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

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

Phone 847 951 0127
Fax 847 733 8812

Combining Functions

TJ Leone

October 2004

Acknowledgements

Special thanks to Matt Vasher for being a careful reader and an excellent student. This document has fewer mistakes because of him.

Copyright © 2004 Leone Learning Systems, Inc.

Introduction

In the last unit, we saw that Logo primitives can be combined by using the output of an operation as the input to another operation or a command. For example, in the instruction

```
show sum 4 product 3 2
```

the output of the operation `sum` is the input for the command `show`, and the output of the operation `product` is the second input to `sum`.

In this unit, we will explore different ways to combine procedures that you have written yourself. In the process, we will revisit Trace and Step, which were discussed earlier in http://www.leonelearningsystems.com/IntroToMSWLogo.pdf (for MSWLogo users) and http://www.leonelearningsystems.com/IntroToBerkeleyLogo.pdf (for Berkeley Logo users). In MSWLogo, Trace and Step are two buttons in the interface, while Berkeley Logo provides the `trace` and `step` commands. As we will see, Trace and Step are useful tools for analyzing and debugging programs.

Invoking a Procedure from Within a Procedure

We have seen that once you create a procedure, you can invoke it by typing it into the Input Box (MSWLogo) or by typing it at the command prompt (Berkeley Logo) and then pressing the Enter key.

For example, consider these two procedures:

```
to triangle :side
     fd :side
     rt 120
     fd :side
     rt 120
     fd :side
     rt 120
end

to feet.to.inches :feet
     op product 12 :feet
end
```

The first procedure, `triangle`, is a command. We can invoke it directly like this:

```
triangle 70
```

The second procedure, `feet.to.inches`, is an operation. When we invoke it, we need to be sure that we pass its output to some other procedure that can use it:

```
show feet.to.inches 3
```

We can even pass the output of `feet.to.inches` to our `triangle` procedure:

```
triangle feet.to.inches 5
```

Try it.

Another way to invoke `feet.to.inches` would be to put it into another procedure. The `feet.to.inches` procedure would then be invoked when its line in the procedure is evaluated.

Invoking a Procedure from Within a Procedure (continued)

For example, we could write a procedure like this:

```
to triangle.inches :side
     fd feet.to.inches :side
     rt 120
     fd feet.to.inches :side
     rt 120
     fd feet.to.inches :side
     rt 120
end
```

We can invoke this procedure like this:

```
triangle.inches 6
```

What happens when you try this one?

The evaluation of these procedures is easier to see if you trace it. If you're using MSWLogo, press the Trace button. If you're using Berkeley Logo, type this at the Logo prompt:

```
? trace [triangle.inches feet.to.inches]
```

Invoking a Procedure from Within a Procedure (continued)

Now invoke `triangle.inches` again like we did above. You should see something like this:

```
triangle.inches 6
( triangle.inches 6 )
 ( feet.to.inches 6 )
 feet.to.inches outputs 72
 ( feet.to.inches 6 )
 feet.to.inches outputs 72
 ( feet.to.inches 6 )
 feet.to.inches outputs 72
triangle.inches stops
```

The first sentence that appears after we press Enter is

```
( triangle.inches 6 )
```

This shows that Logo is trying to execute `triangle.inches` with the input 6. The next line that appears is:

```
( feet.to.inches 6 )
```

Where did this come from? The first line of the `triangle.inches` procedure is

```
     fd feet.to.inches :side
```

so Logo knows that before `( triangle.inches 6 )`can finish executing, we need to execute `( feet.to.inches 6 )`. Notice that during execution, Logo knows that the variable name `:side` should be replaced by the number 6. Also notice that the `( feet.to.inches 6 )` line starts a little farther to the right than the line above it. This is how trace indicates that the `feet.to.inches` instruction was invoked by the `triangle.inches` instruction.

The next line is:

```
feet.to.inches outputs 72
```

After `feet.to.inches` has found the product of 6 and 12, it outputs that product (72).

Invoking a Procedure from Within a Procedure (continued)

You can see by the trace that `feet.to.inches` is invoked three times with input 6 and produces the output 72 each time. We know by looking at the procedure that the input to `fd` is the output of feet.to.inches, so the turtle moves forward 72 turtle steps each time it draws a side of the triangle.

Composite Functions

Suppose we wanted to convert some number of yards into its equivalent in inches.

We already have `yards.to.feet`, which converts yards to feet and `feet.to.inches` which converts feet to inches. We can combine these two functions together like this:

```
to yards.to.inches :yards
     op feet.to.inches yards.to.feet :yards
end
```

In "An explanation from Brian Harvey" (http://www.leonelearningsystems.com/IntroToLogo_MSW.pdf), we saw how Logo primitives are evaluated (primitives are procedures that come pre-made inside Logo when you install it). In this section we will also see how Logo evaluates procedures that you supply.

Suppose we invoke the procedure by sending Logo this message:

```
show yards.to.inches 2
```

We send Logo the message by typing it into the Input Box for MSWLogo or by typing it at the Logo prompt for Berkeley Logo.

Logo looks up the procedure `show`, which it recognizes as a primitive command that needs one input, so Logo goes to the next item after `show` to get the input. The next word is `yards.to.inches.`

Logo looks up the procedure `yards.to.inches`. Logo first looks to see if this procedure is a primitive. The top line of the procedure definition tells Logo that `yards.to.inches` needs one input. The input name is `:yards`, so whenever we see `:yards` in the `yards.to.inches` procedure, we replace it with the input.

Now Logo knows that `yards.to.inches` is a procedure and needs one input, so Logo goes to the next word in the message, which is the number `2`. To execute the procedure, Logo executes each instruction in the procedure, replacing `:yards` with the number `2`. There is only one line in this procedure. Replacing `:yards` with the number `2`, we have:

```
     op feet.to.inches yards.to.feet 2
```

Composite Functions (continued)

The command `op` needs one input. Logo moves to the next word looking for input to `op`. The next word is `feet.to.inches`, which is an operation that needs one input. So

Logo must keep looking while `feet.to.inches` and are `op` waiting for their inputs. The next word after `feet.to.inches` is `yards.to.feet`, which also needs one input. Finally, after `yards.to.feet` Logo finds a constant, the number `2`, which evaluates to itself.

Now things start to happen. Using the input `2`, `yards.to.feet` creates the output `6`, which Logo passes to `feet.to.inches`. Using the input `6`, `feet.to.inches` creates the output `72`, which Logo passes to `op`. The `op` command passes off the `72` to the `show` command, which shows the `72`:

```
show yards.to.inches 2
72
```

Another way to understand the evaluation we just discusses is to trace it. If you're using MSWLogo, press the Trace button. If you're using Berkeley Logo, type this at the Logo prompt:

```
? trace [yards.to.inches yards.to.feet feet.to.inches]
```

Now invoke `yards.to.inches` again like we did above. You should see something like this:

```
show yards.to.inches 2
( yards.to.inches 2 )
 ( yards.to.feet 2 )
 yards.to.feet outputs 6
 ( feet.to.inches 6 )
 feet.to.inches outputs 72
yards.to.inches outputs 72
72
```

Can you see how a trace helps explain the action?

Traces can be very useful in tracking down problems when your procedures get complicated.

Composite Functions (continued)

When you are finished tracing, you may want to turn it off. If you are using MSWLogo, you can turn tracing off by pressing the UnTrace button so that its label changes to Trace. If you are using Berkeley Logo, you can use the command

```
? untrace [yards.to.inches yards.to.feet feet.to.inches]
```

Algebraic form of Composite Functions

We can also write the composition of functions using algebraic notation.

For example, let's start with two Logo operations:

```
to g :x
     op difference product 3 :x 1
end

to h :y
     op product :y :y
end
```

We can invoke these two operations in many different ways:

```
show g 7
show h 2
show h g 4
```

In algebraic notation, we would say

$g(x) = 3x - 1$

and

$h(y) = y^2$

The y^2 is shorthand for $y \times y$. It is read as "y squared".

The Logo functions above could then be written in algebraic form as

$g(7)$
$h(2)$
$h(g(4))$.

Solved Problems

Use the following algebraic function definitions as needed in the problems below:

$f(x) = 2x + 3$
$g(x) = 7 - x$
$h(x) = x^2 + 2x$

1. *Problem:* Write a Logo operation for each of the algebraic functions.

Solution:

The expression $f(x)$ tells us that the function name is f and the input is x. In Logo, input names must start with a colon (:), so the first line of our Logo function definition is:

```
to f :x
```

After the equal sign, we see the expression $2x + 3$. This expression means "Multiply x by 2 and then add 3". In Logo, this is written as

```
sum product 2 :x 3
```

See "An explanation from Brian Harvey" in the document *Logo* (http://www.leonelearningsystems.com/IntroToLogo_MSW.pdf) if this is not clear.

Since we are writing an operation (not a command), we need the `op` command to tell Logo to output the result of the sum. Below is the entire procedure:

```
to f :x
    op sum product 2 :x 3
end
```

In a similar way, we can write the other functions as Logo procedures:

```
to g :x
    op difference 7 :x
end

to h :x
    op sum product :x :x product 2 :x
end
```

Solved Problems (continued)

2. *Problem:* Using a pencil and paper, find:
   a. $h(f(1))$
   b. $g(g(2))$

   Solution:

   a. To find $h(f(1))$, we first evaluate $f(1)$ and then use the output of that function as the input of h. Since $f(x) = 2x + 3$, $f(1) = 2\times1+3$ (remember, $2x$ is shorthand for $2\times x$), which equals $2 + 3 = 5$. Now we use the output of f as the input to h. Since $h(x) = x^2 + 2x$, we now have $h(f(1)) = h(5) = 5^2+2\times5 = 5\times5+2\times5 = 25 + 10 = 35$.

   b. In a similar way, we have

   $$g(g(2)) = g(7 – 2) = g(5) = 7 – 5 = 2$$

   If these solutions aren't clear, take a look at the traces in problem 3. Come back at look at this solution section again if you need to.

3. *Problem:* Use the procedures you wrote in problem 1 and the computer to check your answers in problem 2.

   Solution:

   In 2a, we found that $h(f(1)) = 35$. Here's a command to test it.

```
show h f 1
35
```

   Here's what it looks like with a trace:

```
show h f 1
( f 1 )
f outputs 5
( h 5 )
h outputs 35
35
```

   The output of `h f 1` (35) matches our answer for $h(f(1))$, so our answer checks out.

Solved Problems (continued)

In 2b, we found that $g(g(2)) = g(7 - 2) = g(5) = 7 - 5 = 2$. Compare this to a trace of the corresponding Logo operations:

```
show g g 2
( g 2 )
g outputs 5
( g 5 )
g outputs 2
2
```

Our answer in both cases is 2, so this solution checks out as well.

4. *Problem:* In the last unit, we wrote two operations for converting volumes of liquid:

```
to quarts.to.pints :quarts
    op product :quarts 2
end

to gallons.to.quarts :gallons
    op product 4 :gallons
end
```

Write an operation called that combines these two operations to correctly convert gallons to pints.

Solution: The function `gallons.to.quarts` takes some number of gallons as its input and outputs the equivalent amount as measured in quarts. This output can be used as input to the operation `quarts.to.pints`:

```
to gallons.to.pints :gallons
    op quarts.to.pints gallons.to.quarts :gallons
end
```

Solved Problems (continued)

5. *Problem:* Write a command called `house` that needs one input. The procedure draws a house using the procedures `triangle` and `square`. The input tells the procedure how long to make each side of the triangle and each side of the square.

   Solution: We need to combine the `triangle` and `square` procedures into a bigger procedure called `house`. I started with the following procedures for `triangle` and `square`:

```
to triangle :side
     fd :side
     rt 120
     fd :side
     rt 120
     fd :side
     rt 120
end

to square :side
     fd :side
     rt 90
     fd :side
     rt 90
     fd :side
     rt 90
     fd :side
     rt 90
end
```

   Now I will try invoking the two procedures from inside a new procedure called `house`:

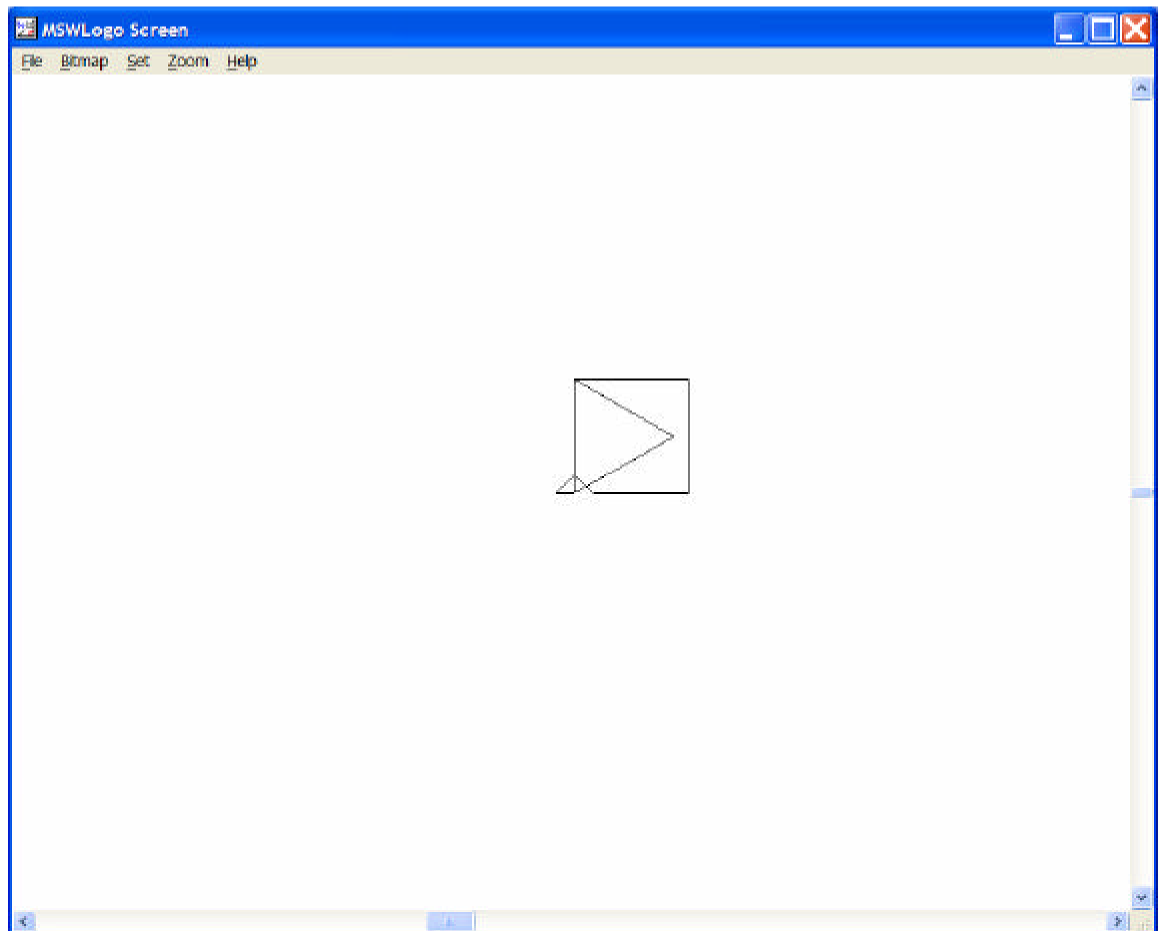
```
to house :side
     pd
     square :side
     triangle :side
end
```

Solved Problems (continued)

Now that I have a `house` procedure, I'll invoke it to draw a house:

```
house 100
```

The command above produced the following picture:



What happened? I have a triangle and square, but it doesn't look anything like a house.

When you have a problem like this, it is often helpful to step through execution. If you are using MSWLogo, you can step through execution by pressing the Step button so its label changes to UnStep. If you are using Berkeley Logo, you can use the command:

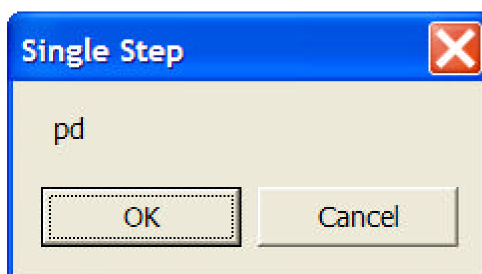
```
? step [house square triangle]
```

Solved Problems (continued)

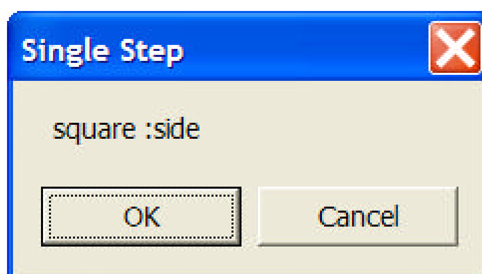
After you have set up stepping with the Step button (MSWLogo) or the `step` command (Berkeley Logo), you can invoke the house procedure:

```
house 100
```

What happens? The step function shows you what instruction is being executed and prompts you to continue. If you are running MSWLogo, you will see a prompt like this:



The first command in my house procedure is pd, so Step shows me that procedure. After I press the OK button, the pd procedure is invoked, and the next dialog box appears:



If you are using MSWLogo, you can continue through each step of each procedure by continuing to click OK or by pressing the Enter key whenever a dialog box appears.

When I stepped through the `house` command with Berkeley Logo, I saw a prompt like this:

```
[square :side] >>>
```

For some reason, the `pd` command wasn't included in the stepping. However, all the other instructions are included. Berkeley Logo users can step through the

Solved Problems (continued)

`house` procedure by pressing the Enter key every time a new prompt appears.

What went wrong?

You probably noticed that when the turtle was finished drawing the square, it was back at the bottom left corner of the square pointing straight up. The next thing it did was to draw a triangle. It went forward 100 along the left side of the square, then turned right 120 into the square, went forward 100, turned right 120, then went forward 100 to complete the triangle. At this point, the turtle was at the bottom left corner of the original square. Finally, the turtle turned 120 to face straight up.

If this explanation is not clear to you, try rereading it as you step through the execution again.

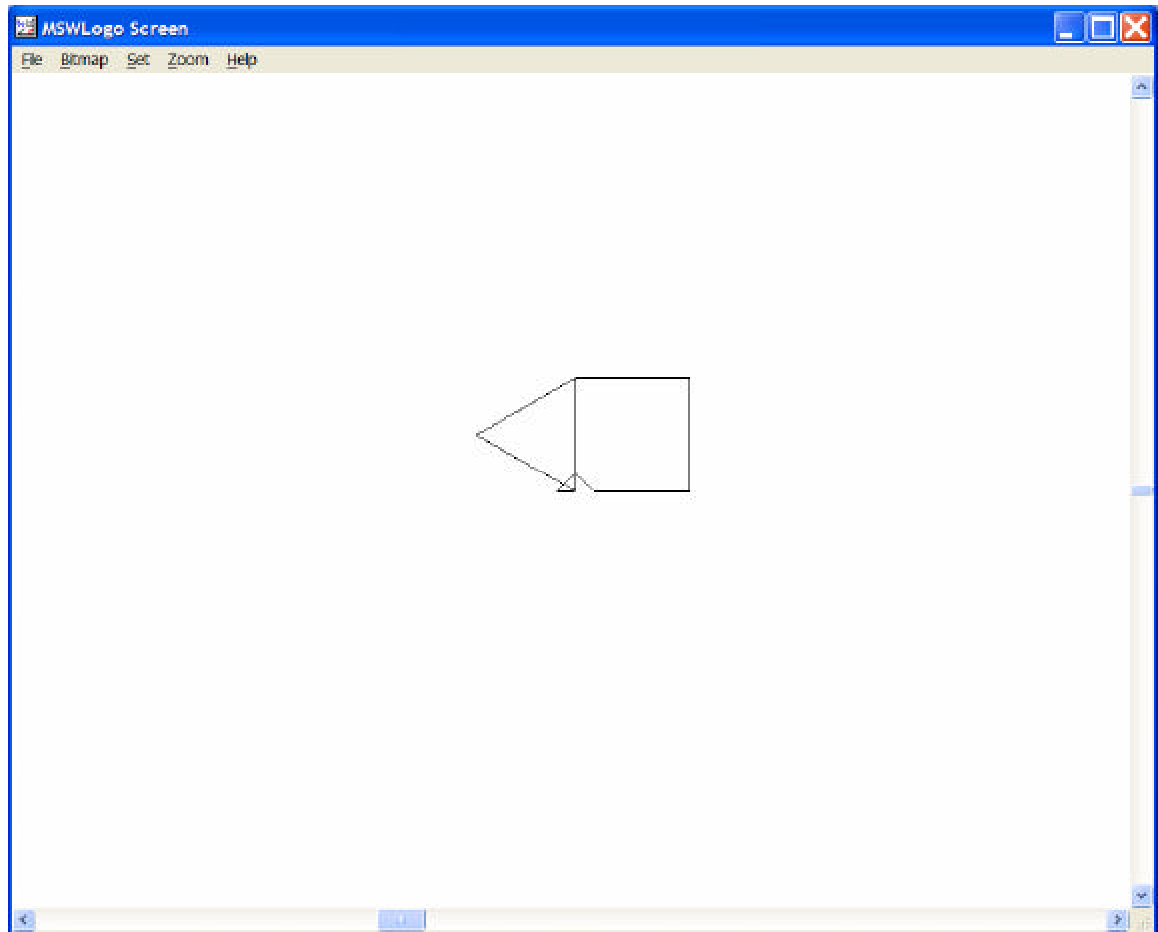
How can we fix this problem? There's more than one way to do it. Try to think of your own way before you read any further.

One of the solutions that might have occurred to you is to make the triangle turn left instead of right. We can create a left-turning triangle called `trianglel`:

```
to trianglel :side
     fd :side
     lt 120
     fd :side
     lt 120
     fd :side
     lt 120
end
```

Solved Problems (continued)

With a left-turning triangle, `house 100` would draw this:



We still have a problem. We don't want the house lying on its side. We want it standing up. Another way to say this is: we want the house to be rotated 90 degrees to the right.

To do this, we simply add a `rt 90` command to `house` before we do any drawing:

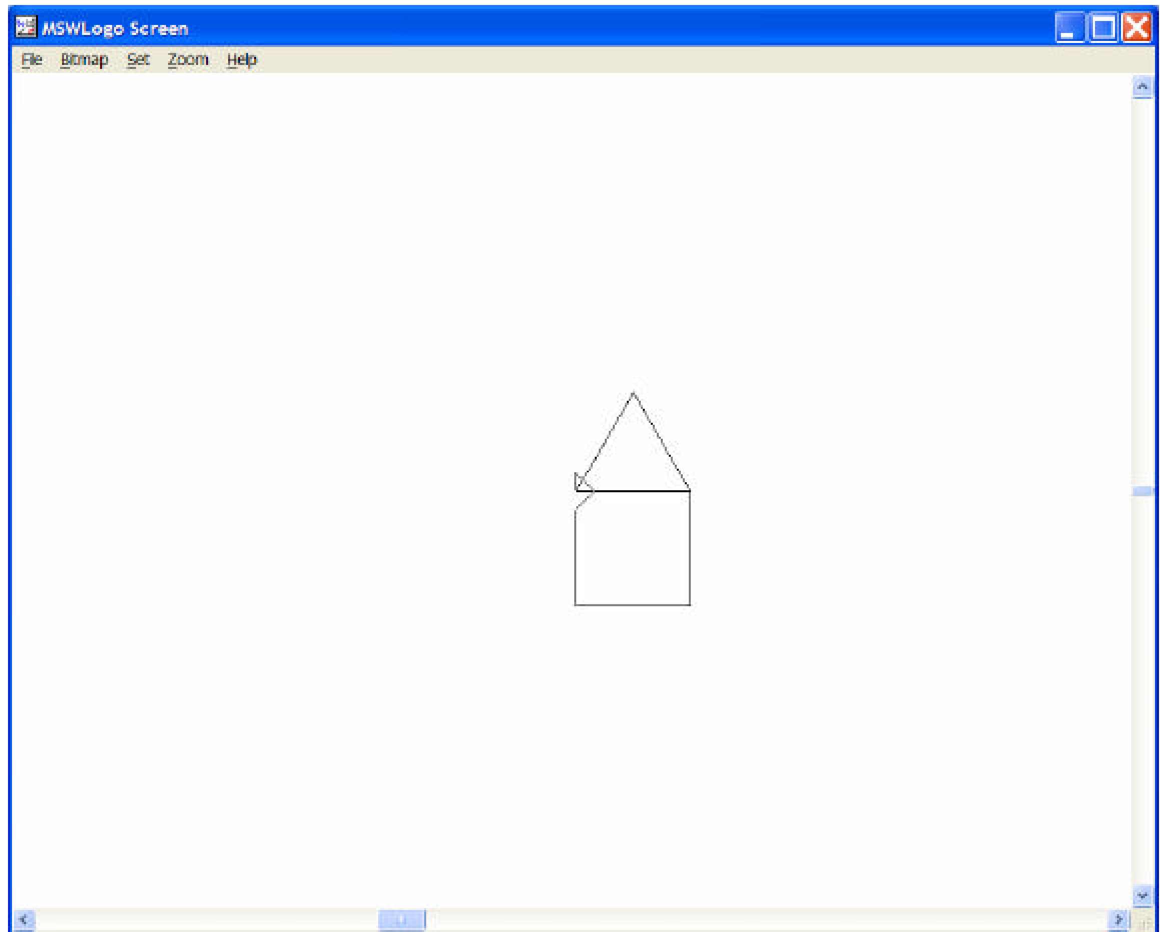
```
to house :side
     rt 90
     pd
     square :side
     trianglel :side
end
```

Copyright © 2004 Leone Learning Systems, Inc.

⋮

Solved Problems (continued)

Now our `house 100` draws this:



Another possible solution is shown in the Supplementary Problems section.

Supplementary Problems

Use the following algebraic function definitions as needed in the problems below:

$f(x) = 3x - 1$
$g(x) = x + 2$
$h(x) = x^2$

1. *Problem*: Write a Logo operation for each of the algebraic functions.

   Solution:

```
to f :x
    op difference product 3 :x 1
end

to g :x
    op sum :x 2
end

to h :x
    op product :x :x
end
```

2. *Problem:* Using pencil and paper find:

   a. $f(g(2))$
   b. $g(f(2))$
   c. $h(g(3))$

   Solution:

   a. $f(g(2)) = f(2 + 2) = f(4) = 3 \times 4 - 1 = 12 - 1 = 11$
   b. $g(f(2)) = g(3 \times 2 - 1) = g(6 - 1) = g(5) = 5 + 2 = 7$
   c. $h(g(3)) = h(3 + 2) = h(5) = 5^2 = 5 \times 5 = 25$

Supplementary Problems (continued)

3. *Problem:* Use the procedures from problem 1 and the computer to check your answers in problem 2.

   Solution:

   Test for $f(g(2)) = 11$

```
show f g 2
( g 2 )
g outputs 4
( f 4 )
f outputs 11
11
```

   $g(f(2)) = 7$

```
show g f 2
( f 2 )
f outputs 5
( g 5 )
g outputs 7
7
```

   $h(g(3)) = h(3 + 2) = h(5) = 5^2 = 5 \times 5 = 25$

```
show h g 3
( g 3 )
g outputs 5
( h 5 )
h outputs 25
25
```

Copyright © 2004 Leone Learning Systems, Inc.

Supplementary Problems (continued)

4. *Problem:* Here are procedures for converting different measures of information:

```
to kilobytes.to.bytes :kilobytes
op product 1024 :kilobytes
end

to bytes.to.bits :bytes
op product 8 :bytes
end
```

Use these two procedures to write a new procedure called `kilobytes.to.bits` that converts kilobytes into bits.

Solution:

```
to kilobytes.to.bits :kilobytes
op bytes.to.bits kilobytes.to.bytes :kilobytes
end
```

Supplementary Problems (continued)

5. *Problem:* We saw above in the Solved Problems section that there was a problem with the following combination of procedures:

```
to triangle :side
     fd :side
     rt 120
     fd :side
     rt 120
     fd :side
     rt 120
end

to square :side
     fd :side
     rt 90
     fd :side
     rt 90
     fd :side
     rt 90
     fd :side
     rt 90
end

to house :side
     pd
     square :side
     triangle :side
end
```

Come up with another way to fix this problem besides the solution in the Solved Problems section.

Solution: Leave square and triangle as they are, and change house to this:

```
to house :side
     pd
     square :side
     fd :side
     rt 30
     triangle :side
end
```

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.