Scaling

You can scale anything that you can draw in Logo. For example, here's a procedure that draws an L:

```
to el
fd 100
bk 100
rt 90
fd 50
bk 50
lt 90
end
```

Try this out.

How would you make an L that's twice as big? If you want to try it before you see the answer, then try it now before you read further.

Here's an L that's twice as big as the one above:

```
to el.double
fd 200
bk 200
rt 90
fd 100
bk 100
lt 90
end
```

What did we change to change the size of the L? Why didn't we change the angles?

Try making an L that's three times as big as the first one, or half as big. Try scaling to change some other procedure.

Scaling with Variables

Variables make it easy for you try out lots of different scales. For example, let's look again at our L procedure:

```
to el
fd 100
bk 100
rt 90
fd 50
bk 50
lt 90
end
```

This procedure is that it only draws an L of one size. Now let's add a variable:

```
to el2 :scale
fd 1.00 * :scale
bk 1.00 * :scale
rt 90
fd 0.50 * :scale
bk 0.50 * :scale
lt 90
end
```

I like having at least one of my `fd`'s equal to 1 because that way it's easier for me to think about how `:scale` will be used in the procedure. That's why I changed the 100 to 1.00 and the 50 to 0.50. Since I divided all my `fd`'s and `bk`'s by the same number (100), the overall shape of the L stays the same.

Now you can try:

```
? el 100
? el 50
? el 27
? el -19
```

Make your own shapes and scale them. What kind of input makes a shape bigger? What makes it smaller? What does negative input do? Why? Experiment with using / instead of *. What happens if you add or subtract the scale amount instead of multiplying or dividing by it?

Scale Randomly

A number of programming languages have special procedures that output numbers that appear to be random. Logo's version of this procedure is called `random`. The `random` procedure is an operation like `heading` or `pencolor`. You can't use the word random as the first word in an instruction. You use it to give information to some other procedure.

Here are some examples:

```
show random 6     ;; Logo picks a number from 0 to 5 and shows it
rt random 360     ;; Logo picks a number from 0 to 359 and turns the
                  ;; turtle that amount to the right
fd 20 + random 80 ;; Logo moves the turtle forward some amount between
                  ;; 20 and 99
```

So, you can use random to pick a size for your scaled path like this:

```
to random.els
cs
pu lt 90 fd 300 rt 90
repeat 10 [
      pd
      el2 random 100
      pu
      rt 90 fd 120 lt 90
]
end
```

The procedure above draws 10 L's in random sizes. The `cs` in first line clears the screen. The instructions in the next line move the turtle over to left to give it room to draw the L's. The line with `repeat` causes `el2` to be invoked ten times.

What does the line with `el2` do?

```
      el2 random 100
```

You can try this line by itself to make sure you see what it's doing:

```
? el2 random 100
```

Now try this:

```
? repeat 30 [random.els wait 10]
```

Try your own random animations.

More Random Behavior

You can also use random to make random turns:

```
to spinsquare.random
repeat 10 [square rt random 360]
end
```

or to repeat something a random number of times:

```
to how.high
repeat random 10 [square2 50 fd 50]
end
```

or to go forward random amounts

```
to weird.path
repeat 100 [fd random 100 rt 144]
end
```

Recursion

We've seen that we can create a procedure (for example, the `square` procedure) and use that procedure name inside another procedure definition. But what happens when you use a procedure name inside its own definition? Here's an example:

```
to poly :side :angle
fd :side
rt :angle
poly :side :angle
end
```

This procedure takes two inputs—one for `:side` and one for `:angle`. So we could try things like:

```
? poly 100 72
? poly 100 144
? poly 100 60
? poly 1 1
```

What is happening? The last line of `poly` keeps things running over and over again by saying "do `poly` again" as part of the definition of `poly`.

Recursion can allow us to create some cool effects, because we can change the input to a procedure each time we invoke it. For example, try this:

```
to polyspi :side :angle
fd :side
rt :angle
polyspi :side + 10 :angle
end
```

The procedure above adds 1 to :side every time `polyspi` is invoked. Try the following:

```
? window
? polyspi 1 95
? polyspi 10 90
? polyspi 10 120
? polyspi 10 117
```

The `window` instruction makes it so that the turtle doesn't wrap around to the other side of the screen when it reaches the edge.

You will need to halt execution each time you invoke `polyspi` by using the Command key and the period.

If you want the turtle to wrap when it draws, use the `wrap` instruction:

```
? wrap
```

More Recursion

You can take one of your scaled paths and make it grow by using recursion. For example:

```
to grow.el :size
pd el2 :size
pu fd 15 + :size
grow.el :size * 1.1
end
```

What does this do?

You can also change the angle with recursion:

```
to inspi :side :angle :inc
fd :side
rt :angle
inspi :side :angle + :inc :inc
end
```

Try these

```
? inspi 10 0 7
? inspi 40 40 30
? inspi 60 2 20
```

Triangles to label

Labeling the angles, turns and side lengths of these triangles might help you draw them. The first triangle is a right isosceles triangle. The second is a scalene right triangle made by cutting an equilateral triangle in half.