outputs the number that represents that character. So ASCII "A outputs 65.

Experiment with these procedures. You'll find that both saving and loading pictures are quite slow. This is because the procedures SAVEPICT1 and LOADPICT1 are invoked 3840 times, once for each byte of screen memory, even if nothing is drawn in that part of the screen. Also, the files written by this version of SAVEPICT are rather large (3840 bytes), so you can't fit very many on a diskette.

Sparse Data Representations

A typical turtle graphics picture is *sparse.* This means that most of the pixels on the screen are unused (background color), which means that most of the bytes of picture memory are zero. It seems silly to write a file that is mostly full of zeros. By using a cleverer representation of the picture, we can write smaller files and make the loading of a picture file much faster.

The idea is this: as we look through the picture memory, we'll find a bunch of zero bytes, and then a nonzero one, and then a bunch more zero bytes, and so on. To make this more specific, consider this sample fragment of a picture memory:

```
0 0 0 0 0 0 0 0 0 23 0 0 0 0 47 0 0 0 0 0 0 0 0 15
```

In the first version of the program, we'd represent these twenty-four bytes of screen memory as twenty-four bytes in the file. But instead, we can think of this as 9 zeros, 23, 4 zeros, 47, 8 zeros, 15. We could store this information in a file in this form:

```
9 23 4 47 8 15
```

In other words, we have decided that odd-numbered bytes in the file represent how many consecutive zero bytes are in the picture, while even-numbered bytes represent actual picture data. By representing the picture in this way, we've reduced twenty-four bytes of picture to six bytes in the file. We'll find that it is also much faster to load a picture stored in this form.

In practice, there may be several hundred consecutive zero bytes in a picture. This poses a slight problem: the largest number that can be represented in a single byte is 255. Therefore, if there are more than that many consecutive zeros, the new `SAVEPICT` procedure writes the sequence `255 0` in the file for each group of 256 zeros.

A second minor detail is that there must be a way for `LOADPICT` to know when the end of the file has been reached. This isn't a problem in the first version of the program because there all picture files are the same length, 3840 bytes. But in the new version, the length of the file depends on the number of pixels that are drawn in a nonbackground color. To solve this problem, `SAVEPICT` writes the sequence `0 0` at the end of the file. This sequence can't be part of real picture data.

Savepict/Loadpict, Version 2

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840 0
REPEAT 2 [TYPE CHAR 0]
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM :NULL
IF :NUM=0 [STOP]
SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 .EXAMINE :LOC :NULL
END
```

```
TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 ASCII RC ASCII RC
SETREAD []
END

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
.DEPOSIT :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII RC
END
```

Experiment with this version of the program. You'll notice that SAVEPICT isn't any faster, but LOADPICT is usually very much faster. The reason is that SAVEPICT must still examine every byte of picture memory, because it doesn't know ahead of time where you've drawn lines. But LOADPICT only has to deposit information into the bytes in picture memory that actually correspond to lines in the saved picture file.

Snapshots

In the second version, LOADPICT doesn't change the parts of picture memory that aren't used in the picture file you're loading. This suggests that it should be possible to *merge* two pictures. (In the first version, loading a picture file completely replaced whatever might have been on the screen before you invoked LOADPICT.) Try drawing a picture, saving it with SAVEPICT, clearing the screen, drawing another picture, and then using LOADPICT to restore the first picture. Make sure that the two pictures aren't in exactly the same part of the screen, so you can see whether the old picture remains intact.

What you'll find is that this merging of two pictures works pretty well, but not perfectly. The problem comes up if the two pictures use pixels that are right next to each other, so that a pixel in one picture is part of the same byte of memory as a pixel of the other picture. (Remember that each byte contains four pixels.) Loading a new number into that byte eliminates the pixel that used to be there. Still, this technique works perfectly if the two pictures are widely separated, and it works pretty well in most cases.

It would be handy to take advantage of this merging capability by using a picture file as a kind of rubber stamp that could be drawn in different positions on the screen. The scheme is this: you draw a small picture near the center of the screen. Then you use a version of SAVEPICT to make a "snapshot" of this picture. You can then use a version of LOADPICT to "stamp" the saved picture anywhere on the screen, depending on the turtle position.

To make this work, the picture file must include information about where the turtle was when the picture was taken. SAVEPICT must be modified to write this information in the file. Then LOADPICT must be modified to compare the current position of the turtle to the one stored in the file. If the two positions are different, the picture should be loaded into a different part of the screen memory.

This third version of the program is quite a bit more complicated than the others. The main reason for this is that it has to deal with the difference between pixels and turtle steps. To know where to "stamp" the saved picture in memory, we have to think in terms of pixels. But Logo tells us the turtle's position in turtle steps. This position has to be rounded off to the nearest pixel. Also, as explained earlier, the conversion between steps and pixels depends on the aspect ratio. There is no easy way for a Logo procedure to find out what this ratio is. The solution used in this program is that it looks for a variable named SCRUNCH in the workspace. If there is such a variable, its value should be the aspect ratio. If not, the standard value of 0.8 is assumed.

Another complication is that if the picture is being loaded into a position that is different from where it came from, part of the picture may extend beyond the edge of the screen. The procedure PUTBYTE in the following program is used like .DEPOSIT, but it checks to be sure that you are trying to deposit into the part of memory that contains the picture.

Savepict/Loadpict, Version 3

```
TO SAVEPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH 0.8]
SETWRITE :FILE
TYPE CHAR (XCOR+160)/2
TYPE CHAR (120-YCOR)*:SCRUNCH/2
SAVEPICT1 16384 3840 0
REPEAT 2 [TYPE CHAR 0]
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM :NULL
IF :NUM=0 [STOP]
SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 .EXAMINE :LOC :NULL
END

TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

TO LOADPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH 0.8]
SETREAD :FILE
LOADPICT1 PICTLOC ASCII RC ASCII RC
SETREAD []
END
```

```
TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
PUTBYTE :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII RC
END

TO PICTLOC
OP 16384+((XDIFF ASCII RC)+160*YDIFF ASCII RC)/4
END

TO PUTBYTE :LOC :BYTE
IF (AND :LOC>16383 :LOC<20224 :BYTE>0) [.DEPOSIT :LOC :BYTE]
END
```

Note: If you have a 16K Atari computer, you should use the following:

```
IF (AND :LOC>8191 :LOC<12032 :BTYE>0) [.DEPOSIT :LOC :BYTE]

TO XDIFF :XLOC
OP INT (XCOR+160)/2-:XLOC
END

TO YDIFF :YLOC
OP INT (120-YCOR)*:SCRUNCH/2-:YLOC
END
```

To experiment with this program, try something like this:

```
CS
REPEAT 4 [FD 40 RT 90]
SAVEPICT "D:SQSNAP
PU
SETPOS [80 60]
LOADPICT "D:SQSNAP
SETPOS [-70 20]
LOADPICT "D:SQSNAP
```

In practice, you wouldn't bother making a snapshot of something as simple as a square, because it's easier to draw another square than to load it from a disk file. But if you draw more complicated pictures, in multiple colors, this technique can really be worthwhile.

Suggestion: Run-Length Encoding

What if you filled in the screen completely with some pen color and tried to save that in a picture file? Using the first version of the program, of course, it doesn't matter what's on the screen; the file ends up with 3840 data bytes. But with the two later versions, something else happens. The picture memory is completely filled with bytes that represent the same number, but not zero. For example, if you fill the screen with pen 0, the picture memory will be

```
85 85 85 85 85 ...
```

In the sparse encoding scheme we've been using, this is thought of this way: 0 zero bytes, 85, 0 zero bytes, 85, and so on. What ends up in the picture file is

```
0 85 0 85 0 85 0 85 0 85 ...
```

The picture file is twice as big as the screen memory! This isn't a very good result. The smart LOADPICT will be slower for this picture than the stupid one. A sparse representation only works well if the picture is, in fact, sparse.

This is an extreme, unlikely example. But it isn't unlikely for *part* of the screen to be filled in solidly. For example, if you're drawing a picture of a farm, the background might be blue to represent the sky, and there might be a large solid green area at the bottom of the screen to represent grass.

Still, although that green area isn't *empty,* it is *uniform.* The bytes representing that area in screen memory are mostly all the same, even if not all zero. We could use a slightly more complicated data representation called *run-length encoding,* which would handle this case well. Here's how it works. Instead of a two-byte sequence representing the number of zero bytes and then the value of a data byte, we can use a sequence representing the value of a data byte and the number of consecutive bytes containing that value. For example, suppose the screen memory looks like this:

```
85 85 85 85 85 1 0 0 0 0 0 0 0 43 85 85 85 85 ...
```

We would represent that in the picture file this way:

```
85 5 1 1 0 7 43 1 85 4 ...
```

In this example, the version in the file is only a little smaller than the screen memory. But in real situations, the run lengths would often be several hundred bytes, not just five or seven.

This run-length technique is often used in serious computer graphics work. It's especially efficient for black-and-white pictures, because there are only two possible values for the data. You can just alternate them and leave them out of the file. You only store the run lengths. That is, the odd-numbered bytes of the file would contain the numbers of consecutive black pixels and the even-numbered bytes would contain the numbers of consecutive white pixels.

On the other hand, for a color picture that really is sparse, the representation we've been using is somewhat more efficient than the run-length representation. The moral is that before you choose a data representation for any problem, you should think hard about different possibilities!

PROGRAM LISTING

VERSION 1

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM
IF :NUM=0 [STOP]
TYPE CHAR .EXAMINE :LOC
SAVEPICT1 :LOC+1 :NUM-1
END
```

```
TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 3840
SETREAD []
END

TO LOADPICT1 :LOC :NUM
IF :NUM=0 [STOP]
.DEPOSIT :LOC ASCII RC
LOADPICT1 :LOC+1 :NUM-1
END
```

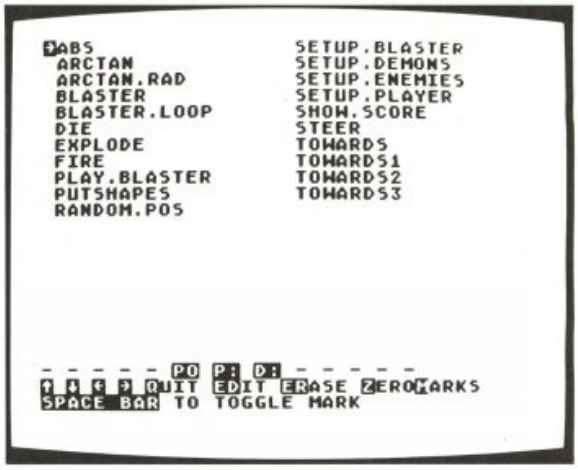
VERSION 2

```
TO SAVEPICT :FILE
SETWRITE :FILE
SAVEPICT1 16384 3840 0
REPEAT 2 [TYPE CHAR 0]
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM :NULL
IF :NUM=0 [STOP]
SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 ▸
     .EXAMINE :LOC :NULL
END

TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

TO LOADPICT :FILE
SETREAD :FILE
LOADPICT1 16384 ASCII RC ASCII RC
SETREAD []
END

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
.DEPOSIT :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII ▸
     RC
END
```

VERSION 3

```
TO SAVEPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH ▸
     0.8]
SETWRITE :FILE
TYPE CHAR (XCOR+160)/2
TYPE CHAR (120-YCOR)*:SCRUNCH/2
SAVEPICT1 16384 3840 0
REPEAT 2 [TYPE CHAR 0]
SETWRITE []
END

TO SAVEPICT1 :LOC :NUM :NULL
IF :NUM=0 [STOP]
SAVEPICT1 :LOC+1 :NUM-1 SAVEPICT2 ▸
     .EXAMINE :LOC :NULL
END

TO SAVEPICT2 :BYTE :NULL
IF AND :BYTE=0 :NULL<255 [OP :NULL+1]
TYPE CHAR :NULL
TYPE CHAR :BYTE
OP 0
END

TO LOADPICT :FILE
IF NOT NAMEP "SCRUNCH [MAKE "SCRUNCH ▸
     0.8]
SETREAD :FILE
LOADPICT1 PICTLOC ASCII RC ASCII RC
SETREAD []
END

TO LOADPICT1 :LOC :NULL :BYTE
IF AND :BYTE=0 :NULL=0 [STOP]
PUTBYTE :LOC+:NULL :BYTE
LOADPICT1 :LOC+:NULL+1 ASCII RC ASCII ▸
     RC
END

TO PICTLOC
OP 16384+((XDIFF ASCII RC)+160*YDIFF ▸
     ASCII RC)/4
END

TO PUTBYTE :LOC :BYTE
IF (AND :LOC>16383 :LOC<20224 :BYTE>0) ▸
     [.DEPOSIT :LOC :BYTE]
END

TO XDIFF :XLOC
OP INT (XCOR+160)/2-:XLOC
END

TO YDIFF :YLOC
OP INT (120-YCOR)*:SCRUNCH/2-:YLOC
END
```

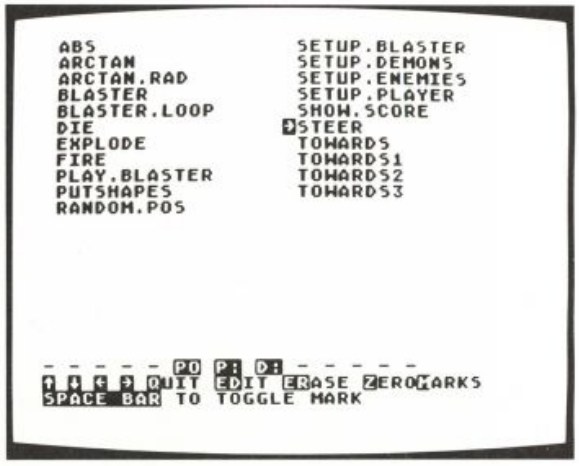
Display Workspace Manager



The Display Workspace Manager (DWM) is a tool that helps you manage projects that involve large numbers of procedures. The program lists all your procedures on the screen. You can move a pointer around, marking particular procedures. Then you can edit, erase, print, or save the marked procedures.

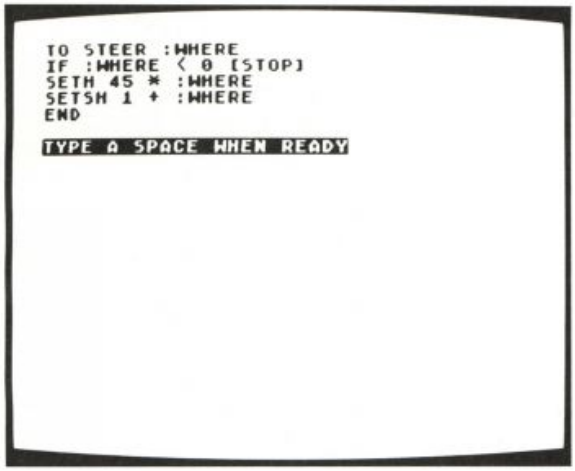
DWM divides the screen into two parts. The top part is used to list the names of procedures. The bottom few lines remind you of the commands you can type to DWM. (For example, you can type ER to *erase* procedures.)

In the figure above, DWM is being used to examine Blaster, a project in this book. The arrow points to the word STEER on the screen. STEER is the name of one of the procedures in Blaster. The pointer arrow can be moved from one procedure name to another by using the arrow keys on the Atari keyboard.

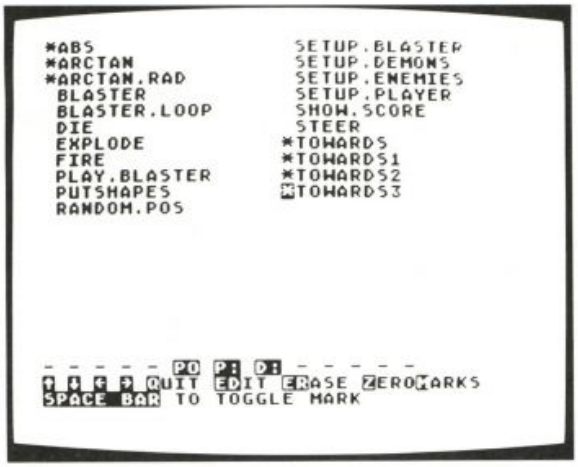


By Brian Harvey.

In the next figure, the user has typed the PO command to DWM. DWM's PO command tells it to print out the definition of the procedure at which the arrow points, in this case STEER.

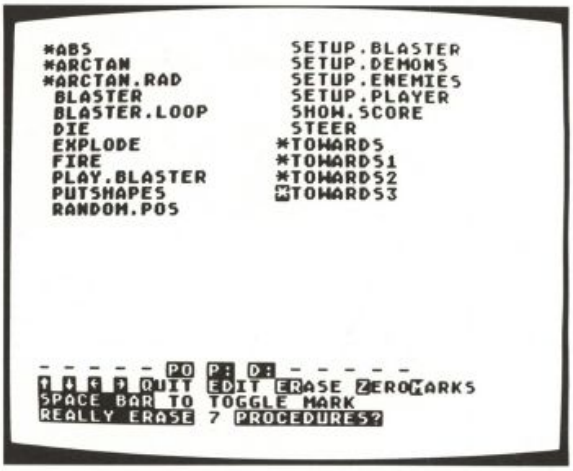


In the following figure, seven procedures have been *marked* with asterisks on the screen.

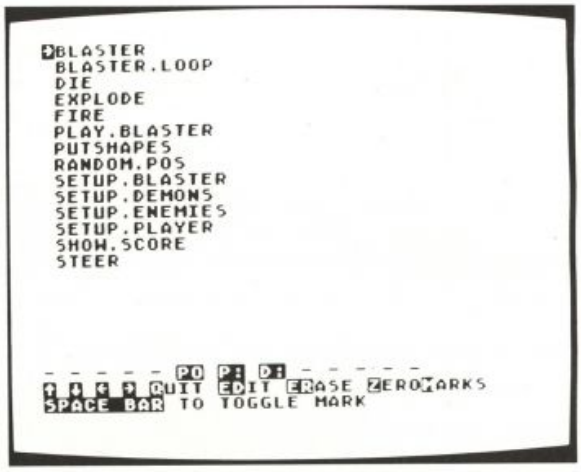


In the next figure the user has typed the command ER, which means to erase all the marked procedures. DWM has printed "Really erase 7 proce-

dures?" on the bottom line of the screen. It asks this question to make it harder for someone to erase many procedures accidentally.



In the next figure, the user has typed Y for yes, and DWM has erased the marked procedures. It now displays a shorter list of the remaining procedures.



This is a large project. I won't attempt a complete explanation of every detail of the program. Instead, I'll indicate the most important parts to understand.

Creating the List of Procedures

In order for DWM to work, it must have a list of the names of all the procedures in your project. This list must be in a global variable named

ALL.PROCEDURES. If this list doesn't already exist, the first thing DWM does is to call DWM.PROCLIST to create the list. This automatic creation of the list requires a disk drive with a writeable disk in it! DWM.PROCLIST works by doing a POTS command while writing to the disk, then rereading the results to find the names of your procedures. When the list is created automatically, it is sorted alphabetically. The sorting process is quite slow, because it's done simply rather than cleverly. (Read the Mergesort project for another sorting technique.) The automatically generated list omits all procedures whose names start "DWM" so that the procedures in the DWM program itself won't clutter up your list.

If you want to save time when starting up DWM, or if you want the procedures in your project listed in some order other than alphabetical, you can create the variable ALL.PROCEDURES yourself and make it part of the workspace file.

How DWM *Arranges the Display*

Once the list of procedures exists, DWM lists them on the screen. This is done by two main procedures, DWM.SIZE.MENU and DWM.DRAW.MENU. The first of these figures out how the names should be arranged on the screen, given the number of procedures you have in your list. The more procedures, the more columns on the screen will be required to list them all. The more columns, the less wide each column can be. This limits the length of a procedure name that can be displayed. Therefore, the program uses the smallest number of columns that will fit your list. Then the DWM.DRAW.MENU procedure uses this information to draw the display.

If the name of a procedure is too long to fit in a screen column, an inverse video plus sign (+) is shown at the end of the truncated name.

Reading DWM *Commands*

The procedure DWM.MAIN.LOOP reads and processes the commands you type to the program. Commands are either one or two characters long. Here is a list of the commands.

arrows	Move the pointer up, down, left, or right. You can type the arrow keys either with or without the CTRL key held down.
space bar	*Mark* the procedure where the pointer is, if it's not marked already, or unmark it if it is. An asterisk is displayed next to the name of marked procedures.
ED	Edit the marked procedures in the Logo editor.
ER	Erase all the marked procedures. This command first tells you how many procedures are marked and insists that you type Y to confirm that you really want to erase the procedures.
PO	Print out on the screen the single procedure whose name is pointed to by the arrow.
P:	List all marked procedures on the printer.
D:	Save all marked procedures on the disk. This command prompts for a filename to be used for the saved procedures. Notice that these save files do not contain the values of varia-

bles! But the procedures in them can be loaded with the LOAD command.

ZM — Zero Marks. Unmark all procedures.

Q — Quit. Exits from DWM.

If there are no marked procedures, the commands that normally apply to marked procedures apply instead to all procedures in the display. Be careful about erasing!

Possible Extensions

DWM takes up just under 2000 nodes, somewhat more than half the available space. This limits the size of the programs you can use it on. (This is particularly unfortunate since it's the big projects that most need this sort of help.)

If there were space, this project could be the basis for implementing workspace management tools like PACKAGE and BURY, which are found in some other versions of Logo. The technique would be to have several lists of procedures instead of just one ALL.PROCEDURES list.

PROGRAM LISTING

In the program listing that follows, characters that are underlined represent inverse-video characters on the Atari.

```
TO DWM
DWM.1 [] [] [] [] [] []
END

TO DWM.1 :PROCS :COLUMNS :CHARS :ROWS :TABS :MARKED
IF NOT NAMEP "ALL.PROCEDURES [DWM.PROCLIST]
DWM.SIZE.MENU
DWM.DRAW.MENU
SETCURSOR [0 0]
DWM.SHOW.CURSOR 1
DWM.MAIN.LOOP RC 0 1 1
END
```

CREATING THE ALL.PROCEDURES ***LIST***

```
TO DWM.PROCLIST
PR [ONE MOMENT, I'M LISTING...]
SETWRITE "D:DWM.TMP
POTS
SETWRITE []
SETREAD "D:DWM.TMP
PR [HANG ON A BIT LONGER...]
MAKE "ALL.PROCEDURES []
DWM.READ.TITLE RL
SETREAD []
ERF "D:DWM.TMP
END
```

```
TO DWM.READ.TITLE :LINE
IF EMPTYP :LINE [STOP]
DWM.READ.TITLE1 FIRST BF :LINE
DWM.READ.TITLE RL
END

TO DWM.READ.TITLE1 :NAME
IF EQUALP "DWM DWM.FIRSTPART :NAME 3 [STOP]
MAKE "ALL.PROCEDURES DWM.INSERT :NAME :ALL.PROCEDURES
END

TO DWM.INSERT :WORD :LIST
IF EMPTYP :LIST [OP FPUT :WORD []]
IF DWM.BEFORE :WORD FIRST :LIST [OP FPUT :WORD :LIST]
OP FPUT FIRST :LIST DWM.INSERT :WORD BF :LIST
END

TO DWM.BEFORE :NEW :OLD
IF EMPTYP :NEW [OP "TRUE]
IF EMPTYP :OLD [OP "FALSE]
IF (ASCII FIRST :NEW) < (ASCII FIRST :OLD) [OP "TRUE]
IF (ASCII FIRST :NEW) > (ASCII FIRST :OLD) [OP "FALSE]
OP DWM.BEFORE BF :NEW BF :OLD
END

TO DWM.FIRSTPART :WORD :NUM
IF EMPTYP :WORD [OP "]
IF EQUALP :NUM 1 [OP FIRST :WORD]
OP WORD FIRST :WORD DWM.FIRSTPART BF :WORD :NUM-1
END
```

PRINTING THE MENU

```
TO DWM.SIZE.MENU
MAKE "PROCS COUNT :ALL.PROCEDURES
MAKE "COLUMNS 1+INT ((:PROCS-1)/20)
MAKE "CHARS (INT 37/:COLUMNS)-2
MAKE "ROWS 1+INT ((:PROCS-1)/:COLUMNS)
MAKE "TABS DWM.SIZE.TABS 1 :CHARS+2 :COLUMNS
END

TO DWM.SIZE.TABS :COL :CHARS :COLS
IF :COLS = 0 [OP [99]]
OP FPUT :COL DWM.SIZE.TABS :COL+:CHARS :CHARS :COLS-1
END

TO DWM.DRAW.MENU
TS CT
DWM.DRAW.M1 :ALL.PROCEDURES 0 1 BF :TABS 1
SETCURSOR [0 20]
PR [- - - - - PO P: D: - - - - -]
(PR CHAR 156 CHAR 157 CHAR 158 CHAR 159    ▸
    [QUIT EDIT ERASE ZEROMARKS])
PR [SPACE BAR TO TOGGLE MARK]
END
```

```
TO DWM.DRAW.M1 :PROCS :ROW :COL :TABS :INDEX
IF EMPTYP :PROCS [STOP]
IF MEMBERP :INDEX :MARKED [DWM.STAR]
SETCURSOR LIST :COL :ROW
TYPE DWM.SHORT FIRST :PROCS :CHARS
IF EQUALP :ROW :ROWS-1    ▶
    [DWM.DRAW.M1 BF :PROCS    ▶
                Ø FIRST :TABS BF :TABS :INDEX+1]    ▶
    [DWM.DRAW.M1 BF :PROCS :ROW+1 :COL :TABS :INDEX+1]
END

TO DWM.STAR
SETCURSOR LIST :COL-1 :ROW
TYPE "•
END

TO DWM.SHORT :NAME :CHARS
IF (COUNT :NAME)<(:CHARS+1) [OP :NAME]
OP DWM.SHORT1 :NAME :CHARS
END

TO DWM.SHORT1 :NAME :CHARS
IF :CHARS=1 [OP CHAR 171]
OP WORD FIRST :NAME DWM.SHORT BF :NAME :CHARS-1
END
```

READING COMMANDS FROM THE KEYBOARD

```
TO DWM.MAIN.LOOP :CMD :ROW :COL :INDEX
DWM.SET.CURSOR
IF :CMD = "- [DWM.UP]
IF :CMD = CHAR 28 [DWM.UP]
IF :CMD = "= [DWM.DOWN]
IF :CMD = CHAR 29 [DWM.DOWN]
IF :CMD = "+ [DWM.LEFT]
IF :CMD = CHAR 30 [DWM.LEFT]
IF :CMD = "* [DWM.RIGHT]
IF :CMD = CHAR 31 [DWM.RIGHT]
IF :CMD = CHAR 32 [DWM.TOGGLE.MARK]
IF :CMD = "E [DWM.CMD.2 "E [[D DWM.EDIT] [R DWM.ERASE]]]
IF :CMD = "Z [DWM.CMD.2 "Z [[M DWM.FLUSH.MARKS]]]
IF :CMD = "P    ▶
   [DWM.CMD.2 "P [[O DWM.PRINTOUT] [: DWM.PRINTER]]]
IF :CMD = "D [DWM.CMD.2 "D [[: DWM.DISKSAVE []]]]
IF :CMD = "Q [SETCURSOR [Ø 23] STOP]
DWM.HIDE.CURSOR
SETCURSOR LIST (DWM.ITEM :COL :TABS)-1 :ROW
MAKE "INDEX (:COL-1)*:ROWS+:ROW+1
DWM.SHOW.CURSOR :INDEX
DWM.MAIN.LOOP RC :ROW :COL :INDEX
END

TO DWM.ITEM :NUM :LIST
IF :NUM=1 [OP FIRST :LIST]
OP DWM.ITEM :NUM-1 BF :LIST
END
```

```
TO DWM.CMD.2 :LETTER :LIST
DWM.PROMPT :LETTER
DWM.CMD.21 RC :LIST
DWM.SET.CURSOR
END

TO DWM.CMD.21 :CHAR :LIST
DWM.PROMPT CHAR 32
IF EMPTYP :LIST [TOOT 0 400 10 10 STOP]
IF EQUALP :CHAR FIRST FIRST :LIST [RUN BF FIRST :LIST STOP]
DWM.CMD.21 :CHAR BF :LIST
END

TO DWM.PROMPT :LETTER
SETCURSOR [0 23]
TYPE :LETTER
END

TO DWM.SHOW.CURSOR :INDEX
TYPE CHAR IF MEMBERP :INDEX :MARKED [170] [159]
END

TO DWM.HIDE.CURSOR
TYPE CHAR IF MEMBERP :INDEX :MARKED [42] [32]
END

TO DWM.SET.CURSOR
SETCURSOR LIST (DWM.ITEM :COL :TABS)-1 :ROW
END
```

MOVING THE POINTER

```
TO DWM.RIGHT
IF (:COL*:ROWS+:ROW+1) > :PROCS [TOOT 0 400 10 10 STOP]
MAKE "COL :COL+1
END

TO DWM.LEFT
IF :COL=1 [TOOT 0 400 10 10 STOP]
MAKE "COL :COL-1
END

TO DWM.DOWN
IF :INDEX+1 > :PROCS [TOOT 0 400 10 10 STOP]
IF :ROWS > :ROW+1 [MAKE "ROW :ROW+1 STOP]
MAKE "ROW 0
MAKE "COL :COL+1
END

TO DWM.UP
IF :ROW>0 [MAKE "ROW :ROW-1 STOP]
IF :COL=1 [TOOT 0 400 10 10 STOP]
MAKE "ROW :ROWS-1
MAKE "COL :COL-1
END
```

SETTING AND CLEARING MARKS

```
TO DWM.TOGGLE.MARK
IF MEMBERP :INDEX :MARKED [DWM.UNMARK] [DWM.MARK]
END

TO DWM.MARK
MAKE "MARKED FPUT :INDEX :MARKED
END

TO DWM.UNMARK
MAKE "MARKED DWM.REMOVE :INDEX :MARKED
END

TO DWM.REMOVE :THING :LIST
IF EMPTYP :LIST [OP []]
IF EQUALP :THING FIRST :LIST [OP BF :LIST]
OP FPUT FIRST :LIST DWM.REMOVE :THING BF :LIST
END

TO DWM.FLUSH.MARKS
MAKE "MARKED []
DWM.DRAW.MENU
END
```

EDIT

```
TO DWM.EDIT
EDIT DWM.MARKLIST
DWM.DRAW.MENU
END

TO DWM.MARKLIST
IF EMPTYP :MARKED [OP :ALL.PROCEDURES]
OP DWM.MARKLIST1 :ALL.PROCEDURES 1
END

TO DWM.MARKLIST1 :LIST :NUM
IF EMPTYP :LIST [OP []]
IF MEMBERP :NUM :MARKED [OP FPUT FIRST :LIST    ▸
    DWM.MARKLIST1 BF :LIST :NUM+1]
OP DWM.MARKLIST1 BF :LIST :NUM+1
END
```

PRINTOUT

```
TO DWM.PRINTOUT
CT
PO DWM.ITEM :INDEX :ALL.PROCEDURES
TYPE [TYPE A SPACE WHEN READY]
DWM.IGNORE RC
DWM.DRAW.MENU
END

TO DWM.IGNORE :CHAR
END
```

ERASE

```
TO DWM.ERASE
DWM.PROMPT (SE [REALLY ERASE]    ▸
    COUNT DWM.MARKLIST [PROCEDURES?])
IF NOT EQUALP RC "Y [DWM.CLEAR.PROMPT STOP]
DWM.CLEAR.PROMPT
ERASE DWM.MARKLIST
DWM.ERASE1 DWM.MARKLIST
MAKE "MARKED []
DWM.SIZE.MENU
DWM.DRAW.MENU
MAKE "ROW 0
MAKE "COL 1
END

TO DWM.ERASE1 :LIST
IF EMPTYP :LIST [STOP]
MAKE "ALL.PROCEDURES DWM.REMOVE FIRST :LIST :ALL.PROCEDURES
DWM.ERASE1 BF :LIST
END

TO DWM.CLEAR.PROMPT
DWM.PROMPT CHAR 32
REPEAT 35 [TYPE CHAR 32]
END
```

SAVE TO D: OR P:

```
TO DWM.DISKSAVE :FILE
DWM.PROMPT "FILE:
MAKE "FILE FIRST RL
CT
SETWRITE WORD "D: :FILE
DWM.SAVE DWM.MARKLIST
SETWRITE []
DWM.DRAW.MENU
END

TO DWM.PRINTER
CT
SETWRITE "P:
DWM.SAVE DWM.MARKLIST
SETWRITE []
DWM.DRAW.MENU
END

TO DWM.SAVE :LIST
IF EMPTYP :LIST [STOP]
PO FIRST :LIST
DWM.SAVE BF :LIST
END
```

A Logo Interpreter

Introduction

Suppose you were marooned on a desert island, with only your Atari and an assembler/editor cartridge. If you wanted to use Logo, you would have to write it yourself. How would you go about writing a computer language? You would have to write a program that runs the language. It is possible to write such a program, with some simplifications, in Logo itself.

Logo is an *interpreted* language. When you run your programs, Logo reads through them one instruction line at a time and executes each instruction in the line before proceeding to the next line. This is called interpreting a program.

Once you grasp the basic principles of interpreter design and operation, you could write an interpreter for any computer language, not just Logo. And you could write your interpreter in another language, like assembly language.

This project is about writing an interpreter for Logo in Atari Logo. We'll call this interpreter MLogo(for micro-Logo), to distinguish it from Atari Logo, which is an interpreter written in Atari machine language.

How to Use MLogo

To use MLogo, first initialize it (`INIT`), then start it (`LOGO`).

MLogo will prompt you for input with a ? in inverse video.

MLogo has fewer primitives than Atari Logo; among them are some list and arithmetic operations and some turtle commands.

```
?PRINT SUM 3 4
7
?PRINT FIRST BF "WALLABEE
A
?FD 100
?
```

You can write procedures in MLogo.

```
?TO POLY :SIDE :ANGLE
>FD :SIDE
>RT :ANGLE
>POLY :SIDE :ANGLE
>END
POLY DEFINED
```

MLogo is different from Atari Logo in some ways. MLogo doesn't care whether a line outputs or not. If you type:

```
SUM 3 4
```

By Jim Davis and Ed Hardebeck. An earlier version of this project was written by Henry Minsky.

the value 7 is just ignored. In Atari Logo you would get the error message:

```
YOU DON'T SAY WHAT TO DO WITH 7
```

MLogo doesn't have the STOP primitive. Every line of a user procedure is executed. It also doesn't have OP. The value of a user procedure is the value of the last line in it.

```
?TO GREET :WHO
>SE "HELLO :WHO
>END
GREET DEFINED
?PRINT GREET "ARTHUR
HELLO ARTHUR
```

If you typed this to Atari Logo, you'd get an error:

```
YOU DON'T SAY WHAT TO DO WITH [HELLO ARTHUR] IN GREET
```

Another difference is that all variables start with the empty list as their value.

```
?SHOW :NOVAL
[]
```

In Atari Logo, you'd get an error.

```
NOVAL HAS NO VALUE
```

If you try to use an undefined procedure in MLogo, you get a mysterious error message, then MLogo "crashes."

```
?ZIPPER 3
$ZIPPER HAS NO VALUE IN FSYMEVAL
```

After MLogo crashes, you are once again talking to Atari Logo. You must restart MLogo.

You may notice other differences as well. The reason for these differences is that it's difficult to implement Logo completely.

Now that you've had a chance to use MLogo and know what it does, we'll explain how it does it. The discussion, however, omits many details about interpreters.* Throughout this explanation we use the technical terms usually used by Logo implementors for describing Logo interpreters.

Interpretation Happens a Line at a Time

The structure of MLogo resembles that of Atari Logo. The normal action of Logo is to repeatedly type a prompt (?), *read* a line from the keyboard,

*For more information see *Structure and Interpretation of Computer Programs* by Gerald J. Sussman and Harold Abelson, MIT Press and McGraw-Hill, 1984.

and *evaluate* it. The top-level loop of MLogo is:

```
TO LOGOLOOP
IGNORE EVLINE GET.LINE
LOGOLOOP
END
```

The output of GET.LINE is a list of what the user typed.

EVLINE accepts a line as its input and carries out whatever instructions the line contains (for example, moves the turtle, prints a sentence, and so on).

```
TO EVLINE :LINE
OP EVLINE1 "$NOVALUE
END

TO EVLINE1 :VALUE
IF EMPTYP :LINE [OP :VALUE]
OP EVLINE1 EVAL NEXT.ITEM
END
```

EVLINE1 does the actual evaluation of the instructions of a line. The easiest way to understand it is to look first at its last line.

The operation NEXT.ITEM removes the first item from the variable LINE and outputs it. Each call to NEXT.ITEM removes one item and outputs it. In this way each item is inspected in turn.

```
TO NEXT.ITEM
OP NEXT.ITEM1 FIRST :LINE
END

TO NEXT.ITEM1 :FIRST
MAKE "LINE BF :LINE
OP :FIRST
END
```

EVAL takes an item, decides what kind of thing it is, and evaluates it to get its value. The value output from this call to EVAL is the input to EVLINE1 when it recurses. If there's nothing left on the line, this is the value to output. Otherwise, there's another instruction on the line.

When EVLINE calls EVLINE1, none of the line has been evaluated. If it should turn out that the line has no instructions (a blank line), then there is no value to output. $NOVALUE is just a default value.

Here's an example of how this recursion works. Suppose you type

```
FD 60 RT 90
```

to MLogo. LOGOLOOP calls EVLINE with the list [FD 60 RT 90] as input. NEXT.ITEM outputs FD, and :LINE is [60 RT 90]. FD is passed to EVAL for evaluation.

In evaluating FD, the number 60 would be removed from the line (it is an input to FD). When EVAL stops, the value of :LINE is [RT 90]. Since this is not an empty list, evaluation would continue.

The Rules of EVAL

The value of an item is determined by these rules.

- The value of a list is just the list.
- The value of a number is just the number.
- The value of a quoted word is the word itself without the quote.
- The value of a word prefaced with a colon (called "dots") is the value of the variable.
- Otherwise the word is the name of a procedure to call. Inputs to the procedure appear after the name of the procedure.

EVAL looks at an item to see what type it is, then carries out the appropriate rule.

How EVAL *Carries Out Its Rules*

```
TO EVAL :ITEM
IF LISTP :ITEM [OUTPUT :ITEM]
IF NUMBERP :ITEM [OUTPUT :ITEM]
IF QUOTED? :ITEM [OUTPUT UNQUOTE :ITEM]
IF DOTTED? :ITEM [OUTPUT GET.VARIABLE.VALUE UNDOT :ITEM]
OUTPUT EVAL.CALL FSYMEVAL :ITEM
END
```

EVAL's first test is for a list. If the item isn't a list, it must be a word. The remaining tests all assume the item is some kind of word and don't include WORDP as part of the test.

The predicate QUOTED? tests whether its input is a quoted word.

```
TO QUOTED? :WORD
OUTPUT EQUALP FIRST :WORD ""
END
```

The operation UNQUOTE removes the quote and outputs the word.

```
TO UNQUOTE :WORD
OUTPUT BF :WORD
END
```

In Logo a colon ("dots") before a word is a request for the value of a variable. The predicate DOTTED? checks this case.

```
TO DOTTED? :WORD
OUTPUT EQUALP FIRST :WORD ":
END
```

The procedure UNDOT outputs the word with the dots removed.

```
TO UNDOT :WORD
OUTPUT BF :WORD
END
```

GET.VARIABLE.VALUE outputs the value of a variable.

```
TO GET.VARIABLE.VALUE :WORD
IF BOUND? :WORD [OUTPUT SYMEVAL :WORD]
OUTPUT []
END
```

The predicate BOUND? checks whether the word has been assigned a value. If it has one, SYMEVAL outputs it, otherwise the value is []. We'll explain more about this later on.

Evaluating a Procedure Call

To evaluate a procedure call, we have to know some things about the procedure being called, such as how many inputs it has and whether it's a primitive or a user procedure. Information about a procedure is kept in the *definition* of the procedure. We'll describe definitions in detail later. For now, we'll just say that the operation FSYMEVAL outputs the definition and leave it at that.

When EVAL wishes to evaluate a procedure, it passes the definition of the procedure to EVAL.CALL.

```
TO EVAL.CALL :DEFINITION
OP APPLY :DEFINITION  EVAL.ARGS NARGS :DEFINITION
END
```

APPLY actually runs the procedure. It takes two inputs. The first is the definition of a procedure, the second is a list of values for the inputs. It causes the procedure to "do its thing," whatever that is, and APPLY outputs whatever the procedure outputs.

Before we can run the procedure, we have to get the values of its inputs. We usually refer to inputs as *arguments* (or *args,* for short).

The operation NARGS outputs the number of arguments this procedure expects. NARGS extracts this from the definition. (We'll see NARGS later.) This number is the input to EVAL.ARGS. EVAL.ARGS takes these inputs from the line being evaluated and evaluates each one, returning a list of the values.

To simplify MLogo, everything outputs. If commands were allowed in MLogo, as they are in Atari Logo, EVAL.CALL would have to know if an output was expected and make the proper complaint if an output was missing or an unexpected output showed up.

Inputs Require Recursive Evaluation

```
TO EVAL.ARGS :NARGS
IF EQUALP :NARGS 0 [OP []]
OUTPUT FPUT EVAL NEXT.ITEM EVAL.ARGS :NARGS - 1
END
```

EVAL.ARGS calls NEXT.ITEM to get the next item in the line being evaluated and makes a recursive call to EVAL to evaluate this item.

Here's an example. Suppose we type:

```
MAKE "DOGS 3
PRINT :DOGS
```

to MLogo. Our example begins after the MAKE, when evaluating the call to PRINT.

EVLINE gets [PRINT :DOGS].

NEXT.ITEM outputs PRINT.

EVAL gets PRINT, and decides it's the first word of a procedure call, so it calls EVAL.CALL with the definition.

EVAL.CALL calls NARGS to get the number of arguments that PRINT wants, and passes this to EVAL.ARGS.

EVAL.ARGS gets 1 as input, so it calls NEXT.ITEM, which outputs :DOGS. EVAL.ARGS calls EVAL to evaluate it.

EVAL gets :DOGS as input. This is a dotted word so GET.VARIABLE.VALUE is called with :DOGS. It outputs the value DOGS, which is 3. EVAL outputs 3.

EVAL.ARGS recurses. :NARGS - 1 is 0.

EVAL.ARGS is called with 0, so it outputs the empty list.

EVAL.ARGS outputs FPUT 3 [] to EVAL.CALL.

EVAL.CALL calls APPLY with the definition and the list of values returned by EVAL.ARGS, in this case [3].

APPLY invokes PRINT with an input of 3. Whatever APPLY outputs is what EVAL.CALL outputs.

EVAL.CALL returns to EVAL, which returns to EVLINE1.

Before we show how APPLY works, we'll give some details of procedure definitions.

Procedure Definitions

Both primitives and user procedures have definitions. Their definitions have some features in common and some differences.

In the remainder of this discussion, we refer to a primitive as an *sfun* (System FUNction), pronounced "ess-fun." Likewise we refer to a user procedure as a *ufun* (User FUNction), pronounced "you-fun." These are the terms usually used by Logo implementors.

Procedure definitions are kept in lists.

The first item in the list is the word SFUN or UFUN. The predicate SFUN? distinguishes sfuns from ufuns by inspecting this item.

```
TO SFUN? :DEFINITION
OP EQUALP FIRST :DEFINITION "SFUN
END
```

The second item is the number of inputs the procedure expects (this may be zero). The operation NARGS outputs this number.

```
TO NARGS :DEFINITION
OP FIRST BF :DEFINITION
END
```

The remaining items of the list differ for the two types of procedures. We'll show you the rest of an sfun definition now and take up ufuns later.

The third and final item in an sfun definition is the name of the Atari Logo procedure that implements the MLogo primitive.

The operation SFUN.FUNC outputs this procedure.

```
TO SFUN.FUNC :DEFINITION
OUTPUT FIRST BF BF :DEFINITION
END
```

If you print the names in the Logo workspace, you'll see definitions for all the MLogo primitives. All the definitions are in words beginning with $.

```
?SHOW :$PRINT
[SFUN 1 %PRINT]
?SHOW NARGS :$PRINT
1
?SHOW SFUN.FUNC :$PRINT
%PRINT
?SHOW :$SUM
[SFUN 2 SUM]
```

Sometimes an MLogo primitive is implemented directly by an Atari Logo primitive (for example, SUM), and sometimes by a procedure (%PRINT).

The operation MAKE.SFUN.DEF makes a definition for an sfun.

```
TO MAKE.SFUN.DEF :NARGS :FUNC
OUTPUT (SE "SFUN :NARGS :FUNC)
END
```

Now we can finish discussing the evaluation of a procedure call.

APPLY *Evaluates a Procedure Call*

The first input to APPLY is the definition of a procedure to evaluate. The second input is a list of input values for that procedure. Sfuns and ufuns are evaluated differently.

```
TO APPLY :DEFINITION :VALUES
IF SFUN? :DEFINITION [OP APPLY.SFUN :DEFINITION :VALUES]
OP APPLY.UFUN :DEFINITION  :VALUES
END
```

The command APPLY.SFUN applies an sfun to its inputs by building a list as input for RUN.

```
TO APPLY.SFUN :DEF :VALUES
OUTPUT RUN SE SFUN.FUNC :DEF ( QUOTIFY :VALUES )
END
```

Suppose you typed the following to MLogo:

```
MAKE "WHO "LOWELL
PRINT :WHO
```

While evaluating the call to PRINT, APPLY.SFUN would get the inputs [SFUN 1 %PRINT] (the definition of PRINT) and LOWELL (the value of the variable WHO).

Recall that SFUN.FUNC outputs the procedure that implements the sfun. In our example it will output %PRINT.

APPLY.SFUN would call RUN with the input [%PRINT "LOWELL]. RUN would call %PRINT on behalf of MLogo and output whatever it output.

Here's the MLogo sfun PRINT.

```
TO %PRINT :ARG
PRINT :ARG
OUTPUT :ARG
END
```

The operation QUOTIFY puts a quote in front of words that need it.

```
TO QUOTIFY :VALS
IF EMPTYP :VALS [OUTPUT []]
OUTPUT FPUT QUOTIFY1 FIRST :VALS QUOTIFY BF :VALS
END

TO QUOTIFY1 :VAL
IF NUMBERP :VAL [OP :VAL]
IF WORDP :VAL [OP WORD "" :VAL]
OUTPUT :VAL
END
```

Variable Values

The values of MLogo variables are stored in Atari Logo variables with slightly "funny" names. (This is useful for learning about how MLogo works. You can stop it and print out names. You can easily spot all MLogo variables by their names.)

The operation VSYM makes these names by adding a # to the front of the name. (The name VSYM stands for Variable SYMbol.)

```
TO VSYM :WORD
OP WORD "# :WORD
END
```

The command SET sets the value of an MLogo word, and SYMEVAL gets the value of an MLogo word. They both use VSYM to get the name of the word to use. VSYM *translates* from an MLogo name to an Atari Logo name.

```
TO SET :SYM :VAL
MAKE VSYM :SYM :VAL
END

TO SYMEVAL :SYM
OP THING VSYM :SYM
END
```

The predicate BOUND? tells whether there is a value for the word.

```
TO BOUND? :WORD
OP NAMEP VSYM :WORD
END
```

A second reason to use "funny" names for MLogo variables is that otherwise an MLogo user might set a variable with the same name as one used in the MLogo program itself. The results would be very strange. Adding the character guarantees that the names will never be the same.

Using a scheme like the one for variables, the definition of a procedure is kept in a variable whose name is the name of the procedure with a "$" prefix. The operation FSYM (Function SYMbol) outputs the Logo variable for the definition of the MLogo procedure.

```
TO FSYM :WORD
OP WORD "$ :WORD
END
```

The command FSET sets the definition of a procedure, and the operation FSYMEVAL outputs the definition of a procedure.

```
TO FSET :SYMBOL :DEF
MAKE FSYM :SYMBOL :DEF
END

TO FSYMEVAL :NAME
OUTPUT THING FSYM :NAME
END
```

How Sfuns Are Defined

The primitives we implemented are all very similar to familiar Logo primitives. In some cases we could call Logo primitives directly. But because every MLogo sfun must output, we had to write small Atari Logo procedures for those that don't output. These procedures call the sfun, then output a value. The value may just be TRUE.

```
TO %PRINT :ARG
PRINT :ARG
OUTPUT :ARG
END

TO %MAKE :SYM :VAL
SET.VARIABLE.VALUE :SYM :VAL
OUTPUT :VAL
END

TO %FD :N
FD :N
OP "TRUE
END
```

DEF.SFUN defines an sfun, that is, it associates the name of an sfun with the definition.

```
TO DEF.SFUN :NAME :NARGS :FUNC
FSET :NAME MAKE.SFUN.DEF :NARGS :FUNC
END
```

All sfun procedures' names begin with a percent sign to distinguish them from procedures that are part of MLogo itself. This makes it easy to spot all the MLogo sfuns in the workspace (except those implemented directly by Atari Logo primitives).

Ufun Definitions Include Arglist and Body

Like sfuns, ufuns have a definition, but the definition is slightly different. A user procedure consists of an *arglist* and a *body*. The arglist is a list of the input variables for the ufun. The body is a list of the lines of the procedure. Like sfuns, ufun definitions are lists.

If we had defined SQUARE by

```
TO SQUARE :N
PRINT :N
PRODUCT :N :N
END
```

. . . then the definition would be

```
?SHOW FSYMEVAL "SQUARE
[UFUN  1 [N] [[PRINT :N][PRODUCT :N :N]]]
```

The arglist is [N], the body is [[PRINT :N] [PRODUCT :N :N]]. Remember that MLogo ufun definitions are stored by the interpreter as Atari Logo *variables*, not as Atari Logo *procedures*.

The operation MAKE.UFUN.DEF makes a definition for a ufun. The operation UFUN.ARGLIST outputs the arglist from the definition, and the operation UFUN.BODY extracts the ufun body from the definition.

```
TO MAKE.UFUN.DEF :ARGS :BODY
OP (SE "UFUN COUNT :ARGS LIST :ARGS :BODY)
END

TO UFUN.ARGLIST :DEFINITION
OUTPUT FIRST BF BF :DEFINITION
END

TO UFUN.BODY :DEFINITION
OUTPUT FIRST BF BF BF :DEFINITION
END
```

Evaluating a Ufun Means Evaluating the Lines of Its Body

An sfun is a primitive, but a ufun body is a collection of lines, each requiring evaluation itself.

```
TO EVAL.BODY :LINES
OP EVAL.BODY1 "$NOVALUE :LINES
END

TO EVAL.BODY1 :VALUE :LINES
IF EMPTYP :LINES [OP :VALUE]
OP EVAL.BODY1 EVLINE FIRST :LINES BF :LINES
END
```

EVAL.BODY1 does the actual evaluation of the lines of the body. The value of the ufun is the value of the last line evaluated.

EVAL.BODY1 recurses in the same way EVLINE does. To understand it, look at the recursive call first. Each time EVAL.BODY1 recurses its first input is the value from evaluating the previous line. When the last line is evaluated, this is the value to output.

When EVAL.BODY1 is first called (from EVAL.BODY), it is passed $NOVALUE as a first input. When first called, EVAL.BODY1 has yet to evaluate a line, so there is no value to output from the ufun. If the ufun body is empty, then $NOVALUE is output. Otherwise there is at least one line to evaluate. EVAL.BODY1 evaluates the first line in the body, and recurses with this value and the remainder of the lines.

Ufuns Have Inputs with Names

Ufuns can have inputs. The title line (and therefore the arglist) of a ufun lists a set of variables that hold the inputs to the ufun. While a ufun is being evaluated, it can find its inputs in these variables.

For example, if you have the procedure:

```
TO GREET :WHO
PRINT SE "HELLO :WHO
END
```

and you type:

```
GREET "PHIL
```

Logo (either Atari Logo or MLogo) responds:

```
HELLO PHIL
```

Logo acts as if the value of :WHO had been set by MAKE before any of the instructions were evaluated. The effect is like what you could get by

```
?MAKE "WHO "PHIL
?PRINT SE "HELLO :PHIL
```

The difference is that after the ufun GREET is finished, the variable WHO has the same value it had before. Try it yourself if you don't already know this.

```
?MAKE "WHO [BAKED HAM]
?GREET "BOB
HELLO BOB
?SHOW :WHO
[BAKED HAM]
?MAKE "WHO "BOB
?PRINT SE "HELLO :WHO
HELLO BOB
?SHOW :WHO
BOB
```

Before EVAL.BODY can do its work, the previous values of certain variables must be saved before those variables receive new values. The input variables hold their values only for the duration of evaluation of the ufun, and then they have their old values restored.

The process of setting and restoring of values is referred to as *binding* the variables. Making binding work properly is one of the most difficult parts of writing an interpreter.

APPLY.UFUN is called by APPLY to evaluate a ufun. See APPLY.SFUN for comparison.

```
TO APPLY.UFUN :DEF :VALUES
BIND.ARGS UFUN.ARGLIST :DEF  :VALUES
OP CLEANUP EVAL.BODY UFUN.BODY :DEF
END
```

BIND.ARGS saves the old values of variables in the arglist, then sets the new values.

EVAL.BODY evaluates the forms of the body and outputs a value.

CLEANUP restores variables to their previous values. It outputs the value of the ufun.

How Binding Is Implemented

BIND.ARGS does two things. It saves old values and it sets new ones. It cooperates with CLEANUP, which restores the old values. These two procedures cooperate through the global variable BIND.STACK, which is where BIND.ARGS saves the values and CLEANUP finds them.

```
TO BIND.ARGS :ARGLIST :VALUES
PUSH "BIND.STACK BIND.FRAME :ARGLIST
SET.ARGS :ARGLIST :VALUES
END
```

The saved values are referred to collectively as a *bind frame.* The operation BIND.FRAME builds a bind frame for the variables in the arglist. BIND.ARGS uses PUSH to save this frame in the shared variable BIND.STACK, then calls SET.ARGS to set the new values.

```
TO SET.ARGS :NAMES :VALUES
IF EMPTYP :NAMES [STOP]
SET FIRST :NAMES FIRST :VALUES
SET.ARGS BF :NAMES BF :VALUES
END
```

SET.ARGS simply recurses through the argument list and the values. There is a one-to-one correspondence between the argument list and the values list. For each input there is a value.

After a ufun is evaluated it outputs a value, and this is the value that APPLY.UFUN should output as the value of the ufun it was asked to apply. But first the bound variables must be unbound. This is the purpose of CLEANUP.

```
TO CLEANUP :VALUE
UNBIND
OP :VALUE
END
```

CLEANUP's input is the output of the ufun. It holds onto this value while UNBIND undoes the binding, then returns the held value. UNBIND is explained later.

A bind frame enables the interpreter to restore the values of the input variables of a single ufun call. But a ufun can call other ufuns. We need one bind frame for each ufun call. There will be as many bind frames as the depth of calling. Bind frames are created as calls occur and cleaned up as the call returns. The most recently added frame is always the one to clean up.

We need to keep track of all these bind frames and ensure we bind and unbind in the same order calls and returns are made. To do this, we use a *stack*.

The Concept of a Stack

A stack is a method of arranging data. You can think of it as a pile of papers on a desk. Only the topmost sheet is visible (if the stack is neat) because it covers the others. If you add another sheet to the pile, it becomes the topmost. You can only touch the top sheet. If you remove it, a new top sheet is exposed.

This order of accessing is sometimes referred to as "Last In, First Out," because the last item added to the stack is the first one that can be removed.

Stacks Are Implemented by Lists

Most computers have machine instructions to implement stacks. But since we wrote MLogo in Logo and not in machine language, we had to implement stacks. We decided to use Logo lists to hold stacks, and to put the top of the stack at the front of the list so that we could use FIRST to get the top item on the stack and FPUT to add a new one.

PUSH puts something on the top of a stack. It takes two inputs. The first is the name of the word containing the stack, the second is the item to add to the stack.

```
TO PUSH :STACK :ITEM
MAKE :STACK FPUT :ITEM THING :STACK
END
```

PUSH makes a new list by adding the item to the old contents of the stack. This new list is assigned to the variable holding the stack.

The operation POP outputs the top value on the stack. This value is removed from the stack.

```
TO POP :STACK
OP POP1 :STACK THING :STACK
END

TO POP1 :STACK :LIST
MAKE :STACK BF :LIST
OP FIRST :LIST
END
```

The input to the operation POP is the variable holding the stack. THING of this variable outputs its value—the list holding the stack. The first item in the list is the item to output. Before outputting it, POP1 sets the stack variable to hold the BF of the list, thus removing the top item from the stack.

This example shows how stacks work.

```
?MAKE "STACK []
?PUSH "STACK 9
?SHOW :STACK
[9]
?PUSH "STACK 5
?PUSH "STACK 2
?SHOW :STACK
[2 5 9]
?PRINT POP "STACK
2
?SHOW :STACK
[5 9]
```

Bind Frame in Detail

A bind frame is a list of bindings. Each binding is a list of a name and a value. The name is the name of a variable that must be saved, and the value is the value it had at the time it was saved.

A typical bind frame might be

```
[[A 3][NAME [JAMES ALLEN]]]
```

This bind frame is holding two variable bindings, for A and NAME.

BIND.FRAME makes a bind frame. Its input is the argument list of a ufun. Each input is a variable whose value must be saved.

```
TO BIND.FRAME :ARGLIST
IF EMPTYP :ARGLIST [OP []]
OP FPUT BIND.ARG FIRST :ARGLIST BIND.FRAME BF :ARGLIST
END
```

BIND.FRAME recurses through the list, collecting one binding for each variable. The operation BIND.ARG makes a binding for a variable. It outputs a list of the variable name and the current value of the variable.

```
TO BIND.ARG :NAME
OP LIST :NAME GET.VARIABLE.VALUE :NAME
END
```

UNBIND pops a single frame off the stack and passes it to UNBIND.ARGS, which acts like BIND.FRAME in reverse, restoring each saved value in the frame.

```
TO UNBIND
UNBIND.ARGS POP "BIND.STACK
END

TO UNBIND.ARGS :FRAME
IF EMPTYP :FRAME [STOP]
UNBIND.ARG FIRST :FRAME
UNBIND.ARGS BF :FRAME
END

TO UNBIND.ARG :PAIR
SET FIRST :PAIR FIRST BF :PAIR
END
```

The interpreter's top-level procedure, LOGO, initializes BIND.STACK to hold an empty list.

```
TO LOGO
MAKE "BIND.STACK []
LOGOLOOP
END
```

The Sfun TO *Is Harder Than Others*

In Logo the primitive TO treats its inputs differently from all other sfuns. It does not evaluate them. The first input to TO is the name of the procedure to define. The rest of the inputs are the names of the inputs of the procedure being defined. These are written with dots to remind you that they are the inputs.

```
?TO SQUARE :A
>PRODUCT :A :A
>END
SQUARE DEFINED
```

TO manages the trick of not evaluating its inputs by lying to the evaluator about its number of inputs. It says it takes none but then goes and takes them off LINE (where the current line is kept) by itself. EVAL.ARGS evaluates the arguments as it collects them. This trick also lets TO take as many arguments as are present on the line.

The TO definition is:

```
[SFUN ∅ %TO]
```

The procedures that implement it are

```
TO %TO
OP TO1 NEXT.ITEM GATHER.ARGS
END

TO TO1 :NAME :ARGLIST
DEF.UFUN :NAME :ARGLIST READ.BODY
PRINT (SE :NAME "DEFINED)
OP "TRUE
END
```

The GATHER.ARGS operation pops the input names directly off LINE and removes the dots.

```
TO GATHER.ARGS
IF EMPTYP :LINE [OP []]
OP FPUT UNDOT NEXT.ITEM GATHER.ARGS
END
```

DEF.UFUN takes as inputs the name of the procedure to define, a list of its arguments, and its body, which is a list of the lines that make up the procedure.

```
TO DEF.UFUN :NAME :ARGS :BODY
FSET :NAME MAKE.UFUN.DEF :ARGS :BODY
END
```

MAKE.UFUN.DEF makes the actual definition. We have already seen it.

READ.BODY reads an entire ufun body, prompting with > before reading each line.

```
TO READ.BODY
OP READ.BODY1 READ.LINE
END

TO READ.BODY1 :LINE
IF EQUALP :LINE [END] [OP []]
OP FPUT :LINE READ.BODY1 READ.LINE
END
```

READ.BODY1 recurses, reading a line each time, until it gets a line END.

Reading Things You Type

Both TO and the LOGOLOOP need to get typein from the user. They don't want empty lines as input. Each has its own prompt character. They can both share INPUT.LINE.

```
TO GET.LINE
OP INPUT.LINE "?
END
```

```
TO READ.LINE
OP INPUT.LINE "≥
END

TO INPUT.LINE :PROMPT
TYPE :PROMPT
OP INPUT.LINE1 RL
END

TO INPUT.LINE1 :INPUT
IF NOT EMPTYP :INPUT [OP :INPUT]
OP INPUT.LINE :PROMPT
END
```

INPUT.LINE1 recurses until the user types a line that isn't empty.

Some Improvements

Here are some modifications to MLogo to make it more like Atari Logo. We didn't include them in MLogo because we wanted to keep it simple to explain. If you want to have these extra features, you can type in the following procedures.

First, a synonym for PRINT.

```
DEF.SFUN "PR 1 "%PRINT
```

Here's the sfun PO:

```
TO %PO :NAME
TYPE SE "TO :NAME
PO.ARGS UFUN.ARGLIST FSYMEVAL :NAME
PO.BODY UFUN.BODY FSYMEVAL :NAME
OP :NAME
END

TO PO.ARGS :ARGLIST
IF EMPTYP :ARGLIST [PRINT [] STOP]
TYPE "\
TYPE ":
TYPE FIRST :ARGLIST
PO.ARGS BF :ARGLIST
END

TO PO.BODY :LINES
IF EMPTYP :LINES [PR "END STOP]
PRINT FIRST :LINES
PO.BODY BF :LINES
END
```

PO is by far the longest sfun yet, because there is no useful Atari Logo primitive for it. The Atari Logo primitive PO prints out Atari Logo user procedures, which are not stored like MLogo user procedures.

The procedure PO.ARGS prints each word in the argument list,

preceded by a space and a colon. (In the second line of PO.ARGS, a space appears after the backslash even though you can't tell from this listing.)

To add OP to MLogo we have to change EVAL.BODY and EVAL.BODY1. As is, each line is always evaluated. By adding a flag variable OP? we can cause evaluation to stop.

Here's the sfun OP.

```
TO %OP :VALUE
MAKE "OP :VALUE
MAKE "OP? "TRUE
OP "TRUE
END

DEF.SFUN "OP 1 "%OP
```

The input to OP is the value to output. This value is stored in the variable OP for reference by the evaluator. %OP sets the flag OP?, which causes the evaluation of the body to stop.

We have to modify the evaluator to check these flags.

```
TO EVAL.BODY :LINES
OP EVAL.BODY1 :LINES "FALSE
END

TO EVAL.BODY1 :LINES :OP?
IF EMPTYP :LINES [OP "$NOVALUE]
IGNORE EVLINE FIRST :LINES
IF :OP? [OP :OP]
OP EVAL.BODY1 BF :LINES "FALSE
```

PROGRAM LISTING

```
TO LOGO
MAKE "BIND.STACK []
LOGOLOOP
END

TO LOGOLOOP
IGNORE EVLINE GET.LINE
LOGOLOOP
END

TO EVLINE :LINE
OP EVLINE1 "$NOVALUE
END

TO EVLINE1 :VALUE
IF EMPTYP :LINE [OP :VALUE]
OP EVLINE1 EVAL NEXT.ITEM
END

TO NEXT.ITEM
OP NEXT.ITEM1 FIRST :LINE
END

TO NEXT.ITEM1 :FIRST
MAKE "LINE BF :LINE
OP :FIRST
END

TO EVAL :ITEM
IF LISTP :ITEM [OUTPUT :ITEM]
IF NUMBERP :ITEM [OUTPUT :ITEM]
IF QUOTED? :ITEM [OUTPUT UNQUOTE ▸
     :ITEM]
IF DOTTED? :ITEM [OUTPUT ▸
     GET.VARIABLE.VALUE UNDOT :ITEM]
OUTPUT EVAL.CALL FSYMEVAL :ITEM
END

TO QUOTED? :WORD
OUTPUT EQUALP FIRST :WORD ""
END

TO UNQUOTE :WORD
OUTPUT BF :WORD
END
```

```
TO DOTTED? :WORD
OUTPUT EQUALP FIRST :WORD ":
END

TO UNDOT :WORD
OUTPUT BF :WORD
END

TO GET.VARIABLE.VALUE :WORD
IF BOUND? :WORD [OUTPUT SYMEVAL :WORD]
OUTPUT []
END

TO VSYM :WORD
OP WORD "# :WORD
END

TO SET :SYM :VAL
MAKE VSYM :SYM :VAL
END

TO SYMEVAL :SYM
OP THING VSYM :SYM
END

TO BOUND? :WORD
OP NAMEP VSYM :WORD
END

TO FSYM :WORD
OP WORD "$ :WORD
END

TO FSYMEVAL :NAME
OUTPUT THING FSYM :NAME
END

TO FSET :SYMBOL :DEF
MAKE FSYM :SYMBOL :DEF
END

TO EVAL.CALL :DEFINITION
OP APPLY :DEFINITION  EVAL.ARGS NARGS ▸
     :DEFINITION
END

TO MAKE.SFUN.DEF :NARGS :FUNC
OUTPUT (SE "SFUN :NARGS :FUNC)
END

TO MAKE.UFUN.DEF :ARGS :BODY
OP (SE "UFUN COUNT :ARGS LIST :ARGS ▸
     :BODY)
END
```

```
TO SFUN? :DEFINITION
OP EQUALP FIRST :DEFINITION "SFUN
END

TO NARGS :DEFINITION
OP FIRST BF :DEFINITION
END

TO SFUN.FUNC :DEFINITION
OUTPUT FIRST BF BF :DEFINITION
END

TO UFUN.ARGLIST :DEFINITION
OUTPUT FIRST BF BF :DEFINITION
END

TO UFUN.BODY :DEFINITION
OUTPUT FIRST BF BF BF :DEFINITION
END


TO APPLY :DEFINITION :VALUES
IF SFUN? :DEFINITION [OP APPLY.SFUN ▸
     :DEFINITION :VALUES]
OP APPLY.UFUN :DEFINITION  :VALUES
END

TO APPLY.SFUN :DEF :VALUES
OUTPUT RUN SE SFUN.FUNC :DEF ( QUOTIFY ▸
     :VALUES )
END

TO QUOTIFY :VALS
IF EMPTYP :VALS [OUTPUT []]
OUTPUT FPUT QUOTIFY1 FIRST :VALS ▸
     QUOTIFY BF :VALS
END

TO QUOTIFY1 :VAL
IF NUMBERP :VAL [OP :VAL]
IF WORDP :VAL [OP WORD "" :VAL]
OUTPUT :VAL
END


TO EVAL.ARGS :NARGS
IF EQUALP :NARGS 0 [OP []]
OUTPUT FPUT EVAL NEXT.ITEM EVAL.ARGS ▸
     :NARGS - 1
END


TO APPLY.UFUN :DEF :VALUES
BIND.ARGS UFUN.ARGLIST :DEF  :VALUES
OP CLEANUP EVAL.BODY UFUN.BODY :DEF
END
```

```
TO BIND.ARGS :ARGLIST :VALUES
PUSH "BIND.STACK BIND.FRAME :ARGLIST
SET.ARGS :ARGLIST :VALUES
END

TO SET.ARGS :NAMES :VALUES
IF EMPTYP :NAMES [STOP]
SET FIRST :NAMES FIRST :VALUES
SET.ARGS BF :NAMES BF :VALUES
END

TO CLEANUP :VALUE
UNBIND
OP :VALUE
END

TO BIND.FRAME :ARGLIST
IF EMPTYP :ARGLIST [OP []]
OP FPUT BIND.ARG FIRST :ARGLIST ▸
     BIND.FRAME BF :ARGLIST
END

TO BIND.ARG :NAME
OP LIST :NAME GET.VARIABLE.VALUE :NAME
END

TO UNBIND
UNBIND.ARGS POP "BIND.STACK
END

TO UNBIND.ARGS :FRAME
IF EMPTYP :FRAME [STOP]
UNBIND.ARG FIRST :FRAME
UNBIND.ARGS BF :FRAME
END

TO UNBIND.ARG :PAIR
SET FIRST :PAIR FIRST BF :PAIR
END

TO EVAL.BODY :LINES
OP EVAL.BODY1 "$NOVALUE :LINES
END

TO EVAL.BODY1 :VALUE :LINES
IF EMPTYP :LINES [OP :VALUE]
OP EVAL.BODY1 EVLINE FIRST :LINES BF ▸
     :LINES
END

TO %PRINT :ARG
PRINT :ARG
OUTPUT :ARG
END

TO %IF :PRED :C1 :C2
IF :PRED [OP EVLINE :C1] [OP EVLINE ▸
     :C2]
END

TO %MAKE :SYM :VAL
SET :SYM :VAL
OUTPUT :VAL
END

TO %FD :N
FD :N OP "TRUE
END

TO %RT :N
RT :N OP "TRUE
END

TO %CS
CS OP "TRUE
END

TO %TO
OP TO1 NEXT.ITEM GATHER.ARGS
END

TO TO1 :NAME :ARGLIST
DEF.UFUN :NAME :ARGLIST READ.BODY
PRINT (SE :NAME "DEFINED)
OP "TRUE
END

TO GATHER.ARGS
IF EMPTYP :LINE [OP []]
OP FPUT UNDOT NEXT.ITEM GATHER.ARGS
END

TO DEF.SFUN :NAME :NARGS :FUNC
FSET :NAME MAKE.SFUN.DEF  :NARGS :FUNC
END

TO DEF.UFUN :NAME :ARGS :BODY
FSET :NAME MAKE.UFUN.DEF :ARGS :BODY
END

TO INIT
INITPRIMS
END

TO DEF.SSFUN :NAME :NARGS
DEF.SFUN :NAME :NARGS :NAME
END
```

```
TO INITPRIMS
DEF.SSFUN "SUM 2
DEF.SSFUN "PRODUCT 2
DEF.SSFUN "EMPTYP 1
DEF.SSFUN "EQUALP 2
DEF.SSFUN "LIST 2
DEF.SSFUN "FIRST 1
DEF.SSFUN "BF 1
DEF.SSFUN "SE 2
DEF.SSFUN "WORD 2
DEF.SFUN "RT 1 "%RT
DEF.SFUN "FD 1 "%FD
DEF.SFUN "CS 0 "%CS
DEF.SFUN "THING 1 "GET.VARIABLE.VALUE
DEF.SFUN "MAKE 2 "%MAKE
DEF.SFUN "PRINT 1 "%PRINT
DEF.SFUN "IF 3 "%IF
DEF.SFUN "TO 0 "%TO
END

TO READ.BODY
OP READ.BODY1 READ.LINE
END

TO READ.BODY1 :LINE
IF EQUALP :LINE [END] [OP []]
OP FPUT :LINE READ.BODY1 READ.LINE
END

TO GET.LINE
OP INPUT.LINE "?
END

TO READ.LINE
OP INPUT.LINE "≥
END

TO INPUT.LINE :PROMPT
TYPE :PROMPT
OP INPUT.LINE1 RL
END

TO INPUT.LINE1 :INPUT
IF NOT EMPTYP :INPUT [OP :INPUT]
OP INPUT.LINE :PROMPT
END

TO PUSH :STACK :ITEM
MAKE :STACK FPUT :ITEM THING :STACK
END

TO POP :STACK
OP POP1 :STACK THING :STACK
END

TO POP1 :STACK :LIST
MAKE :STACK BF :LIST
OP FIRST :LIST
END

TO IGNORE :X
END
```

Map

Have you ever written a procedure like this:

```
TO LINEPRINT :LIST
IF EMPTYP :LIST [STOP]
PRINT FIRST :LIST
LINEPRINT BF :LIST
END
```

By Brian Harvey.

Or like this:

```
TO TUNE :NOTES
IF EMPTYP :NOTES [STOP]
TOOT 0 FIRST :NOTES 15 30
TUNE BF :NOTES
END
```

Or like this:

```
TO FLASH :COLORS
IF EMPTYP :COLORS [STOP]
WAIT 60
SETBG FIRST :COLORS
FLASH BF :COLORS
END
```

All of these procedures have a common pattern. They go through a list, doing something with each member of the list, and then stop when they get to the end of the list. The procedures *differ* in what they do with the members of their input list. In one case it's a list of things to print; in the second it's a list of frequencies of musical notes; in the third it's a list of color numbers. But they all share this structure:

```
TO procedure.name :LIST
IF EMPTYP :LIST [STOP]
do.something.with FIRST :LIST
procedure.name BF :LIST
END
```

You can think of this skeleton procedure as a template for many procedures that do similar work for you.

Mapping Commands

You can write a single procedure that does all these things. What's special about it is that it is a *general* tool that can apply *any* procedure to each member of a list. This general process is called *mapping* the procedure over the list, so we call this general procedure MAP. Here are some examples.

```
?MAP [PRINT] [VANILLA CHOCOLATE GINGER LEMON]
VANILLA
CHOCOLATE
GINGER
LEMON
?

?MAP [PRINT FIRST] [VANILLA CHOCOLATE GINGER LEMON]
V
C
G
L
?
```

```
?MAP [TYPE FIRST] [EVERY GOOD BOY DOES FINE]
EGBDF?
```

The first example of using MAP is equivalent to the procedure LINEPRINT with which we started this discussion. The first input to MAP says what you want to do to each member of the input list (in this example, PRINT it). The second input is the list over which you are mapping. So the instruction

```
MAP [PRINT] [THIS IS A LIST]
```

is equivalent to

```
LINEPRINT [THIS IS A LIST]
```

Here are the procedure definitions.

```
TO MAP :TEMPLATE :LIST
IF EMPTYP :LIST [STOP]
RUN LPUT QUOTED FIRST :LIST :TEMPLATE
MAP :TEMPLATE BF :LIST
END

TO QUOTED :THING
IF LISTP :THING [OP :THING]
OP WORD "" :THING
END
```

You can use MAP with more complicated instructions than just PRINT. In the second example, the first input to MAP is the list [PRINT FIRST]. This example works as if we'd written a special procedure like this:

```
TO FIRST.PRINT :LIST
IF EMPTYP :LIST [STOP]
PRINT FIRST FIRST :LIST
FIRST.PRINT BF :LIST
END
```

We can use MAP to obtain the same effect as the FLASH procedure we showed earlier.

```
MAP [WAIT 60 SETBG] [0 88 74 7]
```

To get the same effect as our TUNE procedure, we have to work a little harder. The problem is that the frequency input to TOOT comes in the middle of the instruction, like this:

```
TOOT 0 FIRST :NOTES 15 30
```

MAP expects to put each member of the list at the end of an instruction, not in the middle. What we have to do is write an auxiliary procedure that takes the frequency as a single input:

```
TO NOTE :FREQ
TOOT 0 :FREQ 15 30
END
```

Now we can use MAP to get the same effect as TUNE:

```
MAP [NOTE] [440 880 220 440]
```

How It Works

What makes it possible for MAP to be a general-purpose tool instead of a procedure for a specific purpose is its use of Logo's RUN command. This replaces the specific commands like PRINT or TOOT or SETBG in the earlier examples. The input to RUN is a Logo instruction that is assembled out of two parts: the *template,* which is the first input to MAP, and one member of the list, which is MAP's second input.

Let's look at an example. If we say

```
MAP [PRINT] [VANILLA CHOCOLATE GINGER LEMON]
```

then MAP has to carry out these four instructions:

```
PRINT "VANILLA
PRINT "CHOCOLATE
PRINT "GINGER
PRINT "LEMON
```

Each of these four instructions is made by combining the template [PRINT] with one member of [VANILLA CHOCOLATE GINGER LEMON]. The combination is made using LPUT, which adds the list member at the end of the template. For example, the expression

```
LPUT "VANILLA [PRINT]
```

outputs the list

```
[PRINT "VANILLA]
```

The procedure MAP itself has much the same pattern as the examples at the beginning of this discussion. The first instruction inside MAP is the IF EMPTYP stop rule; the last instruction is the recursive use of MAP with the BUTFIRST of the input list. Compare MAP with LINEPRINT, for example:

```
TO LINEPRINT :LIST
IF EMPTYP :LIST [STOP]
PRINT FIRST :LIST
LINEPRINT BF :LIST
END

TO MAP :TEMPLATE :LIST
IF EMPTYP :LIST [STOP]
RUN LPUT QUOTED FIRST :LIST :TEMPLATE
MAP :TEMPLATE BF :LIST
END
```

One possibly confusing detail in MAP has to do with quotation marks. Notice that if you want Logo to print the word VANILLA, you can't say

```
PRINT VANILLA                                    Wrong!
```

but must quote the input to PRINT:

```
PRINT "VANILLA
```

To assemble this instruction, the first input to LPUT must be the word "VANILLA, including the quotation mark as part of the word. The procedure QUOTED is used by MAP to supply the needed quotation marks.

Mapping Operations

So far, the templates we've used have been *commands.* That is, they have been Logo procedures that do something external, like print something, make a sound, or change the color of the screen. An even more powerful facility is to map *operations* over a list, producing (outputting) a new list of the results. Perhaps an example will make this clearer.

```
?SHOW MAP.LIST [FIRST] [THIS IS A LIST]
[T I A L]
?

?SHOW MAP.LIST [SQRT] [1 2 3 4]
[1 1.414214 1.732051 2]
?
```

Like MAP, MAP.LIST generalizes a common pattern of Logo procedures. The examples here could have been written as special-purpose procedures this way:

```
TO EVERY.FIRST :LIST
IF EMPTYP :LIST [OP []]
OP FPUT (FIRST FIRST :LIST) (EVERY.FIRST BF :LIST)
END

TO EVERY.SQRT :LIST
IF EMPTYP :LIST [OP []]
OP FPUT (SQRT FIRST :LIST) (EVERY.SQRT BF :LIST)
END
```

MAP.LIST is an operation. Its output is a list of the same length as its second input. Each member of the output list is the result of applying the template to a member of the input list.

MAP.LIST itself follows the same pattern it generalizes.

```
TO MAP.LIST :TEMPLATE :LIST
IF EMPTYP :LIST [OP []]
OP FPUT (RUN LPUT QUOTED FIRST :LIST :TEMPLATE)
        (MAP.LIST :TEMPLATE BF :LIST)
END
```

Here the first input to FPUT is the same expression that was used to assemble the instructions in MAP.

An example of using MAP.LIST to apply a procedure to each word of a sentence is this program to translate a sentence into Pig Latin.

```
TO PIGLATIN :WORD
IF MEMBERP FIRST :WORD [A E I O U Y] [OP WORD :WORD "AY]
OP PIGLATIN WORD BF :WORD FIRST :WORD
END

?PRINT PIGLATIN 'HELLO
ELLOHAY
?PRINT MAP.LIST [PIGLATIN] [THIS IS GREEK TO ME]
ISTHAY ISAY EEKGRAY OTAY EMAY
?
```

Mapping Over Words

In Logo, we can assemble letters into words, just as we can assemble words into lists. We can extend the idea of mapping to apply a procedure to each letter of a word.

```
TO MAP.WORD :TEMPLATE :WORD
IF EMPTYP :WORD [OP "]
OP WORD (RUN LPUT QUOTED FIRST :WORD :TEMPLATE)
        (MAP.WORD :TEMPLATE BF :WORD)
END
```

MAP.WORD is the same as MAP.LIST, except that it uses WORD instead of FPUT as the combining operation, and it builds onto an empty word instead of an empty list.

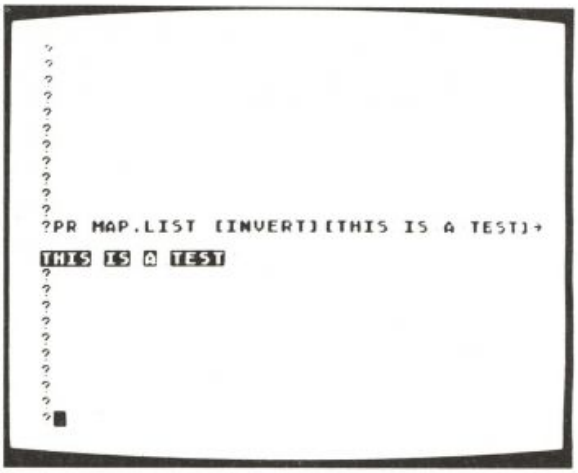
Here is an example of how to use MAP.WORD. Suppose you want to print a word in inverse video (black on white). On the Atari computer, to print any character in inverse video, you must add 128 to the code that represents that character.

```
?PRINT MAP.WORD [CHAR 128+ASCII] 'HELLO
```


```
?
```

If we put this into a procedure, we can print an entire sentence with each word inverted by combining MAP.WORD and MAP.LIST.

```
TO INVERT :WORD
OP MAP.WORD [CHAR 128+ASCII] :WORD
END

?PRINT MAP.LIST [INVERT] [THIS IS A TEST.]
```



List Reduction

There is one more way in which an operation can be applied to the members of a list. Consider an operation with two inputs, like SUM or PRODUCT. It is often convenient to be able to add up all the numbers in a list, or multiply them together. Of course, as in the earlier situations, we could write special-purpose procedures.

```
TO ADD :LIST
IF EMPTYP :LIST [OP 0]
OP SUM (FIRST :LIST) (ADD BF :LIST)
END

TO MULTIPLY :LIST
IF EMPTYP :LIST [OP 1]
OP PRODUCT (FIRST :LIST) (MULTIPLY BF :LIST)

?PR ADD [1 2 3 4]
10
?PR MULTIPLY [1 2 3 4]
24
?
```

What we'd like to do is produce a general tool for these situations.

```
?PR REDUCE [SUM] [1 2 3 4]
10
```

```
?PR REDUCE [PRODUCT] [1 2 3 4]
24
?
```

There is one slight complication that prevents REDUCE from following exactly the pattern of ADD and MULTIPLY. The problem is that each of those procedures knows about the *identity element* for the corresponding operation. The identity element is the value to start with when the input list is empty: 0 for SUM, 1 for PRODUCT. To make REDUCE a general tool, we want to avoid building this kind of information into it. The solution is to apply REDUCE recursively only down to the point where there are *two* members remaining in the input list, then just apply the template to those two. The resulting procedure is a little messy, but if you go through it carefully you'll see that it's really much like the mapping procedures we've used before.

```
TO REDUCE :TEMPLATE :LIST
IF EMPTYP BF :LIST [OP FIRST :LIST]
IF EMPTYP BF BF :LIST [OP RUN SE :TEMPLATE
        LIST (QUOTED FIRST :LIST) (QUOTED FIRST BF :LIST)]
OP RUN SE :TEMPLATE LIST (QUOTED FIRST :LIST)
        (QUOTED REDUCE :TEMPLATE BF :LIST)
END
```

Here are more examples of how REDUCE can be used.

```
?PRINT REDUCE [WORD] [A B C D]
ABCD
?

TO REVERSE :LIST
OP REDUCE [LPUT] LPUT [] :LIST
END

?SHOW REVERSE [A B C D]
[D C B A]
?
```

SUGGESTIONS

- You could modify these procedures so that the list members could be inserted anywhere in the template, instead of only at the end. For example, the music example that earlier required writing an auxiliary procedure NOTE could instead be written

```
MAP [TOOT 0 ? 15 30] [440 880 220 440]
```

  where the question mark indicates the position in the template into which the members of the input list are placed.
- The general name for doing something over and over is *iteration*. Mapping is a particular kind of iteration, based on using the members of a list, one after the other. Other kinds of iteration can also be

invented using the RUN primitive. For example, here is an iteration procedure that tests a predicate to control the repetition.

```
TO WHILE :PREDICATE :COMMAND
IF NOT RUN :PREDICATE [STOP]
RUN :COMMAND
WHILE :PREDICATE :COMMAND
END

?CS
?WHILE [HEADING < 270] [FD 10 RT 10]
?
```

You might try to write a procedure to create numeric iteration.

```
?STEP "NUM 3 7 [PRINT :NUM * :NUM]
9
16
25
36
49
?
```

- Use MAP.WORD and MAP.LIST to implement a *substitution cipher.* A cipher is a technique for protecting secret messages by transforming each letter into some other form. (Ciphers are sometimes called *codes,* but, strictly speaking, a code is a technique that transforms a word by looking it up in a dictionary, rather than by manipulating it letter by letter. A foreign language is like a code.) Write a procedure that takes a single letter as input and outputs some secret representation of the input letter. Then you can encipher a word by applying MAP.WORD to it, and you can encipher a sentence by applying MAP.LIST to encipher each word. The example of inverse video works like a cipher, although of course the result isn't very secret.
- MAP.LIST uses FPUT to accumulate the results for each member of the input list, and MAP.WORD uses WORD to accumulate its results. Logo has other accumulating operations: SE, LIST, and LPUT. Try writing versions of MAP.LIST that use each of these. Are any of them useful?
- Here is a tricky example.

```
TO FLATTEN :LIST
IF WORDP :LIST [OP :LIST]
OP REDUCE [SE] MAP.LIST [FLATTEN] :LIST
END

?SHOW FLATTEN [[THIS IS] [A [LIST]]]
[THIS IS A LIST]
?
```

FLATTEN combines iteration over a list, list reduction, and recursion, since the template input to MAP.LIST uses FLATTEN itself. The procedure converts any list into a *flat list,* one that has only words as

members. Can you see why both REDUCE and MAP.LIST must be used? Compare the result of FLATTEN to these:

```
SHOW REDUCE [SE] [[THIS IS] [A [LIST]]]
SHOW MAP.LIST [FLATTEN] [[THIS IS] [A [LIST]]]
```

PROGRAM LISTING

```
TO MAP :TEMPLATE :LIST
IF EMPTYP :LIST [STOP]
RUN LPUT QUOTED FIRST :LIST :TEMPLATE
MAP :TEMPLATE BF :LIST
END

TO QUOTED :THING
IF LISTP :THING [OP :THING]
OP WORD "" :THING
END

TO MAP.LIST :TEMPLATE :LIST
IF EMPTYP :LIST [OP []]
OP FPUT (RUN LPUT QUOTED FIRST :LIST ▸
    :TEMPLATE) (MAP.LIST :TEMPLATE BF ▸
    :LIST)
END

TO MAP.WORD :TEMPLATE :WORD
IF EMPTYP :WORD [OP "]
OP WORD (RUN LPUT QUOTED FIRST :WORD ▸
    :TEMPLATE) (MAP.WORD :TEMPLATE BF ▸
    :WORD)
END

TO REDUCE :TEMPLATE :LIST
IF EMPTYP BF :LIST [OP FIRST :LIST]
IF EMPTYP BF BF :LIST [OP RUN SE ▸
    :TEMPLATE LIST (QUOTED FIRST ▸
    :LIST) (QUOTED FIRST BF :LIST)]
OP RUN SE :TEMPLATE LIST (QUOTED FIRST ▸
    :LIST) (QUOTED REDUCE :TEMPLATE ▸
    BF :LIST)
END
```

Mergesort

People often want to use computers to *sort* information of various kinds. For example, you may want to list your friends' addresses in alphabetical order, or you may want the same information arranged in order of their birthdays to remind you when to send cards. Programmers have invented many different techniques to solve the sorting problem. Generally, the methods that are easy to understand tend to run slowly, while the faster methods are rather complicated. Here is a method that is medium-fast and medium-tricky. Its name is *mergesort*.

In Logo, we'll represent the information we want to sort as a list of items. The general strategy is this:

1. Divide the list into two smaller parts.
2. Sort each part separately.
3. Merge the two sorted lists into one big sorted list.

This may not seem like much of a strategy, because we are still left with the problem of sorting the smaller lists in the second step. But the clever part

Program by Danny Hillis; write-up by Brian Harvey.

is that if we keep applying the strategy to the smaller lists, eventually we get lists with just one member, and we can simply declare these lists sorted.

Here's a specific example. To make it easy to read, we'll sort a list of numbers in size order. Start with this list:

```
[14 3 27 1 10 5]
```

Divide it into two smaller lists.

```
[14 3 27]                [1 10 5]
```

Now sort the first of the smaller lists. To do that, divide it into two smaller lists.

```
[14 3]                   [27]
```

Now sort the first of *these* lists, again by dividing it into two smaller lists.

```
[14]                     [3]
```

Each of these lists has only one member, so each is already sorted. Now we merge them to get

```
[3 14]
```

Now we can merge this list with its "partner," which is the list `[27]`. The result is

```
[3 14 27]
```

The next step is to sort the "partner" of *this* list, namely the list `[1 10 5]`. This also involves dividing it into smaller lists, as before. To make this example shorter, we'll skip the steps of sorting the list `[1 10 5]`. Finally we are left with two sorted lists:

```
[3 14 27]                [1 5 10]
```

The last step is to merge these:

```
[1 3 5 10 14 27]
```

Dividing a List into Two Parts

The first step in the sorting process is to divide a list into two parts. To do that, we can use procedures `FIRST.PART` and `LAST.PART`.

```
TO FIRST.PART :LIST
OP FIRST.N (INT (COUNT :LIST)/2) :LIST
END

TO FIRST.N :NUMBER :LIST
IF :NUMBER=0 [OP :LIST]
OP FIRST.N :NUMBER-1 BL :LIST
END

TO LAST.PART :LIST
OP LAST.N (INT (1+COUNT :LIST)/2) :LIST
END
```

```
TO LAST.N :NUMBER :LIST
IF :NUMBER=0 [OP :LIST]
OP LAST.N :NUMBER-1 BF :LIST
END
```

You may notice that LAST.PART refers to 1 + COUNT :LIST instead of COUNT :LIST. The reason for this difference is that if the input list has an odd number of members, we must divide the list into two pieces that differ in length by one. For example, if the input list has five members, FIRST.PART will output the first three members of the list and LAST.PART will output the last two members.

```
?SHOW FIRST.PART [14 3 27 1 10]
[14 3 27]
?SHOW LAST.PART [14 3 27 1 10]
[1 10]
?
```

Merging Two Ordered Lists

The last step of the sorting procedure is to *merge* two lists. The MERGE procedure assumes that each of the two lists is already in the correct order. MERGE takes two inputs, namely, the two lists.

MERGE compares the first member of one input list with the first member of the other list. One of these becomes the first member of the final merged list; MERGE is applied recursively to the remaining members of the input lists.

```
TO MERGE :A :B
IF EMPTYP :A [OP :B]
IF EMPTYP :B [OP :A]
IF COMPARE FIRST :A FIRST :B
        [OP FPUT FIRST :A MERGE BF :A :B]
OP FPUT FIRST :B MERGE :A BF :B
END
```

MERGE uses a subprocedure, COMPARE, which tells whether one item should come before or after another. COMPARE takes two inputs. It outputs the word TRUE if the first input comes before the second input, or FALSE otherwise.

You can write different versions of COMPARE depending on what ordering you want to use for your sorted lists. If you are sorting numbers by size, as in the earlier example, you can use this version:

```
TO COMPARE :A :B
OUTPUT :A < :B
END
```

If you want to sort words alphabetically, or use some other ordering, you need a more complicated version of COMPARE. We'll show an example later.

Putting It All Together

We've written the easy parts of this sorting method. The hard part is putting it all together. The main procedure SORT does this. It takes one input, which must be a list. It outputs the same list, but with its members in sorted order.

```
TO SORT :A
IF EMPTYP :A [OP []]
IF EMPTYP BF :A [OP :A]
OP MERGE (SORT FIRST.PART :A) (SORT LAST.PART :A)
END
```

If the input list is empty, or has only one member, then the list is already sorted. SORT outputs the list unchanged. For larger lists, SORT goes through the steps we described at the beginning.

1. It uses FIRST.PART and LAST.PART to divide the list in two.
2. It uses SORT to sort each of these smaller lists.
3. It uses MERGE to combine the resulting ordered lists.

Alphabetical Order

Sometimes we want to deal with information composed of words or sentences, rather than numbers. Here are procedures to alphabetize lists of words.

```
TO COLLATE.BEFORE :A :B
IF EMPTYP :A [OP "TRUE]
IF EMPTYP :B [OP "FALSE]
IF COLLATE.BEFORE.WORD FIRST :A FIRST :B [OP "TRUE]
IF NOT EQUALP FIRST :A FIRST :B [OP "FALSE]
OP COLLATE.BEFORE BF :A BF :B
END

TO COLLATE.BEFORE.WORD :A :B
IF EMPTYP :A [OP "TRUE]
IF EMPTYP :B [OP "FALSE]
IF (ASCII FIRST :A) < (ASCII FIRST :B) [OP "TRUE]
IF NOT EQUALP FIRST :A FIRST :B [OP "FALSE]
OP COLLATE.BEFORE.WORD BF :A BF :B
END
```

COLLATE.BEFORE takes two inputs. Each input is a sentence (in other words, a list of words). It outputs the word TRUE if the first input comes before the second alphabetically.

COLLATE.BEFORE.WORD is similar to COLLATE.BEFORE, except that its two inputs are single words instead of lists of words.

An Example

Here is a list of the greatest songs of all time.

```
MAKE "RECORDS [
        [[SHE LOVES YOU] [BEATLES]]
        [[SHE'S NOT THERE] [ZOMBIES]]
        [[WATERLOO SUNSET] [KINKS]]
        [[FLYING ON THE GROUND IS WRONG]
         [BUFFALO SPRINGFIELD]]
        [[MY GENERATION] [WHO]] ]
```

(To type in a long list like this, you have to use the Logo editor. When you are typing directly to the ? prompt in Atari Logo, there is a limit to how long a line you can type.)

This list contains five items. Each item is itself a list with two members, the title and artist of a record. This is a simple example of a *data structure.* That is, instead of having a list of words or a list of numbers, we have a list of more complicated things, each of which is itself made up of smaller parts.

Suppose we want to sort these songs by title. We can define a COMPARE procedure to do that.

```
TO COMPARE :A :B
OP COLLATE.BEFORE FIRST :A FIRST :B
END
```

The FIRST of each song is a list containing its title, so this version of COMPARE sees which title comes first alphabetically.

```
?SHOW SORT :RECORDS
[[[FLYING ON THE GROUND IS WRONG] [BU->
FFALO SPRINGFIELD]] [[MY GENERATION] ->
[WHO]] [[SHE LOVES YOU] [BEATLES]] [[->
SHE'S NOT THERE] [ZOMBIES]] [[WATERLO->
O SUNSET] [KINKS]]]
?
```

We can make this prettier by using a formatting procedure to print each record on a separate line.

```
TO FORMAT :LIST
IF EMPTYP :LIST [STOP]
PRINT SE "TITLE: FIRST FIRST :LIST
PRINT SE "...ARTIST: LAST FIRST :LIST
FORMAT BF :LIST
END

?FORMAT SORT :RECORDS
TITLE: FLYING ON THE GROUND IS WRONG
...ARTIST: BUFFALO SPRINGFIELD
TITLE: MY GENERATION
...ARTIST: WHO
TITLE: SHE LOVES YOU
...ARTIST: BEATLES
TITLE: SHE'S NOT THERE
...ARTIST: ZOMBIES
TITLE: WATERLOO SUNSET
...ARTIST: KINKS
?
```

Now suppose we want to sort the same list of records, this time by artist. To do this, we replace the COMPARE procedure with one that uses the LAST of each item instead of the FIRST.

```
TO COMPARE :A :B
OP COLLATE.BEFORE LAST :A LAST :B
END
```

The LAST of each song is a list containing the name of the group that performed it.

```
?FORMAT SORT :RECORDS
TITLE: SHE LOVES YOU
...ARTIST: BEATLES
TITLE: FLYING ON THE GROUND IS WRONG
...ARTIST: BUFFALO SPRINGFIELD
TITLE: WATERLOO SUNSET
...ARTIST: KINKS
TITLE: MY GENERATION
...ARTIST: WHO
TITLE: SHE'S NOT THERE
...ARTIST: ZOMBIES
?
```

SUGGESTIONS

If you are interested in learning about other ways to write sorting programs, the standard reference book on this subject is *Sorting and Searching,* volume 3 of *The Art of Computer Programming,* by Donald E. Knuth (Reading, Mass.: Addison-Wesley, 1973).

PROGRAM LISTING

Note: There are three different versions of COMPARE in the write-up. The one here is the first version. COMPARE2 and COMPARE3 are the other two versions and can be substituted for COMPARE in MERGE.

```
TO FIRST.PART :LIST
OP FIRST.N (INT (COUNT :LIST)/2) :LIST
END

TO FIRST.N :NUMBER :LIST
IF :NUMBER=0 [OP :LIST]
OP FIRST.N :NUMBER-1 BL :LIST
END

TO LAST.PART :LIST
OP LAST.N (INT (1+COUNT :LIST)/2) ▸
     :LIST
END

TO LAST.N :NUMBER :LIST
IF :NUMBER=0 [OP :LIST]
OP LAST.N :NUMBER-1 BF :LIST
END

TO MERGE :A :B
IF EMPTYP :A [OP :B]
IF EMPTYP :B [OP :A]
IF COMPARE FIRST :A FIRST :B [OP FPUT ▸
     FIRST :A MERGE BF :A :B]
OP FPUT FIRST :B MERGE :A BF :B
END
```

```
TO COMPARE :A :B
OUTPUT :A < :B
END

TO SORT :A
IF EMPTYP :A [OP []]
IF EMPTYP BF :A [OP :A]
OP MERGE (SORT FIRST.PART :A) (SORT ▸
     LAST.PART :A)
END

TO COLLATE.BEFORE :A :B
IF EMPTYP :A [OP "TRUE]
IF EMPTYP :B [OP "FALSE]
IF COLLATE.BEFORE.WORD FIRST :A FIRST ▸
     :B [OP "TRUE]
IF NOT EQUALP FIRST :A FIRST :B [OP ▸
     "FALSE]
OP COLLATE.BEFORE BF :A BF :B
END

TO COLLATE.BEFORE.WORD :A :B
IF EMPTYP :A [OP "TRUE]
IF EMPTYP :B [OP "FALSE]
IF (ASCII FIRST :A) < (ASCII FIRST :B) ▸
     [OP "TRUE]
IF NOT EQUALP FIRST :A FIRST :B [OP ▸
     "FALSE]
OP COLLATE.BEFORE.WORD BF :A BF :B
END

TO COMPARE2 :A :B
OP COLLATE.BEFORE FIRST :A FIRST :B
END

TO FORMAT :LIST
IF EMPTYP :LIST [STOP]
PRINT SE "TITLE: FIRST FIRST :LIST
PRINT SE "...ARTIST: LAST FIRST :LIST
FORMAT BF :LIST
END

TO COMPARE3 :A :B
OP COLLATE.BEFORE LAST :A LAST :B
END

MAKE "RECORDS [ [[SHE LOVES YOU] ▸
     [BEATLES]] [[SHE'S NOT THERE] ▸
     [ZOMBIES]] [[WATERLOO SUNSET] ▸
     [KINKS]] [[FLYING ON THE GROUND ▸
     IS WRONG] [BUFFALO SPRINGFIELD]] ▸
     [[MY GENERATION] [WHO]] ]
```

Bestline

Bestline is a Logo project that draws the "best-fitting" straight line on a Cartesian graph of some data points. It is a strategy commonly used among scientists to predict the value of some quantity based on another.

An Example: A Scientific Experiment

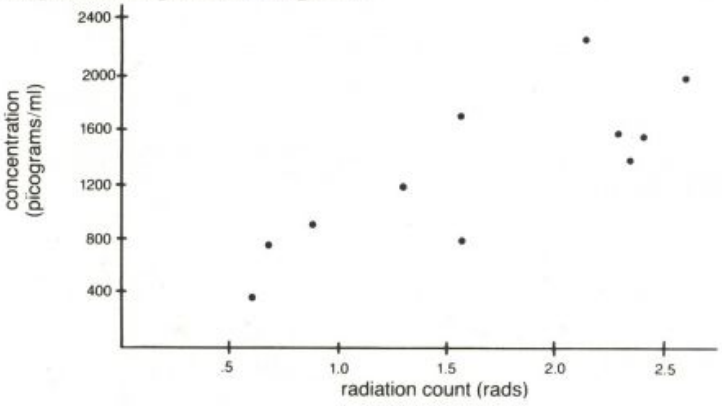
I got the idea for this project while helping a friend interpret data from a laboratory experiment. The purpose of the experiment was to find the concentration of antibodies in each of a large number of test tubes. This is done by adding radioactive iodine to the antibodies. A certain amount of the iodine bonds to the antibody and the rest is removed. The concentration of antibodies can be determined by measuring how much iodine bonded to them. Since the iodine is radioactive, you can run it through a machine that measures how much radiation is emitted by each test tube. In this experiment, the radiation (rad) counts were collected and processed by a computer in my friend's lab. I thought I could write a Logo program that could generate a "best-fit" line for this data and for samples of other data.

By Julie Minsky.

*In statistics, this kind of plot is called a scattergram.

Making a Graph of the Data

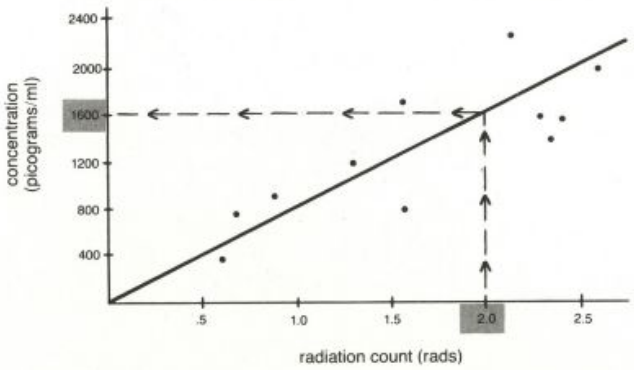
In the antibody experiment, we take samples of known antibody concentrations, measure their radiation counts, and plot them on a graph. For each known concentration, we plot the corresponding radiation count.* When we plot all the points, we might see:



We can use this plot for looking at the data from our samples of known concentrations. How can we use this data to estimate the *unknown* concentrations of our experimental samples?

We know that for this kind of experiment, the radiation count of a sample is proportional to the concentration of antibodies in it. That is, when we double the concentration, the radiation emitted will be doubled. This relationship suggests that the graph of radiation versus concentration is a *line*. We need to find a line on which we can look up an estimate of the concentration of a sample once we have experimentally found its radiation count.

Looking Things Up on a Graph



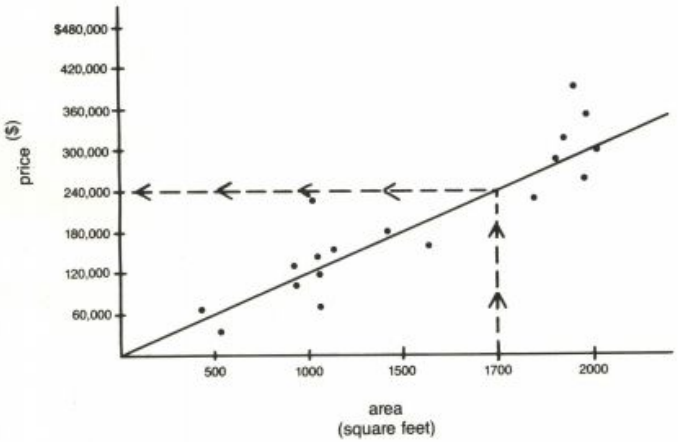
Let's look at the graph above. It shows a regression line plotted for the data points in this experiment.* Once we know how much radiation is emitted,

*The "best-fitting" line through a sample of data points is called a "least squares," or regression, line and is calculated from the data points.

we can estimate the concentration of antibodies present. For example, if the radiation count is 2, then the concentration is estimated to be 1600 picograms/ml.

This line was already calculated for this particular experiment; someone plotted the data points and determined the line. Different samples of data generate different graphs and regression lines.

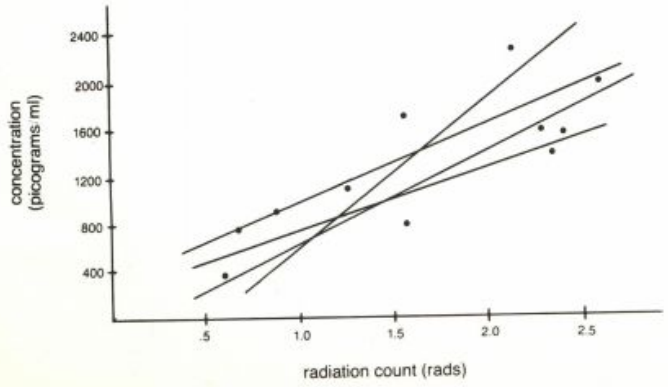
For example, a realtor selling office space might want to know how much to charge for a 1700-square-foot building. Let's say the realtor called other realtors who sold office space in the same community and asked them how much they charged for buildings of different square footages. A helpful graph would be price plotted against square footage. While the realtor might consider other factors in setting the price (for example, property location, condition of the building), she is able to estimate the market price for the office space.



Possible Lines

Many different lines might be drawn through the known sample points. How can we find a line that goes through all the measured points and makes sense for our data?

Since this is a real-world experiment, the sample points don't all lie exactly on a straight line. We could draw lots of lines near these data points. We would like to find the one line that goes as close as possible to *all* of them.



Moving the line closer to some points will increase its distance from others. Some of the lines fit the data so poorly that we wouldn't even consider them. Others would seem to be pretty good fits to the data. We need to find a way of determining the line with the best fit, the line that comes closest to all the points. How can we choose which line is the best fit?

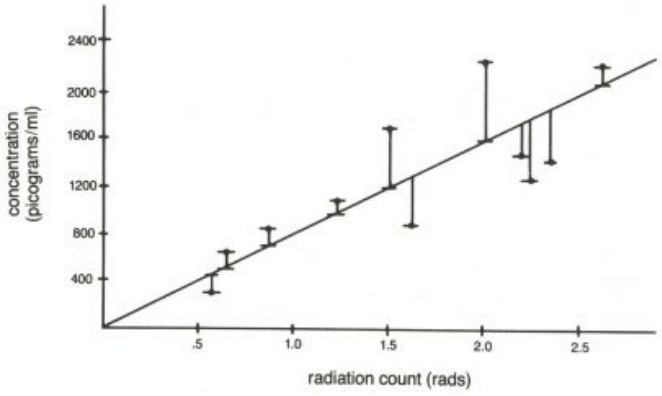
A Technique for Finding the Best Line

There are two things you need to know to plot a line: its slope (m) and its y-intercept (b). The standard equation for a line is

$$y = mx + b$$

Once you know m and b, you can use the equation to find the y coordinate for any x.

A method usually used to find the best-fit line is called "least squares." The least squares line is that which minimizes the sum of the squares of the vertical distances between the line and the data points. It is a neat way of solving the problem when the real-world data points are not exactly on the line (this is usually called minimizing the error). Let's look at the following graph.



Not all the data points fall on the regression line. The amount of "error" of the regression line is the sum of the squares of the vertical distance of each point from the line. If all the data points fall on the best-fit line the error would be 0. This is the ideal; most real-world data do not behave so neatly. The method of least squares is used to calculate a line that *minimizes* the error.

Given a set of points, you can find the best-fit least squares line by solving two equations: one to calculate the slope of the best-fit line and one to calculate its y-intercept.

In our experiment, we have the x and y coordinates of N points. We can use the coordinates and these two equations to find m and b:

$$m = \frac{N\Sigma xy - \Sigma x}{N\Sigma x^2 - (\Sigma x)^2}$$

$$b = \frac{\Sigma y - m\Sigma x}{N}$$

The Greek letter sigma, Σ, is called summation notation; it means that you add a set of numbers. Σx means add up all the x coordinates from the set of points. Σxy means you should multiply the x and y coordinates of each point and add up all the products.

Using the Program

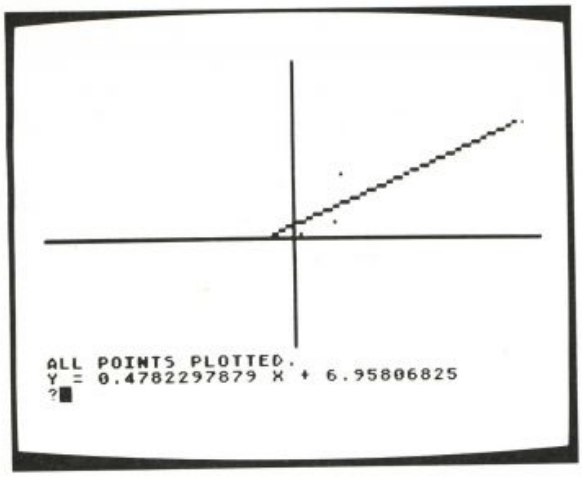
Here is an example of how to use BESTLINE. The text that is boldface is what you type. Say your points are (28, 39), (25, 10), (140, 72), and (5, 2).

```
BESTLINE
PLEASE TYPE YOUR POINTS: X Y X Y ...
28 39 25 10 140 72 5 2
```

Before plotting the line, BESTLINE prints:

```
SLOPE = 0.4756966082
Y-INTERCEPT = 7.203018
Y = 0.4756966882 X + 7.203018
PRESS ANY KEY TO CONTINUE
```

When BESTLINE continues, it plots your points and draws the line that best fits them.



At the bottom of the screen BESTLINE prints:

```
ALL POINTS PLOTTED.
THE X FOR 2 = -10.93768151
THE X FOR 72 = 136.214933
```

To find the y coordinate on the best (fit line for a certain x-100, for example) type:

```
SOLVE.Y 100
THE Y FOR 100 = 54.77267882
```

Similarly, you can use a procedure called SOLVE.X to find the x value for a certain y.

How the Program Works

Overview

BESTLINE is the top-level procedure. It sets up a global variable, POINTLIST, to contain the list of points the user types in. The list of x and y coordinates the user types is converted into a list of lists by PAIRUP. Each sublist contains the x and y coordinates for each point. In our example, :POINTLIST is [[28 39] [25 10] [140 72] [5 2]]. BESTLINE calls LINE.EQUATION to find the slope and the y-intercept of the line that best fits these points. BESTLINE then calls PLOTLINE to plot the points and draw the best-fit line.

```
TO BESTLINE
HT TS CT
PR [PLEASE TYPE YOUR POINTS: X Y X Y ...]
MAKE "POINTLIST PAIRUP RL
LINE.EQUATION :POINTLIST
PR [PRESS ANY KEY TO CONTINUE]
IGNORE RC
WINDOW
PLOTLINE :POINTLIST
END

TO IGNORE :THING
END
```

LINE.EQUATION creates two global variables, M and B. :M is the slope of the line and is computed by LEAST.SQUARES.SLOPE. :B is the y-intercept and is computed by YINTERCEPT.

```
TO LINE.EQUATION :POINTLIST
MAKE "M LEAST.SQUARES.SLOPE :POINTLIST
PR SE [SLOPE IS] :M
MAKE "B YINTERCEPT :POINTLIST
PR SE [Y\-INTERCEPT =] :B
PR ( SE [Y =] :M [X +] :B )
END

TO LEAST.SQUARES.SLOPE :POINTS
MAKE "SUMX BIGE :POINTS "JUSTX
OP ( ( COUNT :POINTS ) * ( BIGE :POINTS "XTIMESY )
      - ( :SUMX * BIGE :POINTS "JUSTY ) )
     / ( ( COUNT :POINTS ) * ( BIGE :POINTS "XSQUARED )
       - ( :SUMX * :SUMX ) )
END
```

```
TO YINTERCEPT :POINTS
OP ( ( ( BIGE :POINTS "JUSTY )
        - ( :M * BIGE :POINTS "JUSTX ) ) / COUNT :POINTS )
END
```

Both LEAST.SQUARES.SLOPE and YINTERCEPT rely on a collection of procedures used by BIGE.

BIGE

BIGE takes two inputs, a list of points and the name of another procedure. It sums the result of applying that procedure to each point in the list.* (This procedure is called BIGE, pronounced "big-ee," because the Greek letter Σ, used as the summation symbol, looks like an upper-case "E." For example, if you type BIGE :POINTS "JUSTX, JUSTX will extract just the x coordinate from each point in the list and BIGE will end up adding up just the x's! BIGE :POINTS "XTIMESY adds up the products of the x and y coordinates for each point. The procedures used with BIGE in the formulas are JUSTX, JUSTY, XTIMESY, and XSQUARED.

```
TO BIGE :LIST :PROC
IF EMPTYP :LIST [OP 0]
OP ( SUM ( RUN LIST :PROC FIRST :LIST )
                BIGE BF :LIST :PROC )
END

TO JUSTX :POINT
OP FIRST :POINT
END

TO JUSTY :POINT
OP FIRST BF :POINT
END

TO XTIMESY :POINT
OP ( JUSTX :POINT ) * ( JUSTY :POINT )
END

TO XSQUARED :POINT
OP ( JUSTX :POINT ) * ( JUSTX :POINT )
END
```

Graphing

After the equation of the best-fit line is determined, PLOTLINE plots your points and the line. PLOTLINE first uses PLOT.POINTS to draw your points.

*BIGE is a mapping procedure. See Brian Harvey's Map project (p.322) for more about mapping.

```
TO PLOTLINE :LIST
SS
PLOT.AXES
PLOT.POINTS :LIST
WAIT 60
RANGE :LIST
IF :M = 0 [PLOT.HORIZ STOP]
PU
SETPOS LIST XVALUE :MINY :MINY
PD
SETPOS LIST XVALUE :MAXY :MAXY
PRINT (SE [Y = ] :M [X + ] :B)
END

TO PLOT.AXES
MAKE "PN PN
IF :PN = 2 [SETPN 0] [SETPN PN + 1]
SETPC PN 0
PU SETPOS [-150 0]
PD SETPOS [150 0]
PU SETPOS [0 -110]
PD SETPOS [0 110]
PU SETPN :PN HOME
END

TO PLOT.POINTS :LIST
IF EMPTYP :LIST [PR [ALL POINTS PLOTTED.] STOP]
PU
SETPOS FIRST :LIST
PD FD 0
PLOT.POINTS BF :LIST
END

TO SOLVE.Y :X
PR ( SE [THE Y FOR] :X "= ( :M * :X + :B ) )
END

TO SOLVE.X :Y
IF :M = 0 [PRINT [NO SOLUTION, M=0] STOP]
PRINT (SE [THE X FOR] :Y "= XVALUE :Y)
END

TO XVALUE :Y
OP (:Y - :B) / :M
END
```

PLOTLINE then draws the best-fit line. If the slope (:M) is 0, then PLOT.HORIZ draws the line. The procedure RANGE finds the smallest and largest x and y coordinates for your set of points. SOLVE.X finds the best-fit line's x coordinate for the minimum y and maximum y computed by RANGE. These are the procedures for finding and plotting the endpoints of the best-fit line.

```
TO RANGE :PLIST
MAKE "MINX LEAST.NUM XLIST :PLIST
MAKE "MINY LEAST.NUM YLIST :PLIST
MAKE "MAXX GREATEST.NUM XLIST :PLIST
MAKE "MAXY GREATEST.NUM YLIST :PLIST
END

TO LEAST.NUM :NUMS
IF EMPTYP BF :NUMS [OP FIRST :NUMS]
IF ( FIRST :NUMS ) < ( FIRST BF :NUMS )
               [OP LEAST.NUM SE BF BF :NUMS FIRST :NUMS]
OP LEAST.NUM BF :NUMS
END

TO GREATEST.NUM :NUMS
IF EMPTYP BF :NUMS [OP FIRST :NUMS]
IF ( FIRST BF :NUMS ) > FIRST :NUMS
                [OP GREATEST.NUM BF :NUMS]
OP GREATEST.NUM SE BF BF :NUMS FIRST :NUMS
END

TO XLIST :POINTLIST
IF EMPTYP :POINTLIST [OP []]
OP FPUT JUSTX FIRST :POINTLIST XLIST BF :POINTLIST
END

TO YLIST :POINTLIST
IF EMPTYP :POINTLIST [OP []]
OP FPUT JUSTY FIRST :POINTLIST YLIST BF :POINTLIST
END
```

PLOT.HORIZ is used in the special case when the slope of the line is 0.

```
TO PLOT.HORIZ
PU SETPOS LIST :MINX :B
PD SETPOS LIST :MAXX :B
END
```

PAIRUP and PAIRS are used by BESTLINE to convert a list of coordinates typed by the user into a list of points that the program can use.

```
TO PAIRUP :LIST
IF 1 = REMAINDER COUNT :LIST 2 [MAKE "LIST BL :LIST]
OP PAIRS :LIST
END

TO PAIRS :LIST
IF EMPTYP :LIST [OP []]
OP FPUT SE FIRST :LIST FIRST BF :LIST PAIRS BF BF :LIST
END
```

PROGRAM LISTING

```
TO BESTLINE
HT TS CT
PR [PLEASE TYPE YOUR POINTS: X Y X Y ►
    ...]
MAKE "POINTLIST PAIRUP RL
LINE.EQUATION :POINTLIST
PR [PRESS ANY KEY TO CONTINUE]
IGNORE RC
WINDOW
PLOTLINE :POINTLIST
END

TO IGNORE :THING
END

TO LINE.EQUATION :POINTLIST
MAKE "M LEAST.SQUARES.SLOPE :POINTLIST
PR SE [SLOPE IS] :M
MAKE "B YINTERCEPT :POINTLIST
PR SE [Y\-INTERCEPT =] :B
PR ( SE [Y =] :M [X +] :B )
END

TO LEAST.SQUARES.SLOPE :POINTS
MAKE "SUMX BIGE :POINTS "JUSTX
OP ( ( COUNT :POINTS ) * ( BIGE ►
     :POINTS "XTIMESY ) - ( :SUMX * ►
     BIGE :POINTS "JUSTY ) ) / ( ( ►
     COUNT :POINTS ) * ( BIGE :POINTS ►
     "XSQUARED ) - ( :SUMX * :SUMX ) )
END

TO YINTERCEPT :POINTS
OP ( ( ( BIGE :POINTS "JUSTY ) - ( :M ►
     * BIGE :POINTS "JUSTX ) ) / COUNT ►
     :POINTS )
END

TO BIGE :LIST :PROC
IF EMPTYP :LIST [OP 0]
OP ( SUM ( RUN LIST :PROC FIRST :LIST ►
     ) BIGE BF :LIST :PROC )
END

TO JUSTX :POINT
OP FIRST :POINT
END

TO JUSTY :POINT
OP FIRST BF :POINT
END

TO XTIMESY :POINT
OP ( JUSTX :POINT ) * ( JUSTY :POINT )
END

TO XSQUARED :POINT
OP ( JUSTX :POINT ) * ( JUSTX :POINT )
END

TO PLOTLINE :LIST
SS
PLOT.AXES
PLOT.POINTS :LIST
WAIT 60
RANGE :LIST
IF :M = 0 [PLOT.HORIZ STOP]
PU
SETPOS LIST XVALUE :MINY :MINY
PD
SETPOS LIST XVALUE :MAXY :MAXY
PRINT (SE [Y = ] :M [X + ] :B)
END

TO PLOT.AXES
MAKE "PN PN
IF :PN = 2 [SETPN 0] [SETPN PN + 1]
SETPC PN 0
PU SETPOS [-150 0]
PD SETPOS [150 0]
PU SETPOS [0 -110]
PD SETPOS [0 110]
PU SETPN :PN HOME
END

TO PLOT.POINTS :LIST
IF EMPTYP :LIST [PR [ALL POINTS ►
     PLOTTED.] STOP]
PU
SETPOS FIRST :LIST
PD FD 0
PLOT.POINTS BF :LIST
END

TO SOLVE.Y :X
PR ( SE [THE Y FOR] :X "= ( :M * :X + ►
     :B ) )
END

TO SOLVE.X :Y
IF :M = 0 [PRINT [NO SOLUTION, M=0] ►
     STOP]
PRINT (SE [THE X FOR] :Y "= XVALUE :Y)
END
```

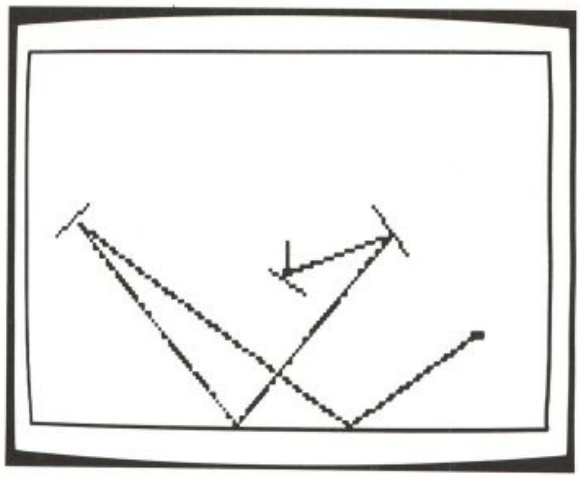
```
TO XVALUE :Y
OP (:Y - :B) / :M
END

TO RANGE :PLIST
MAKE "MINX LEAST.NUM XLIST :PLIST
MAKE "MINY LEAST.NUM YLIST :PLIST
MAKE "MAXX GREATEST.NUM XLIST :PLIST
MAKE "MAXY GREATEST.NUM YLIST :PLIST
END

TO LEAST.NUM :NUMS
IF EMPTYP BF :NUMS [OP FIRST :NUMS]
IF ( FIRST :NUMS ) < ( FIRST BF :NUMS ►
     ) [OP LEAST.NUM SE BF BF :NUMS ►
     FIRST :NUMS]
OP LEAST.NUM BF :NUMS
END

TO GREATEST.NUM :NUMS
IF EMPTYP BF :NUMS [OP FIRST :NUMS]
IF ( FIRST BF :NUMS ) > FIRST :NUMS ►
     [OP GREATEST.NUM BF :NUMS]
OP GREATEST.NUM SE BF BF :NUMS FIRST ►
     :NUMS
END

TO XLIST :POINTLIST
IF EMPTYP :POINTLIST [OP []]
OP FPUT JUSTX FIRST :POINTLIST XLIST ►
     BF :POINTLIST
END

TO YLIST :POINTLIST
IF EMPTYP :POINTLIST [OP []]
OP FPUT JUSTY FIRST :POINTLIST YLIST ►
     BF :POINTLIST
END

TO PLOT.HORIZ
PU SETPOS LIST :MINX :B
PD SETPOS LIST :MAXX :B
END

TO PAIRUP :LIST
IF 1 = REMAINDER COUNT :LIST 2 [MAKE ►
     "LIST BL :LIST]
OP PAIRS :LIST
END

TO PAIRS :LIST
IF EMPTYP :LIST [OP []]
OP FPUT SE FIRST :LIST FIRST BF :LIST ►
     PAIRS BF BF :LIST
END
```

Lines and Mirrors

This program was designed to simulate a beam of light bouncing off mirrors or a ball bouncing off walls. The user enters the coordinates of endpoints of lines. The program then draws the lines and starts the turtle going in a random direction. When the turtle hits one of those lines, it will bounce off at the same angle at which it came in. The turtle draws its path as it goes. You can think of the turtle's path as a beam of light and the lines as mirrors.

In this write-up there are three main sections: first, how the program calculates the angle at which the turtle should bounce after hitting a line; second, how the information about the lines is remembered; and third, the detailed structure of the program.

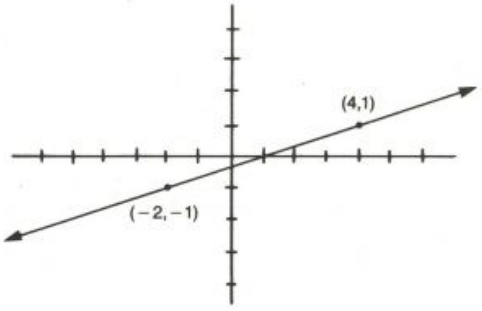
By Toby Mintz.



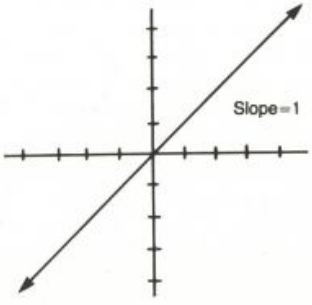
Bouncing Off a Line

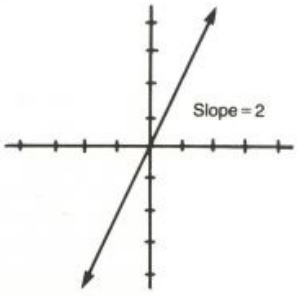
What Is a Line?

You probably know that two points determine a line, as in the following illustration.

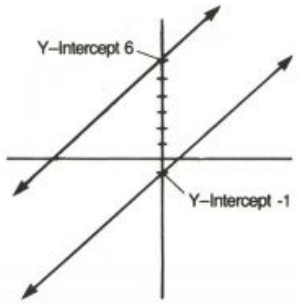


Another way to determine a line is by its slope (m) and y-intercept (b). The slope is the steepness of the line. In the following figure the line on the right rises twice as fast as the line on the left.





There may be many different lines with the same slope. A way to distinguish these lines is by their y-intercept. The y-intercept is the point at which the line crosses the y axis.



Calculating the Turning Angle

For our purposes we need a representation that tells us *where* the line is and what its *orientation* is. (The orientation is particularly important because the problem we are trying to solve is about directions of motion.) These aspects of lines are reminiscent of the major components of the state of a turtle: position and heading. This suggests that the best way to represent the orientation of a line is by the heading that a turtle would take to draw it.

It is important for us to know the heading of a line in order to figure out how the turtle should bounce off it. The angle at which the turtle comes in (angle of incidence) should be the same as the angle at which it bounces out (angle of reflection).

The angle of incidence is the amount the turtle must turn to get from its initial heading to the heading of the line.

So we get

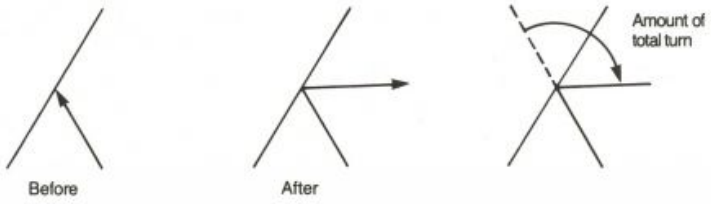
(*line's heading*) − (*turtle's heading*)

as the angle of incidence. The angle of reflection should also be

(*line's heading*) − (*turtle's heading*)

The total amount through which the turtle turns is therefore

$2 \times [(\textit{line's heading}) - (\textit{turtle's heading})]$



Figuring Out the Heading

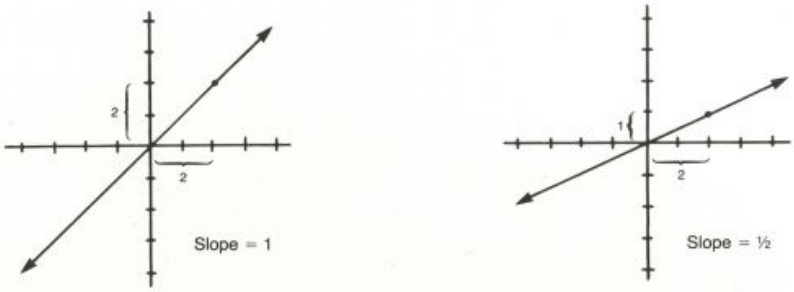
In fact, we are not given the heading or the position of a line. All we are given are its two endpoints. With the two endpoints we can figure out the slope. Then we can use the ARCTAN procedure, described in the Towards and Arctan project, to figure out the heading. The procedure to figure out the heading is as follows.

```
TO FIGH :LINE
IF ( DX :LINE ) = 0 [OP 0]
OP 90 - ARCTAN SLOPE :LINE
END
```

(It is 90-ARCTAN because Logo headings are clockwise from north, not counterclockwise from east as in algebra.)

Figuring Out the Slope

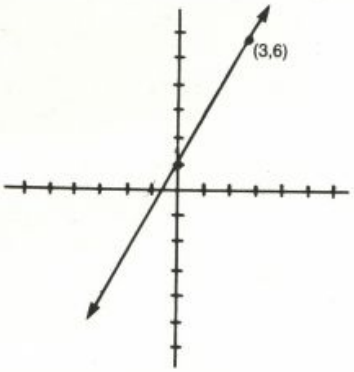
The slope is the difference between the y coordinates of any two points on the line divided by the difference between the x coordinates of those points ($\Delta y / \Delta x$, where Δ stands for "difference in").



Traditionally, a line is represented by the equation $y = mx + b$. Given m and b, we can tell whether a particular point is on a particular line. For example, if

$$\text{YCOR} = (m \times \text{XCOR}) + b$$

then we know that the turtle's position is on the line.



m of line = ⅔
b of line = 1
YCOR = 6
XCOR = 3

(⅔ * 3) + 1 = 6
So the point is on the line.

We use $\Delta y / \Delta x$ to figure out the slope. But if the line is vertical (the two x coordinates are the same), then the slope is infinite, so the procedure to figure out the slope has to treat that case in a special way. The procedures to figure out the slope (m) and y-intercept (b) of the line are as follows.

```
TO FIGM :LINE
IF ( DX :LINE ) = 0 [OP []] [OP SLOPE :LINE]
END

TO SLOPE :LINE
OP ( DY :LINE ) / DX :LINE
END

TO DY :LINE
OP ( LAST POINT1 :LINE ) - ( LAST POINT2 :LINE )
END

TO DX :LINE
OP ( FIRST POINT1 :LINE ) - ( FIRST POINT2 :LINE )
END

TO FIGB :LINE
IF (M :LINE) = [] [OP []]
OP (LAST POINT1 :LINE) - ((M :LINE) * (FIRST POINT1 :LINE))
END
```

Information We Need About a Line

What information does this program need about a line?

It needs the heading for use in calculating the turning angle when the turtle hits the line.

It uses slope and y-intercept to figure out if a position is on a line, with the equation $y = mx + b$.

It also needs the endpoints. Why? So far we have been talking about *lines,* which are infinitely long. Really the program has to deal with line *segments,* which have two endpoints.

So we finally need five pieces of information to represent a line segment: two endpoints, heading, slope, and y-intercept.

Storing the Lines

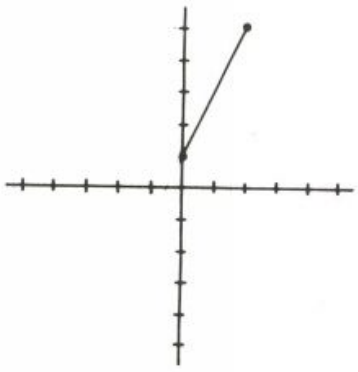
What Each Line Looks Like

The only thing that the user gives the program is the endpoints of lines. From this the slope, heading, and y-intercept are figured out. The program has to have a way of storing all this information in some kind of organized structure. It stores the five pieces of information about the line in a list of the following format:

```
[[x1 y1] [x2 y2] slope heading y-intercept]
```

Here is a sample line and the list that represents it.

```
[[0 1] [2 5] 2 26.5651 1]
```



Retrieving Information About Lines

Throughout the program we need to recall information about lines. We could do it in a messy way. For example, to retrieve the slope of a line we could say

```
FIRST BF BF :LINE
```

The program would get very ugly and confusing if we used that approach. A much clearer and neater way is to have a procedure that extracts one piece of information about a line. So we could say `M :LINE` to retrieve the slope, or `POINT2 :LINE` to retrieve the coordinates of the second endpoint. By using these procedures, other parts of the program do not have to know the detailed structure of a line list. The procedures also make

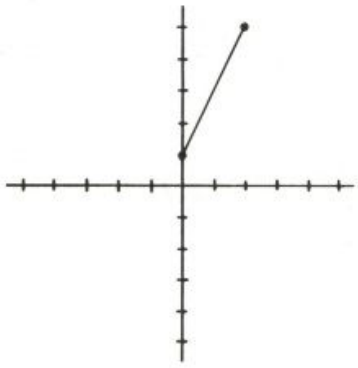
the program much easier to read and understand. Here are the information retrieving procedures.

```
TO POINT1 :LINE
OP ITEM 1 :LINE
END

TO POINT2 :LINE
OP ITEM 2 :LINE
END

TO M :LINE
OP ITEM 3 :LINE
END

TO D :LINE
OP ITEM 4 :LINE
END

TO B :LINE
OP ITEM 5 :LINE
END

TO ITEM :INUM :LIST
IF :INUM = 1 [OP FIRST :LIST]
OP ITEM :INUM - 1 BUTFIRST :LIST
END
```

The List of Lines

Since the program has to keep track of a lot of lines, it stores them all in a list called LINES. The elements of :LINES are themselves lists, each representing a line. For example, the border lines and the line shown earlier would be represented as follows:

```
[ [[-158 -119] [-158 120] [] 0 []]
  [[161 -119] [161 120] [] 0 []]
  [[-158 120] [161 120] 0 90 120]
  [[-158 -119] [161 -119] 0 90 -119]
  [[0 1] [2 5] 2 26.5651 1] ]
```



Program Structure

The top-level procedure is BOUNCE. It has four tasks. First it creates the list of lines. Then it sets up the initial position and shape of the turtle. The third task is to draw the lines in the list. Finally it starts the turtle moving and prepares to turn the turtle when it bounces off a wall. There is a subprocedure for each of these tasks.

```
TO BOUNCE
LEARN.LINES
SETUP.GRAPHICS
DRAW.LINES
START.TURTLE
END
```

Creating the List of Lines

When the program starts up, LEARN.LINES creates the list of lines. It calls INFO, to remember the lines which the user enters. It also calls BORDER, which remembers the border lines.

```
TO LEARN.LINES
MAKE "LINES []
INFO
BORDER
END
```

INFO lets the user enter lines. It calls GETLINE to get each line and calls REMEMBER to add each line to the list :LINES.

```
TO INFO
REMEMBER GETLINE
PRINT [ANOTHER LINE?]
IF EQUALP RL [YES] [INFO]
END
```

GETLINE asks the user to type in the endpoints of a line segment. It calls FIGLINE to calculate the other information about the line. The output from GETLINE is the list representing the line.

```
TO GETLINE
TYPE [X AND Y OF FIRST POINT?]
MAKE "FP RL
TYPE [X AND Y OF SECOND POINT?]
MAKE "SP RL
MAKE "LINE LIST :FP :SP
OP FIGLINE :LINE
END
```

FIGLINE takes the endpoints of a line segment as its input. It calls FIGM, FIGH, and FIGB to compute the slope, heading, and y-intercept of the line.

```
TO FIGLINE :LINE
MAKE "LINE LPUT FIGM :LINE :LINE
MAKE "LINE LPUT FIGH :LINE :LINE
MAKE "LINE LPUT FIGB :LINE :LINE
OP :LINE
END
```

REMEMBER adds a line to the list of lines (:LINES).

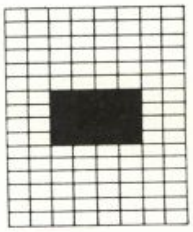
```
TO REMEMBER :LINE
MAKE "LINES FPUT :LINE :LINES
END
```

BORDER remembers the four lines making up the border of the screen.

```
TO BORDER
REMEMBER FIGLINE [[-158 -119] [-158 120]]
REMEMBER FIGLINE [[161 -119] [161 120]]
REMEMBER FIGLINE [[-158 120] [161 120]]
REMEMBER FIGLINE [[-158 -119] [161 -119]]
END
```

Setting Up Graphics

SETUP.GRAPHICS selects turtle 0 and changes its shape. We don't use the normal turtle shape because later on we will need to know the precise position of the turtle. The normal turtle shape is big enough that its edges are at a very different position from the position of the center, which XCOR and YCOR output. Instead, we use a small square dot shape.



```
TO SETUP.GRAPHICS
ASK [1 2 3] [HT]
TELL 0
PUTSH 1 :SHAPE1
SETSH 1
CS
ST
FS
END

MAKE "SHAPE1 [0 0 0 0 0 0 60 60 60 60 0 0 0 0 0 0]
```

Drawing the Lines

DRAW.LINES draws the lines in :LINES on the screen. The lines are drawn with pen 1. Later, when the turtle is moving, its trajectory is drawn with pen 0. Using a different pen for the walls allows the demon to notice collisions with the walls and not notice collisions between the turtle and its own earlier path.

```
TO DRAW.LINES
SETPN 1
DRAW :LINES
END
```

```
TO DRAW :LIST
IF EMPTYP :LIST [STOP]
PU
SETPOS POINT1 FIRST :LIST
PD
SETPOS POINT2 FIRST :LIST
DRAW BUTFIRST :LIST
END
```

Starting the Turtle

START.TURTLE positions the turtle in the center of the screen, points it in a randomly chosen direction, and starts it moving. It also creates the demon that waits for collisions with lines. Finally, START.TURTLE calls LOOP, which is explained next.

```
TO START.TURTLE
SETPN 0
PU
SETPOS [0 0]
PD
SETH RANDOM 360
SETSP 20
WHEN OVER 0 1 [SETSP 0 WHEN OVER 0 1 []]
LOOP
END
```

Knowing When a Line Is Hit

While the turtle is moving, LOOP continually checks if it has hit a line. LOOP knows the turtle has hit a line when its speed becomes zero.

```
TO LOOP
IF SPEED = 0 [NEWHEAD]
LOOP
END
```

How does the speed become zero? There is a demon, created by START.TURTLE, whose instructions include SETSP 0.

```
WHEN OVER 0 1 [SETSP 0 WHEN OVER 0 1 []]
```

Setting the turtle's speed to zero is a convenient way for the demon to signal to LOOP that the turtle has hit a line. We could have changed something else as the signal, but we had to stop the turtle anyway. Otherwise the turtle would go through the line. Using the speed as the signal solves two problems at once.

When the speed is zero, LOOP calls NEWHEAD to figure out which line was hit and how much the turtle should turn.

Which Line Is Being Hit?

When a line is hit, NEWHEAD calls SEARCH to go through the list of lines, finding the one that was hit. Then NEWHEAD uses that line as the input to FIGTURN, which figures out how much the turtle should turn. Finally, NEWHEAD restarts the turtle and the demon.

```
TO NEWHEAD
MAKE "L SEARCH :LINES
IF NOT EMPTYP :L [RIGHT FIGTURN :L]
SETSP 20
WHEN OVER 0 1 [SETSP 0 WHEN OVER 0 1 []]
END
```

SEARCH goes through the list of lines, looking for the one the turtle hit. It calls the predicate CHECK for each line in the list. If CHECK outputs TRUE, SEARCH outputs the line that has been found.

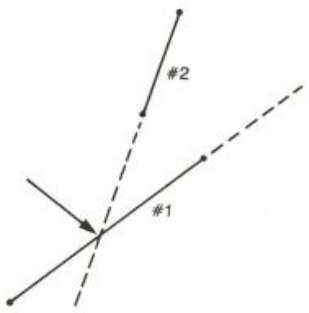
```
TO SEARCH :LINES
IF EMPTYP :LINES [OP []]
IF CHECK FIRST :LINES [OP FIRST :LINES]
OP SEARCH BF :LINES
END
```

How to Check a Line

The straightforward way to check if the turtle hit a certain line is to use the equation $y = mx + b$, substituting the XCOR and YCOR of the turtle for x and y. If the equation holds true, then the turtle hit that line.

```
TO CHECK1 :LINE
OP YCOR = SOLVE :LINE XCOR
END

TO SOLVE :LINE :X
OP ( ( M :LINE ) * :X ) + B :LINE
END
```

There are three problems. The first problem has to do with the fact that we are using line *segments* and not lines. Look at the following picture.



The turtle has hit line segment #1. The turtle's coordinates satisfy the equation $y = mx + b$ for the line containing that segment. The turtle has *not* hit line segment #2. However, the line containing that segment happens to pass through the turtle's position. Therefore, the equation $y = mx + b$ for *that* line is also satisfied. CHECK must also check to see if the turtle is *between* the two endpoints of the line segment.

```
TO CHECK2 :LINE
IF NOT BETWEEN XCOR (FIRST POINT1 :LINE)
         (FIRST POINT2 :LINE) [OP FALSE]
OP YCOR = SOLVE :LINE XCOR
END

TO BETWEEN :THING :LIMIT1 :LIMIT2
IF :LIMIT1 > :LIMIT2 [OP AND ( GE :LIMIT1 :THING )
        ( GE :THING :LIMIT2 )]
        [OP AND ( GE :LIMIT2 :THING ) ( GE :THING :LIMIT1 )]
END

TO GE :A :B
OP NOT :A < :B
END
```

The second problem is that the turtle isn't actually one point; it is slightly bigger. This means that when the edge of the turtle hits a line, the turtle's coordinates won't match up exactly with the line's, because the turtle's coordinates are those of its center, not those of its edge. In this program the turtle has a square shape. If its YCOR is within seven units of the line's y value for the turtle's XCOR, we consider the turtle to be on the line. The number seven worked out best experimentally. Larger numbers lead to false hits. Smaller ones lead to not finding any hits at all. Our updated version of CHECK looks like this.

```
TO CHECK3 :LINE
IF NOT BETWEEN XCOR (FIRST POINT1 :LINE)
         (FIRST POINT2 :LINE) [OP FALSE]
OP ( ABS (SOLVE :LINE XCOR) - YCOR ) < 7
END

TO ABS :NUMBER
OP IF :NUMBER < 0 [- :NUMBER] [:NUMBER]
END
```

The last problem is that vertical lines have an infinite slope (Δx is zero in $\Delta y / \Delta x$). SOLVE won't work for a vertical line, because it needs a numeric slope. Also, we can't check to see if the turtle is between the x values of the endpoints of the line, because the x values are the same; there is no "be-

tween."* So, for a vertical segment, we have to see if the YCOR of the turtle is between the y values of the two endpoints.

Instead of calling SOLVE, we see if the XCOR of the turtle is within seven units of the x value of one of the endpoints. Our final version of CHECK looks like this.

```
TO CHECK :LINE
OP IF EMPTYP M :LINE [CHECK.VERT :LINE] [CHECK.SLANT :LINE]
END

TO CHECK.SLANT :LINE
IF NOT BETWEEN XCOR ( FIRST POINT1 :LINE )
        ( FIRST POINT2 :LINE ) [OP "FALSE]
OP ( ABS ( SOLVE :LINE XCOR ) - YCOR ) < 7
END

TO CHECK.VERT :LINE
IF NOT BETWEEN YCOR ( LAST POINT1 :LINE )
        ( LAST POINT2 :LINE ) [OP "FALSE]
OP ( ABS XCOR - FIRST POINT1 :LINE ) < 7
END
```

Turning the Turtle

Once NEWHEAD knows which line was hit, it can figure out how much to turn the turtle. The turtle's original heading is provided by the primitive procedure HEADING. The heading of the line is provided by

```
D :LINE
```

Recall that the angle through which the turtle should turn is therefore

```
2 * ( ( D :LINE ) - HEADING )
```

NEWHEAD calls FIGTURN to figure out how much the turtle should turn:

```
TO FIGTURN :L
OP 2 * ( ( D :L ) - HEADING )
END
```

*This is not exactly true. If the turtle's XCOR is *equal* to the x values of the line, then in a sense the turtle is between the x values. This doesn't mean the turtle is on the line segment. The problem is that for a vertical line segment, the turtle's YCOR might not be between the y values of the endpoints of the line segment, even though the XCOR is in the right range. For diagonal lines, if one coordinate is in range, the other must also be in range.

PROGRAM LISTING

```
TO FIGH :LINE
IF ( DX :LINE ) = 0 [OP 0]
OP 90 - ARCTAN SLOPE :LINE
END

TO FIGM :LINE
IF ( DX :LINE ) = 0 [OP []] [OP SLOPE ▸
     :LINE]
END

TO SLOPE :LINE
OP ( DY :LINE ) / DX :LINE
END

TO DY :LINE
OP ( LAST POINT1 :LINE ) - ( LAST ▸
     POINT2 :LINE )
END

TO DX :LINE
OP ( FIRST POINT1 :LINE ) - ( FIRST ▸
     POINT2 :LINE )
END

TO FIGB :LINE
IF (M :LINE) = [] [OP []]
OP (LAST POINT1 :LINE) - ( ( M :LINE) ▸
     * ( FIRST POINT1 :LINE) )
END

TO POINT1 :LINE
OP ITEM 1 :LINE
END

TO POINT2 :LINE
OP ITEM 2 :LINE
END

TO M :LINE
OP ITEM 3 :LINE
END

TO D :LINE
OP ITEM 4 :LINE
END

TO B :LINE
OP ITEM 5 :LINE
END

TO ITEM :INUM :LIST
IF :INUM = 1 [OP FIRST :LIST]
OP ITEM :INUM - 1 BUTFIRST :LIST
END

TO BOUNCE
LEARN.LINES
SETUP.GRAPHICS
DRAW.LINES
START.TURTLE
END

TO LEARN.LINES
MAKE "LINES []
INFO
BORDER
END

TO INFO
REMEMBER GETLINE
PRINT [ANOTHER LINE?]
IF EQUALP RL [YES] [INFO]
END

TO GETLINE
TYPE [X AND Y OF FIRST POINT?]
MAKE "FP RL
TYPE [X AND Y OF SECOND POINT?]
MAKE "SP RL
MAKE "LINE LIST :FP :SP
OP FIGLINE :LINE
END

TO FIGLINE :LINE
MAKE "LINE LPUT FIGM :LINE :LINE
MAKE "LINE LPUT FIGH :LINE :LINE
MAKE "LINE LPUT FIGB :LINE :LINE
OP :LINE
END

TO REMEMBER :LINE
MAKE "LINES FPUT :LINE :LINES
END

TO BORDER
REMEMBER FIGLINE [[-158 -119] [-158 ▸
     120]]
REMEMBER FIGLINE [[161 -119] [161 ▸
     120]]
REMEMBER FIGLINE [[-158 120] [161 ▸
     120]]
REMEMBER FIGLINE [[-158 -119] [161 ▸
     -119]]
END
```

```
TO SETUP.GRAPHICS
ASK [1 2 3] [HT]
TELL 0
PUTSH 1 :SHAPE1
SETSH 1
CS
ST
FS
END

TO DRAW.LINES
SETPN 1
DRAW :LINES
END

TO DRAW :LIST
IF EMPTYP :LIST [STOP]
PU
SETPOS POINT1 FIRST :LIST
PD
SETPOS POINT2 FIRST :LIST
DRAW BUTFIRST :LIST
END

TO START.TURTLE
SETPN 0
PU
SETPOS [0 0]
PD
SETH RANDOM 360
SETSP 20
WHEN OVER 0.1 [SETSP 0 WHEN OVER 0 1 ▸
     []]
LOOP
END

TO LOOP
IF SPEED = 0 [NEWHEAD]
LOOP
END

TO NEWHEAD
MAKE "L SEARCH :LINES
IF NOT EMPTYP :L [RIGHT FIGTURN :L]
SETSP 20
WHEN OVER 0 1 [SETSP 0 WHEN OVER 0 1 ▸
     []]
END

TO SEARCH :LINES
IF EMPTYP :LINES [OP []]
IF CHECK FIRST :LINES [OP FIRST ▸
     :LINES]
OP SEARCH BF :LINES
END

TO SOLVE :LINE :X
OP ( ( M :LINE ) * :X ) + B :LINE
END

TO BETWEEN :THING :LIMIT1 :LIMIT2
IF :LIMIT1 > :LIMIT2 [OP AND ( GE ▸
     :LIMIT1 :THING ) ( GE :THING ▸
     :LIMIT2 )] [OP AND ( GE :LIMIT2 ▸
     :THING ) ( GE :THING :LIMIT1 )]
END

TO GE :A :B
OP NOT :A < :B
END

TO ABS :NUMBER
OP IF :NUMBER < 0 [- :NUMBER] ▸
     [:NUMBER]
END

TO CHECK :LINE
OP IF EMPTYP M :LINE [CHECK.VERT ▸
     :LINE] [CHECK.SLANT :LINE]
END

TO CHECK.SLANT :LINE
IF NOT BETWEEN XCOR ( FIRST POINT1 ▸
     :LINE ) ( FIRST POINT2 :LINE ) ▸
     [OP "FALSE]
OP ( ABS ( SOLVE :LINE XCOR ) - YCOR ) ▸
     < 7
END

TO CHECK.VERT :LINE
IF NOT BETWEEN YCOR ( LAST POINT1 ▸
     :LINE ) ( LAST POINT2 :LINE ) [OP ▸
     "FALSE]
OP ( ABS XCOR - FIRST POINT1 :LINE ) < ▸
     7
END

TO FIGTURN :L
OP 2 * ( ( D :L ) - HEADING )
END

TO ARCTAN :X
OP 57.3*ARCTAN.RAD :X
END

TO ARCTAN.RAD :X
IF :X>1 [OP 1.571-ARCTAN.RAD (1/:X)]
OP :X/(1+0.28*:X*:X)
END

MAKE "SHAPE1 [0 0 0 0 0 0 60 60 60 60 ▸
     0 0 0 0 0 0]
```