

**tjeone.com**  
*Wonder. Create. Grow.*

TJ Leone  
237 Custer Ave  
Evanston, IL 60202  
Email [tjeone@chiaravalle.org](mailto:tjeone@chiaravalle.org)

Phone 847 951 0127  
Fax 847 733 8812

# Logo

---

## An Introduction

**TJ Leone**

**June 2009**

---

## Introduction

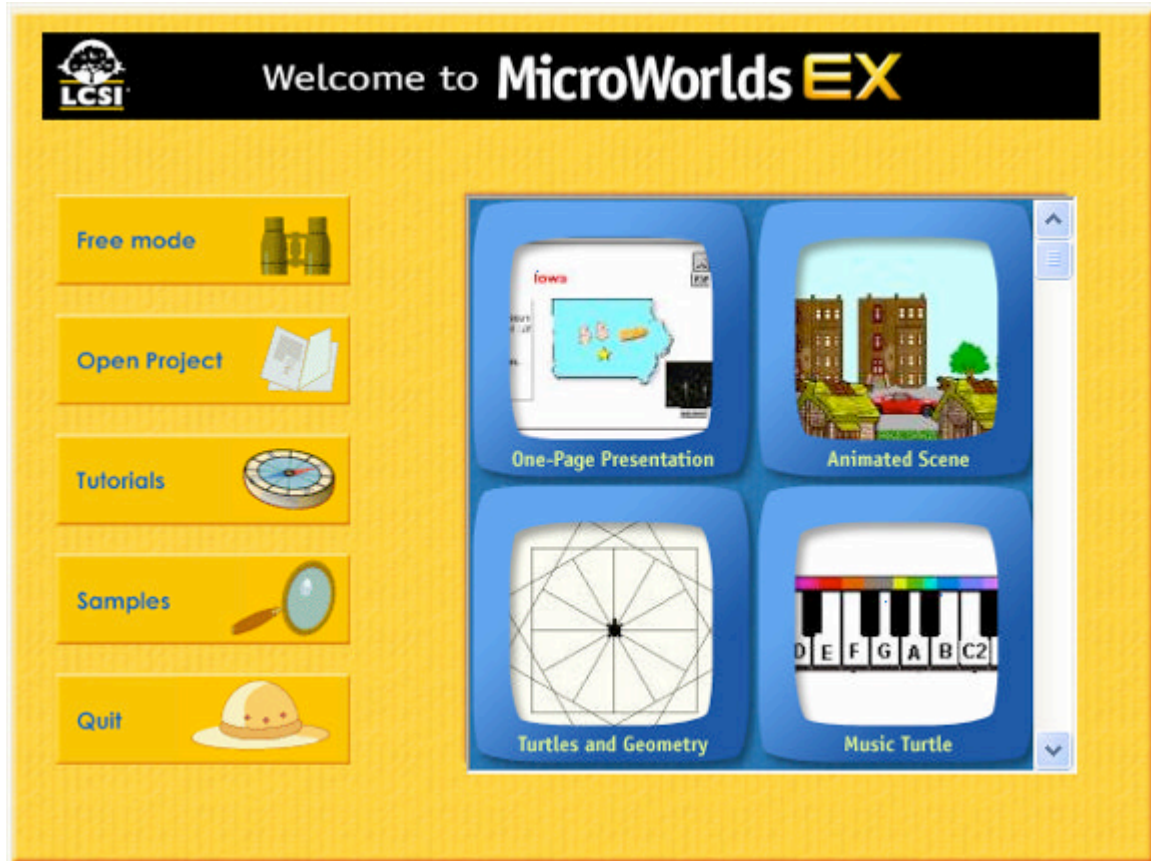
This guide was written to give you an understanding of the basics of the computer language Logo.

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 MicroWorlds Logo.

Brian Harvey's web site (<http://www.cs.berkeley.edu/~bh/>) is a great source of information on Logo, especially the free downloadable PDFs for the three volumes of the second edition of *Computer Science Logo Style*. MicroWorlds EX comes with an extensive tutorial, samples and help section. There is more help available at the MicroWorlds web site (<http://www.microworlds.com>).

## The MicroWorlds EX Welcome Screen

When you start up your MicroWorlds EX application, you'll see the following screen:



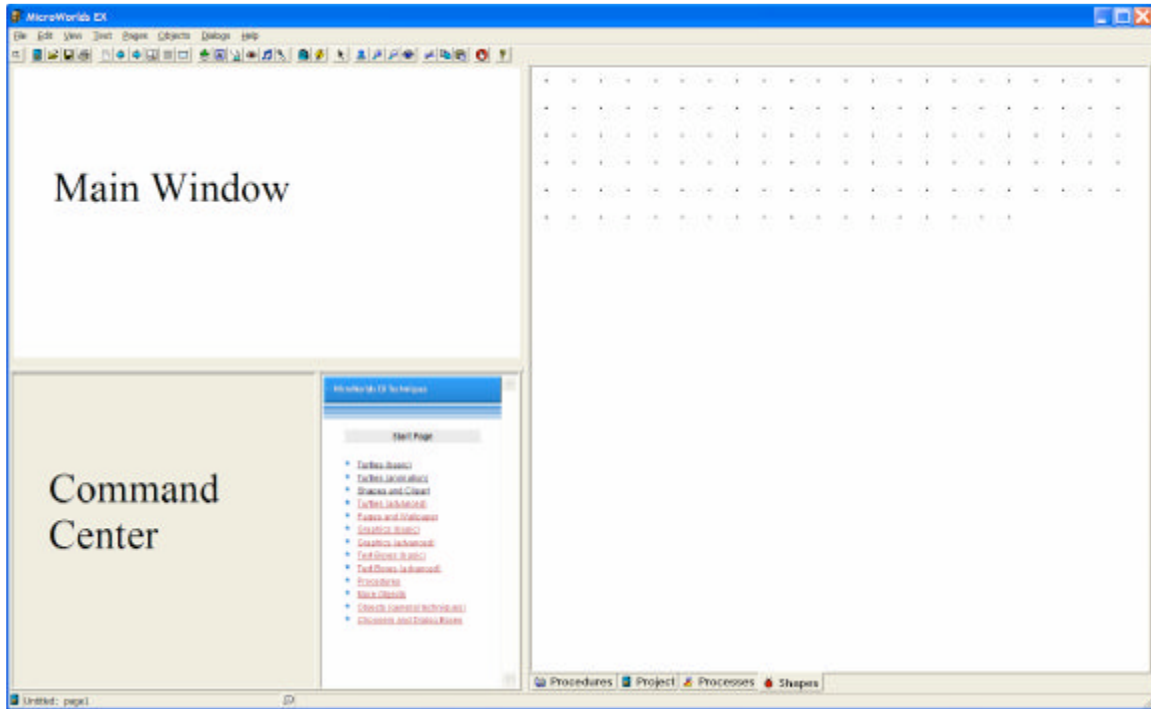
This is the MicroWorlds EX Welcome Screen. On the left is a list of buttons that suggest what you can do next:

<i>Free mode</i>	Start a brand new MicroWorlds project
<i>Open Project</i>	Open up a MicroWorlds project you already started
<i>Tutorials</i>	Get step-by-step instructions in building different projects
<i>Samples</i>	Look at projects that others have created
<i>Quit</i>	Close the MicroWorlds application

For now, click on the *Free mode* button. This will bring up the main MicroWorlds EX window.

## MicroWorlds EX Window

The picture below is about what you should see when you start a new MicroWorlds project in *Free Mode*. I added labels for the Main Window and the Command Center.



## Messages

The Command Center is the place for typing messages to Logo. Click in the Command and type something there. For example, try typing

```
Hello, Logo
```

Then press the Enter key. That sends the message. After you send Logo the message, Logo will respond somehow. Logo might respond by changing something in the Main Screen or in the Command Center. When I typed my message, Logo responded right below my message in the Command Center:

```
Hello, Logo  
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 into your Command Center (when I use words “enter it into your Command Center” or “enter the instruction” I mean to type it into the Command Center and then press the Enter key):

```
show "Hello
```

What happened? When I typed it, Logo responded in the Command Center:

```
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 in the Command Center”.

What happens if we just enter the word `show` into the Command Center with nothing after it? Try it. Here’s what I got:

```
show  
show needs more inputs
```

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 into the Command Center.

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. Before you start, make sure you have a turtle in the main window. Keep your eye on the turtle!

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

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?

## Reporters 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 a *reporter*.

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 Command Center. If your pet is a `fd` command and you feed it a `50`, it will move the turtle in the Main Screen by 50 turtle steps.

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

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

## Reporters 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 reporter 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.

## Reporters can get hungry, too

There are also reporters that need inputs.

For example, try this one:

```
show sum 2 3
```

The `sum` reporter prepares food for the `show` command, but the `sum` reporter needs to eat, too. The `sum` reporter needs two inputs. Each of them must be numbers. The `sum` reporter prepares its output by adding together the two inputs. The `show` command takes this output as its input. Once `show` is 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, reporters prepare outputs that can be used by commands or by other reporters, 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. Before you try out the `print` instruction below, make sure you have a text box on the page:

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

The `print` command is like `show` command except that it prints to the current text box. What does the `product` command do? Here's a new error message that I got:

```
sum 4 7
I don't know what to do with 11
```

Reporters prepare output (food). That output can be used (eaten) by other commands or reporters, 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 reporter 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 reporter, Logo *invokes* the procedure for that command or reporter. 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.)

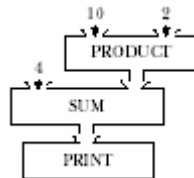
## An explanation from Brian Harvey (cont'd)

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*.

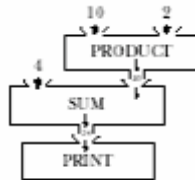
Some people find it helpful to look at a pictorial form of this analysis. We can represent each procedure as a kind of tank, with input hoppers on top and perhaps an output pipe at the bottom. (This organization makes sense because gravity will pull the information downward.) For example:



`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 reporter 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
```

in the Command Center, 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 reporters that output lists. Try these:

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

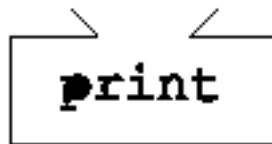
## Solved Problems

1. Make a tank and pipe diagram for the instruction below, tell what would be printed, and explain why.

```
print quotient 12 difference 7 3
```

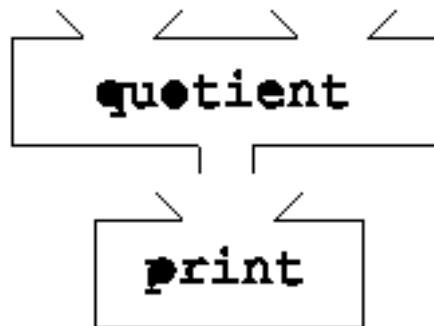
*Solution:*

The first procedure name in the instruction is `print`, so we can start by drawing a tank for `print` at the bottom of our diagram:



```
print quotient 12 difference 7 3
```

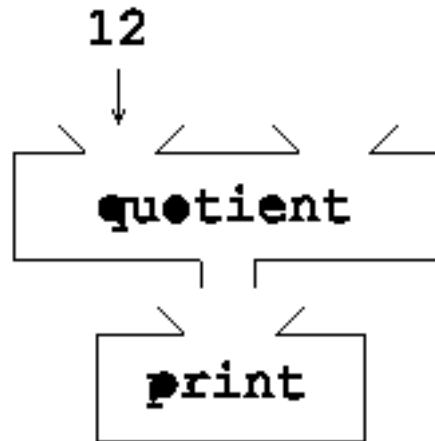
`print` is a command (not a reporter) so there is no output pipe at the bottom of the tank. `print` has one input (the world or list to be printed), so there is one hopper above the tank.



```
print quotient 12 difference 7 3
```

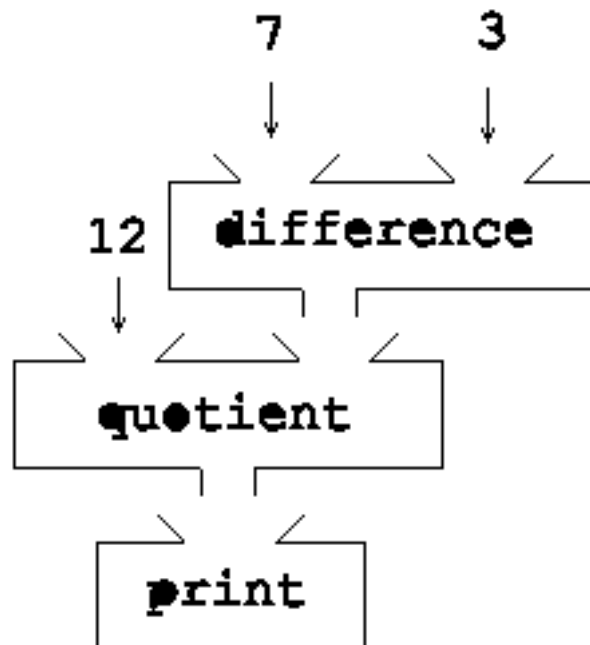
The next thing in the instruction is the procedure name `quotient`. `quotient` is a reporter, so we draw a tank with an output pipe that feeds into the `print` tank's hopper. `quotient` needs two inputs, so we draw two hoppers above the `quotient` tank:

## Solved Problems (continued)



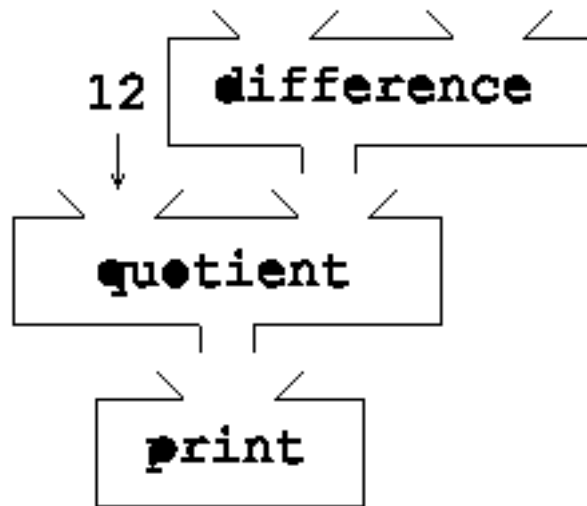
```
print quotient 12 difference 7 3
```

After `quotient`, the next thing we see on the instruction line is the number `12`, which evaluates to itself, so we feed the number `12` directly into the first hopper for `quotient`.



## Solved Problems (continued)

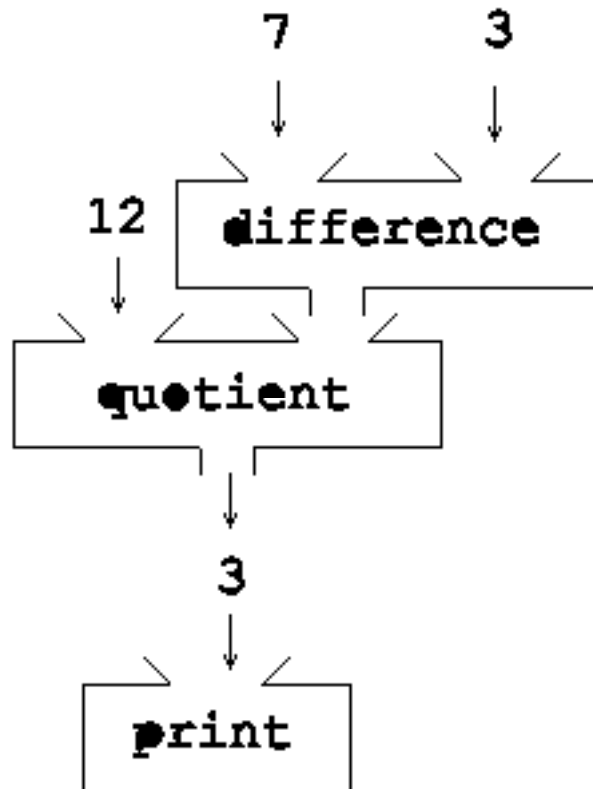
After the 12, we see the procedure name of another reporter, `difference`. Since `difference` is a reporter, we need to draw a tank with an output pipe. `Quotient`'s second hopper is still free, so we put the output pipe for `difference` over the second input hopper for `quotient`. Since `difference` needs two inputs, we need to draw two hoppers over the `difference` tank.



```
print quotient 12 difference 7 3
```

## Solved Problems (continued)

The next input, 7, evaluates to itself and feeds into the first hopper for `difference`. The last input, 3, also evaluates to itself and feeds into the second hopper for `difference`.

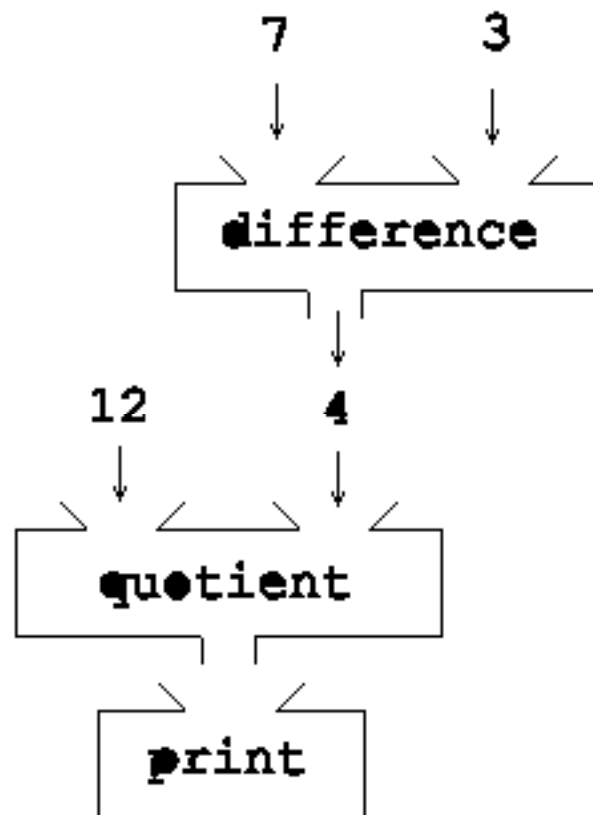


```
print quotient 12 difference 7 3
```

## Solved Problems (continued)

Now we can move through the diagram from top to bottom to see what the instruction does.

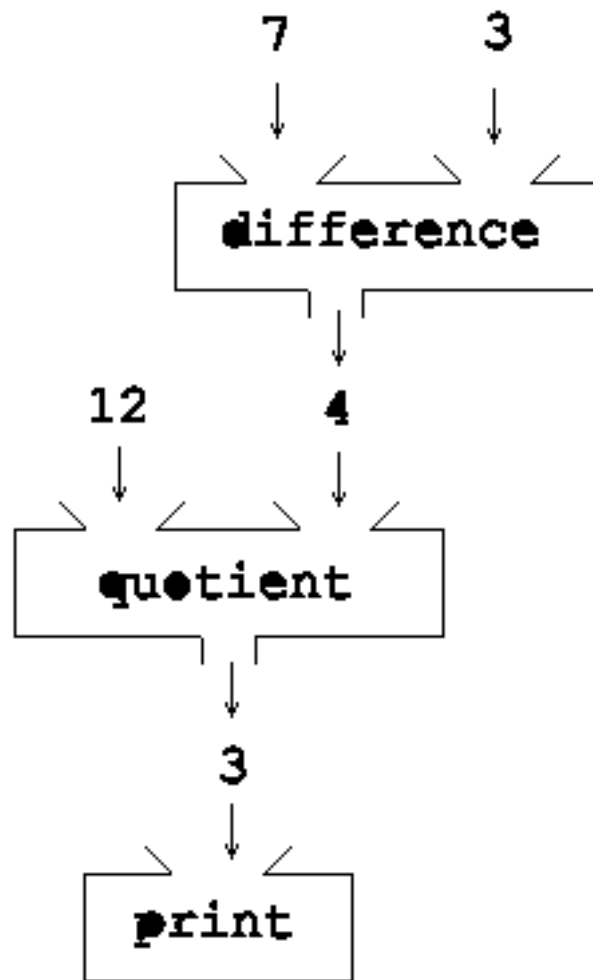
At the top of the diagram, we have 7 and 3, which evaluate to themselves. These give `difference` the two inputs needed (one for each hopper), so the reporter `difference` can now report a 4.



```
print quotient 12 difference 7 3
```

The output pipe from the `difference` tank empties into the second hopper of the `quotient` tank, so the second input to `quotient` is 4. This gives `quotient` all the inputs needed to produce an output

## Solved Problems (continued)



```
print quotient 12 difference 7 3
```

The `quotient` procedure uses the inputs 12 and 4 to produce an output of 3, which it reports to `print`. The `print` command prints the 3 to the current text box.

## Homework

Make diagrams of instructions using tanks, input hoppers and output pipes. Explain what Logo would do with the instruction. Here are some ideas for instructions you could try:

```
show sum 3 4
show difference 8 difference 5 1
print sum 5 product 3 4
show sum product 3 4 5
```

```
print butlast "tricky
show first butfirst "hello
show first butfirst [abc def ghi]
```

For an extra challenge, try more complicated instructions:

```
print product sum quotient 20 difference 8 3 2 4
show se butlast "farm butlast butlast butlast "output
show word word last "awful first butfirst "computer
first [go to the store, please.]
```

You might also try drawing the diagram first and then writing out the instruction that goes with it.