Leone Learning Systems, Inc.
Wonder. Create. Grow.

Leone Learning Systems, Inc.
237 Custer Ave
Evanston, IL 60202
Emal tj@leonelearningsystems.com

Phone 847 951 0127
Fax 847 733 8812

Logo

An introduction

TJ Leone

October 2004

Introduction

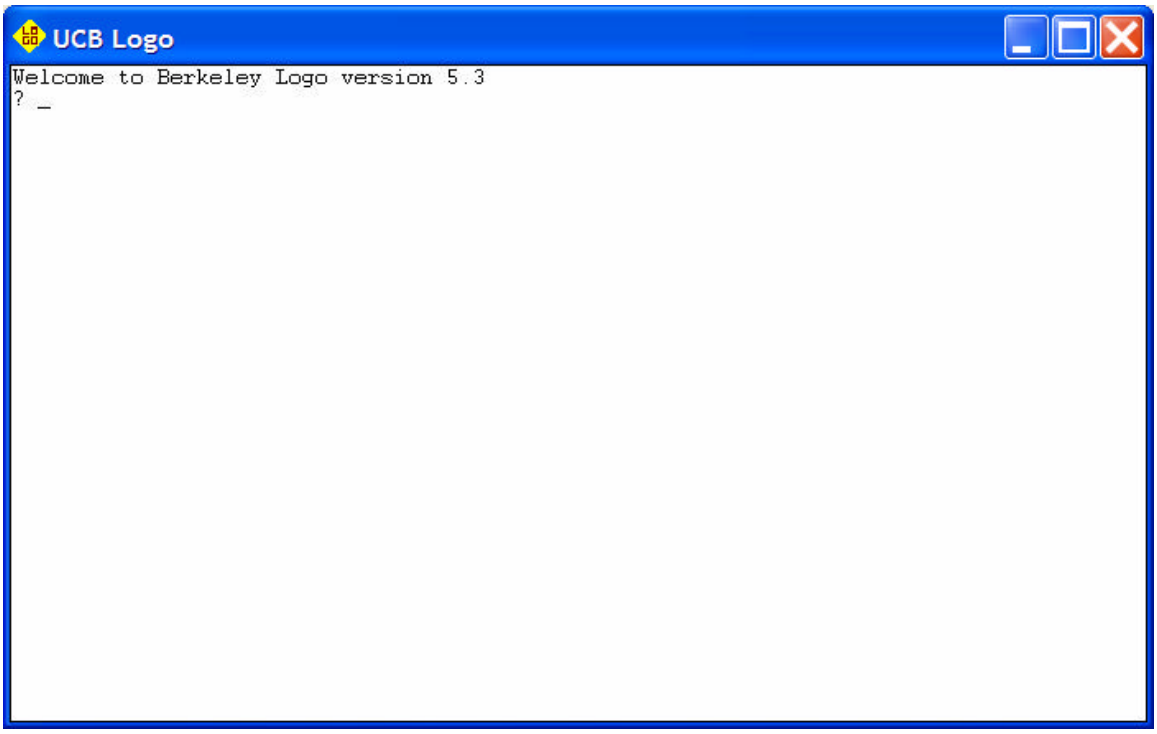
This guide was written to give you an understanding of the basics of the computer language Logo.

The web site http://logosurvey.co.uk/ lists 45 different versions of Logo. Among the most popular versions are MicroWorlds (http://www.microworlds.com), Terrapin Logo (http://www.terrapinlogo.com/), Berkeley Logo (http://www.cs.berkeley.edu/~bh/), MSWLogo (http://www.softronix.com/logo.html), and NetLogo (http://www.ccl.sesp.northwestern.edu/netlogo/). The last three versions listed can be downloaded free of charge. Examples in this guide are in MSWLogo.

Berkeley Logo was built by Brian Harvey at the University of California at Berkeley. Brian's web site (http://www.cs.berkeley.edu/~bh/) is a great source of information on Logo, especially the free downloadble PDFs for the three volumes of the second edition of *Computer Science Logo Style*.

The Berkeley Logo Screen

When you start up your Berkeley Logo application, you'll see a window that looks like the one in the picture below.



Under the "Welcome to Berkeley Logo" message, you will see a question mark and an underscore. This is called a *prompt*. The question mark is a signal that Logo is ready to hear from you.

Messages

Think of Logo as somebody who lives in your computer. When you type something at the prompt and press the Enter key, you send Logo a text message. Logo tries to understand the message and do what it says.

Type something at the Logo prompt. For example, try typing

```
? Hello, Logo
```

Then press the Enter. Pressing the Enter key sends your message to Logo. After Logo receives the message, it tries to figure out what job you want done and acts accordingly. When I typed my message, Logo responded by putting this into the Recall List Box:

```
? Hello, Logo
I don't know how to Hello,
? _
```

In this case, Logo didn't understand me because the word Hello is not in Logo's vocabulary. So Logo's responded by sending its own message as a reply to my message. Logo's message was:

```
I don't know how  to Hello,
```

The messages that you send to Logo are called *instructions*. Whenever you send Logo a message, Logo assumes that the first word in the message is some kind of *command*. If it knows how to do the command, it tries to do it. If it doesn't know how to do the command, it responds by sending a message back to you. The "`I don't know how to`…" message is one kind of *error message*.

Error messages are messages that Logo sends you when it's confused. These messages can be very useful when you need to figure out why Logo is having a hard time understanding you.

Commands and Inputs

The word `show` is a command that Logo understands. Try entering the instruction below at the Logo prompt (don't forget to press the Enter key when you're done):

```
? show "Hello
```

What happened? When I typed it, Logo responded like this:

```
? show "Hello
Hello
```

When Logo sees the `show` command, it takes it as the order: "Take whatever comes after the word `show` and type it on the next line.

What happens if we just enter the word `show` at the prompt nothing after it? Try it. Here's what I got:

```
? show
not enough inputs to show
```

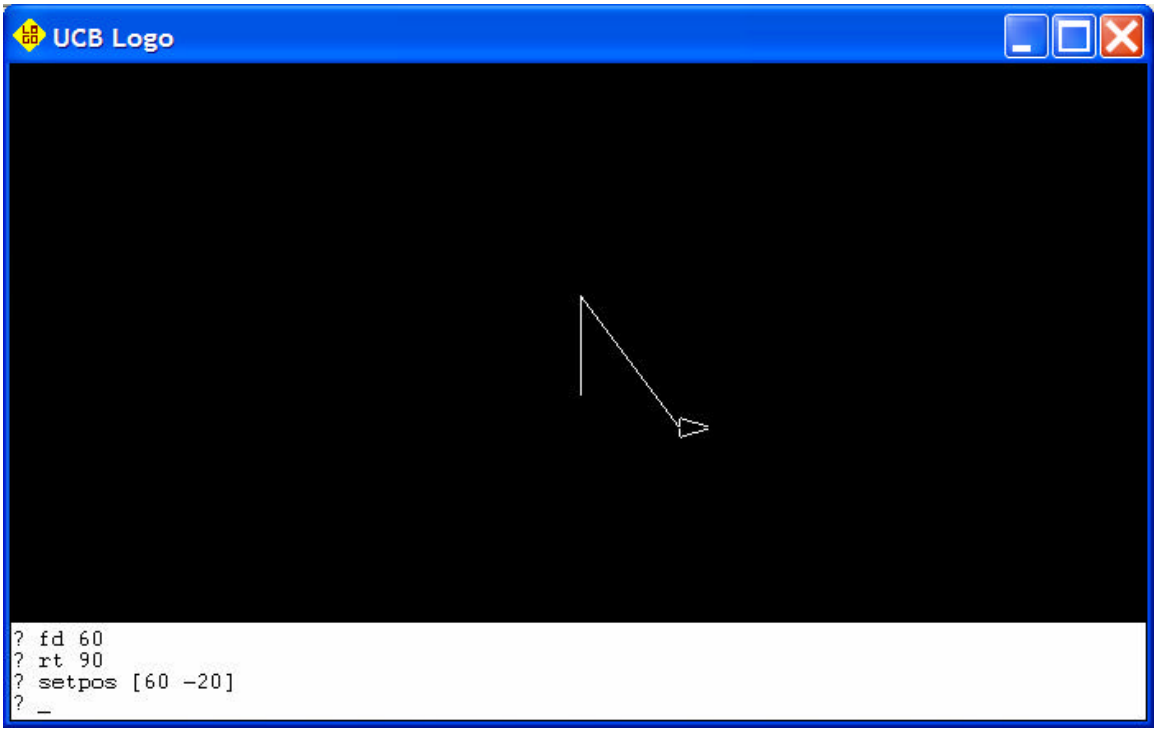
Logo sent us another error message. What does this one mean? Remember, when Logo sees the `show` command, Logo expects to see something after the word `show`. If that something isn't there, then the `show` command can't be completed, because there isn't anything to type on the next line.

The "something after the word `show`" is called the *input* to `show`. There are lots of commands that need inputs. Try out some of these instructions. Change the inputs, or enter them without inputs, and see what happens.

```
fd 60
rt 90
setpos [60 -20]
```

Commands and Inputs (continued)

This is what happened to my UCB window:



The commands `fd`, `rt`, and `setpos` are *turtle* commands. Logo responds to these commands by changing the position or direction of a little triangle on the screen, called a *turtle*. As the turtle moves around the screen, it can leave a trail behind it.

Whenever you send Logo a turtle command, Logo makes the turtle and its work area visible.

Here's an instruction that gave me a new error message:

```
? fd "hello
fd doesn't like hello as input
```

What do you suppose that means?

Operations and Output

So far, the `show` command doesn't seem very interesting. It just types out its input. But `show` can become a powerful command when we use it in combination with another kind of procedure called an *operation*.

We can think of commands as pets that need to be fed. The command's food is its inputs. Some commands don't need any inputs, so they're never hungry. Some commands are picky eaters who spit out inputs that they don't like (remember when we tried `fd "hello`?).

Once the pets are fed, they are happy, and they do the things they were meant to do. If your pet is a `show` command and you feed it the word `"hello`, it will type out `hello` in the Recall List Box. If your pet is a `fd` command and you feed it a `50`, it will move the turtle in the UCB Screen by 50 turtle steps.

Operations are critters that can prepare food for commands. For example, the word `heading` is an operation that prepares a number. The number gives the direction that the turtle is facing. We call this number the *output* of the `heading` operation. This number, the output that `heading` prepared, can be eaten by any commands that might be hungry (we will see shortly that some operations can get hungry, too).

Here's an example. Since `show` works with numbers (try it), we can execute the instruction:

```
? show heading
```

and get the response:

```
? show heading
0
```

If we change the direction the turtle is facing, we change the heading, so `heading` prepares a different meal for `show`:

```
? cs rt 90
? show heading
90
? lt 270
? show heading
180
? rt 180
? show heading
0
```

Operations and Output (cont'd)

Notice that the `rt` and `lt` commands tell the turtle to turn a certain number of degrees from its current position. The heading command tells you which way the turtle is facing, regardless of what turns it took to get there.

Even though fd is a finicky eater (it only likes numbers), the heading operation prepares the food that makes it happy, so we can also execute instructions like this:

```
? rt 30
? fd heading
? lt 90
? fd heading
? rt 60
? fd heading
? rt 45
? fd heading
```

Notice that when the heading is 0, the fd command is still executed without an error message. It just moves the turtle 0 steps. In other words, it doesn't move the turtle at all.

Operations can get hungry, too

There are also operations that need inputs.

For example, try this one:

```
? show sum 2 3
```

The `sum` operation prepares food for the show command, but the sum operation needs to eat, too. The sum operation needs two inputs. Each of them must be numbers. The sum operation prepares its output by adding together the two inputs. The show command takes this output as its input. Once show it happy, it does its job, showing the number it just ate:

```
? show sum 2 3
5
```

What will this instruction do?

```
? fd sum 100 20
```

Remember, operations prepare outputs that can be used by commands or by other operations, so we can also run instructions like the ones below. Try them out. Make changes to see if you can get error messages. When you get error messages, try to figure out why Logo is confused.

```
? fd sum 100 heading
? print sum 4 product 10 2
```

The `print` command is more or less the same as the `show` command. What does the `product` command do? Here's a new error message that I got:

```
? sum 4 7
You don't say what to do with 11
```

Operations prepare output (food). That output can be used (eaten) by other commands or operations, but that output is not an order that Logo can use. If you just hand some output to Logo, it doesn't know what it's supposed to do with that output.

Some terms

One of the instructions in the last section might have looked kind of confusing:

```
? print sum 4 product 10 2
```

I actually got this example from Brian Harvey's book, *Computer Science Logo Style: Symbolic Computing* (Harvey, 1997). I'll give you Brian's explanation of this instruction in a minute. But first I should explain a few more computer science terms.

Each command and operation has a to do list that tells it what it needs to do after it gets its input (or immediately, if it doesn't need input). This to do list is called a *procedure*. When you give Logo an instruction that names a particular command or operation, Logo *invokes* the procedure for that command or operation. Logo looks up the procedure and carries out the steps given in the procedure. This process of looking at instructions and invoking the necessary procedures is called *evaluation*.

OK, now you can go to the next page for Brian's explanation of the funny-looking `print` instruction.

An explanation from Brian Harvey (Harvey, 1997)

```
print sum 4 product 10 2
```

Here are the steps Logo takes to evaluate the instruction:

1. The first thing in the instruction is the name of the procedure `print`. Logo knows that `print` requires one input, so it continues reading the instruction line.
2. The next thing Logo finds is the word `sum`. This, too, is the name of a procedure. This tells Logo that the *output* from `sum` will be the *input* to `print`.
3. Logo knows that `sum` takes two inputs, so `sum` can't be invoked until Logo finds `sum`'s inputs.
4. The next thing in the instruction is the number 4, so that must be the first input to `sum`. This input, too, must be evaluated. Fortunately, a number simply evaluates to itself, so the value of this input is 4.
5. Logo still needs to find the second input to `sum`. The next thing in the instruction is the word `product`. This is, again, the name of a procedure. Logo must carry out that procedure to evaluate `sum`'s second input.
6. Logo knows that `product` requires two inputs. It must now look for the first of those inputs. (Meanwhile, `print` and `sum` are both "on hold" waiting for their inputs to be evaluated. `Print` is waiting for its single input; `sum`, which has found one input, is waiting for its second.) The next thing on the line is the number 10. This number evaluates to itself, so the first input to `product` is 10.
7. Logo still needs another input for `product`, so it continues reading the instruction. The next thing it finds is the number 2. This number evaluates to itself, so the second input to `product` has the value 2.
8. Logo is now ready to invoke the procedure `product`, with inputs 10 and 2. The output from `product` is 10 times 2, or 20.
9. This output, 20, is the value of the second input to `sum`. Logo is now ready to invoke `sum`, with inputs 4 and 20. The output from `sum` is 24.
10. The output from `sum`, 24, is the input to `print`. Logo is now ready to invoke `print`, which prints 24. (You were only waiting for this moment to arise.)

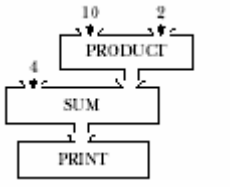
An explanation from Brian Harvey (cont'd)

That's a lot of talking about a pretty simple instruction! I promise not to do it again in quite so much detail. It's important, though, to be able to call upon your understanding of these details to figure out more complicated situations later. Using the output from one procedure as an input to another procedure is called *composition of functions.*

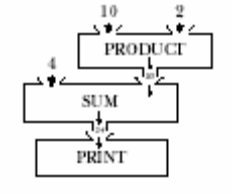
Some people find it helpful to look at a pictorial form of this analysis. We can represent each procedure as a kind of tank, with input hoppers on top and perhaps an output pipe at the bottom. (This organization makes sense because gravity will pull the information downward.) For example:

`Print` has one input, which is represented by the hopper above the tank. It doesn't have an output, so there is no pipe coming out the bottom. `Sum` has two inputs, shown at the top, and an output, shown at the bottom.

We can put these parts together to form a kind of "plumbing diagram" of the instruction:



In that diagram the output pipes from one procedure are connected to the input hoppers of another. Every pipe must be connected to something. The inputs that are explicitly given as numbers in the instruction are shown with arrows pointing into the hoppers.
You can annotate the diagram by indicating the actual information that flows through each pipe. Here's how that would look for this instruction:



Word and Lists

Notice that in step 4 of Brian's explanation above, he said that "a number simply evaluates to itself". This means that when we enter an instruction like

```
? fd 50
```

Logo knows that `50` is not some operation that needs to be evaluated. It's just the number `50`, so we can feed it directly to fd without invoking any procedure.

Anything that doesn't need to be evaluated by Logo is called a *constant*. There are basically two kinds of constants—words and lists. A number is a special kind of word.

Words

When Logo sees quote (") followed by other keyboard characters, it doesn't evaluate those characters. That's why

```
? show "hello
```

gives you

```
? show "hello
hello
```

but

```
show hello
```

gives you an error message:

```
? show hello
I don't know how to hello
```

Try these two instructions:

```
? show "heading
? show heading
```

How does Logo treat them differently? Why? Remember, the quotation mark tells Logo "Don't evaluate what follows".

Word and Lists (cont'd)

Numbers are Words, Too

A number is a special kind of word. You can enter numbers like you would enter any other word. Try these:

```
? fd "100
? show sum "3 "7
```

However, as we have seen, numbers can also be used without quotes:

```
? fd 100
? show sum 3 7
```

The tradeoff is that you can't use numbers as the names of procedures, because Logo never evaluates them. The designers of Logo figured this was a small price to pay to be able to use numbers without quotes.

Lists

Besides words, the other main kind of constant is a list. Lists are groups of words enclosed in brackets. For example, suppose you wanted Logo to say "Hi, how are you?". You could use this instruction:

```
? show [Hi, how are you?]
```

There are other procedures that like lists as input. For example, try this:

```
? setpos [100 100]
```

Try it with other lists. What kind of lists work? What kind of lists don't work? What does the command do?

There are also operations that output lists. Try these:

```
? show pos
? show butfirst [a b c]
? show butlast [how are you?]
```

What now?

Now that you know the basics of Logo, it would help you to understand more about the Berkeley Logo environment. You can read more about it at http://www.leonelearningsystems.com/IntroToMSWLogo.pdf.

There are some great resources for learning more about Logo at Brian Harvey's web site (http://www.cs.berkeley.edu/~bh/).

References

Harvey, B. (1997). *Symbolic Computing* (2nd ed. Vol. 1). Cambridge, MA: The MIT Press.

The Author

TJ Leone owns and operates Leone Learning Systems, Inc., a private corporation that offers tutoring and educational software. He has a BA in Math and an MS in Computer Science, both from the City College of New York. He spent two years in graduate studies in education and computer science at Northwestern University, and six years developing educational software there. He is a former Montessori teacher and currently teaches gifted children on a part time basis at the Center for Talent Development at Northwestern University in addition to his tutoring and software development work. His web site is http://www.leonelearningsystems.com.