Logotacular - tutorial

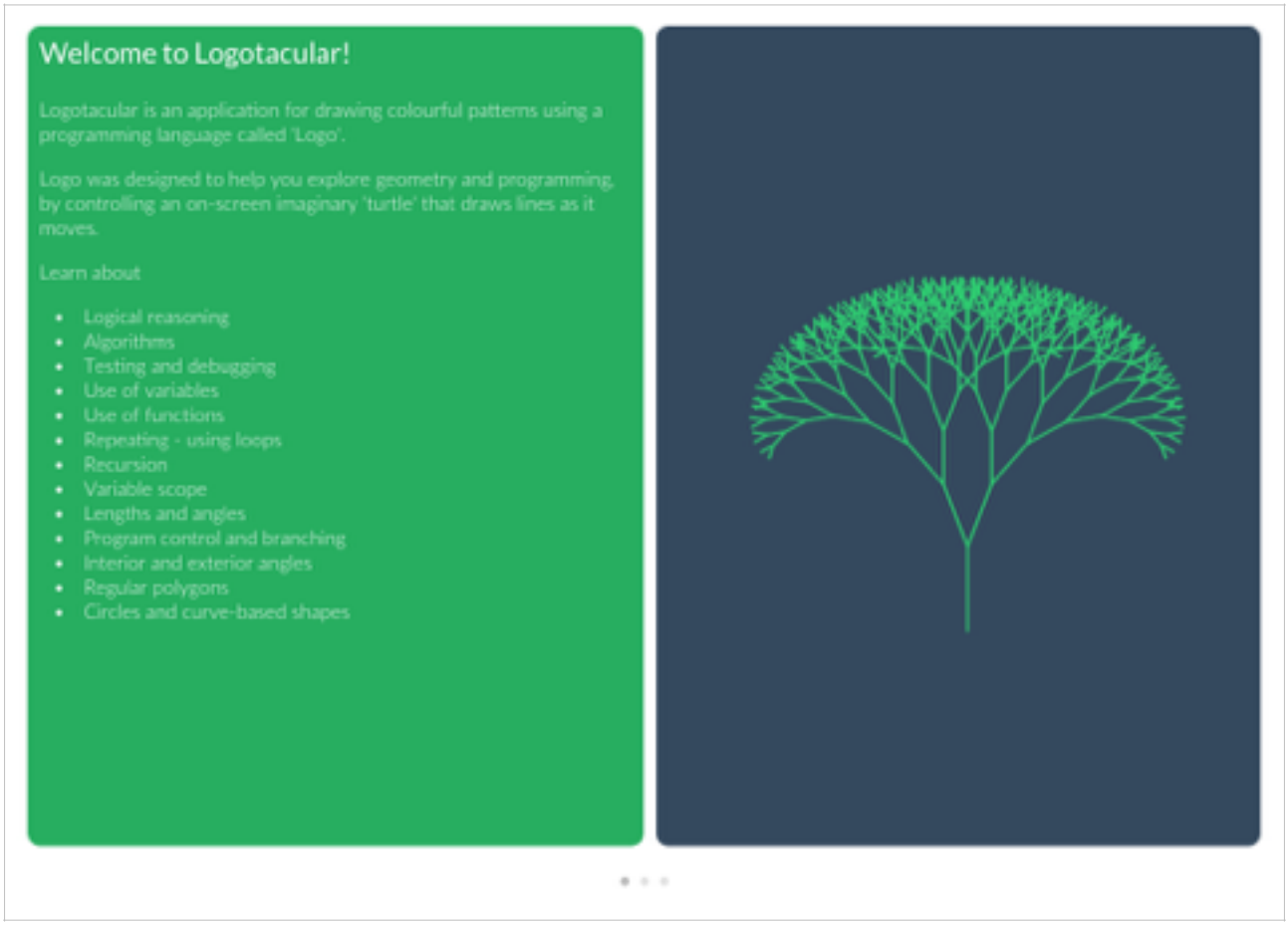
Logotacular is a colourful app to help you learn 'Logo', a programming language designed to assist in learning geometry and programming.

The most basic commands in Logotacular are the 'go forward' command and the 'turn right' command. You use these commands to control a small on-screen 'turtle', which draws a colourful path as it moves. You can change the background colour, line colour and the thickness of the lines drawn, as well as and raising/ lowering the pen.

Loops can be programmed by use of the 'repeat' command which tells the turtle to execute a set of commands a specific number of times.

More advanced users can use variables are used to tell the turtle the value assigned to a letter, and functions (or 'procedures') are used to name a set of commands so that you can easily execute them later.

I no longer support or work on the web application version of Logotacular since the iPad app has gained more traction. The screenshots below are taken from the iPad app, but I keep the legacy web application in case anyone wants to use it and doesn't have an iPad.



2. How to write a program

In Logo we imagine a 'turtle' that starts at the centre of the screen, facing upwards, and is controlled by commands. The turtle is shown by a triangle, pointing in the direction it is going to move in.

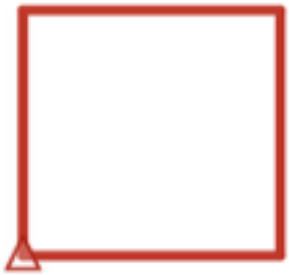
As it moves around the screen the turtle leaves a trail showing you where it has moved to.

The most basic commands in Logo are 'go forward' and 'turn right'. For example `fd 100` tells the turtle to go forward 100 units (in whatever direction it is facing), and `rt 90` tells it turn 90 degrees clockwise.

Try drawing a square using:

```
fd 100
rt 90
fd 100
rt 90
fd 100
rt 90
fd 100
rt 90
```

You don't have to put each command on a new line, but it can make it easier to read!



3. Basic controls

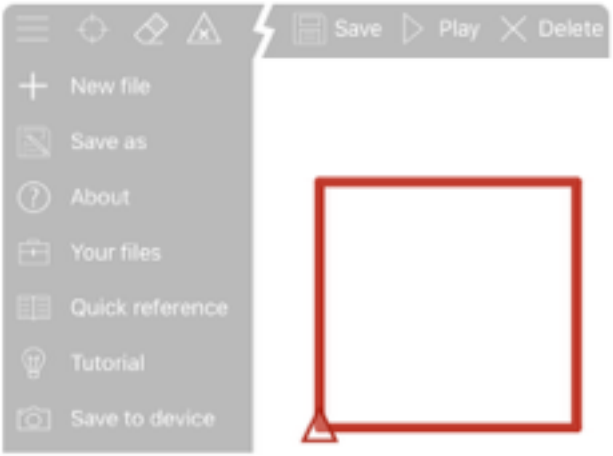
You can turn left instead of right using the `lt` command, go backwards using the `bk` command and at any time you can reset the turtle's position to the centre, facing upwards, by using the `home` command.

The four icons along the top-left of the app comprise the following:

- Open a drop down menu containing links to make a new file, save your file, and other useful actions such as saving a screenshot to your device.
- The cross-hairs button resets the viewport, useful if you have scrolled or scaled the view.
- The eraser button clears the drawing area (but not your code!)
- The triangular tick and cross button lets you show and hide the turtle. Seeing where the turtle is can be useful, but you might want to hide it when you're done.

The icons at the top-right comprise:

- Save your file with its current filename.
- Play (tell the turtle to draw your code) or stop drawing.
- Clear the code panel completely and start from scratch.



4. Drawing with colors

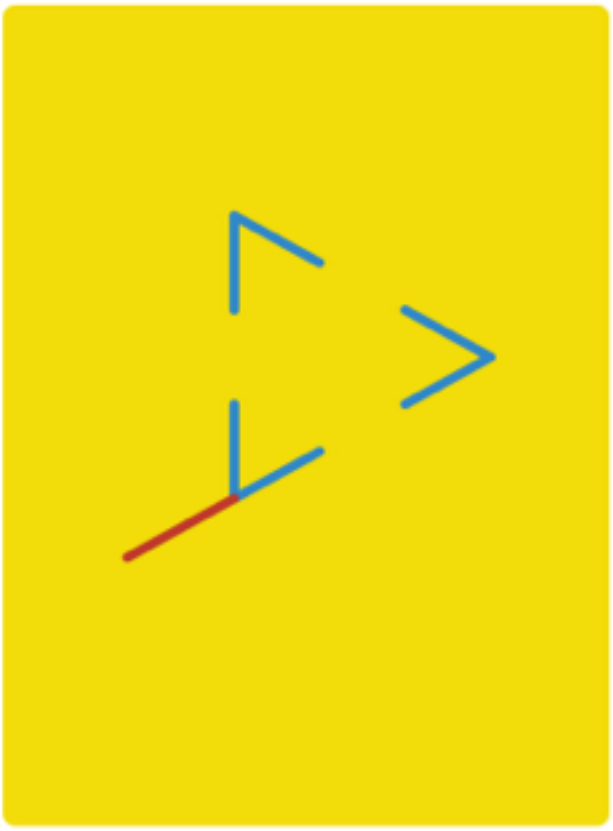
You can change the color of the lines, as well as the background color of the page by using the **color** (or **colour**) and **bg** commands:

```
bg yellow / bg 17
color blue / color 8
```

You can change the thickness of the lines drawn (from 1 to 20), and raise or lower the pen. Lifting the pen (**penup** or **pu**) will stop lines being drawn until you place it down again (**pendown** or **pd**).

```
bg yellow color blue
thick 8
fd 80 penup
fd 80 pendown
fd 80 rt 120
fd 80 penup
fd 80 pendown
fd 80 rt 120
fd 80 penup
fd 80 pendown fd 80
```

You can get a random colour using the keyword **random**:



5. Using variables

Variables let you define a number so you can use it in many places later. Not only do they let you re-use a value, but later on you only have to change them one place if you need to. Use the **make** command to set the value of a variable:

```
make "side 200
```

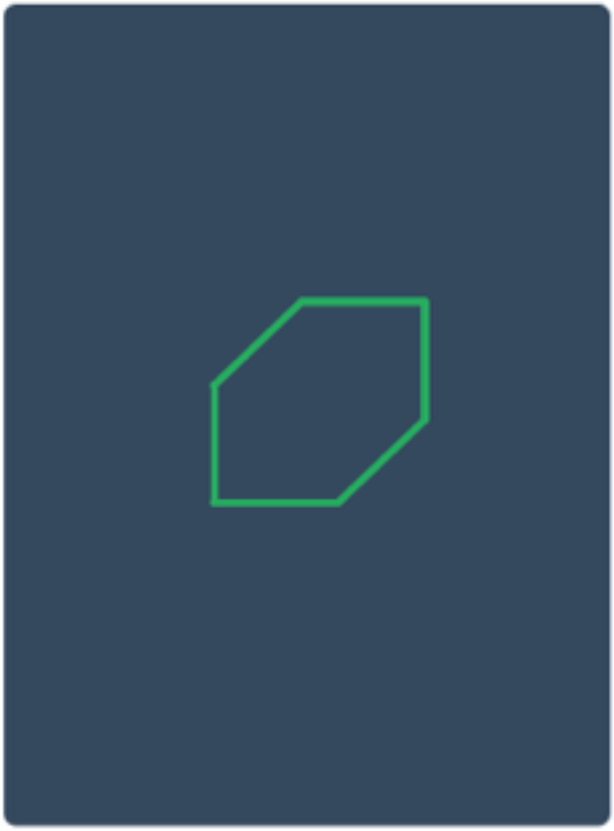
You must put a **"** symbol in front when you set a variable. To use the variable, put a colon **:** in front:

```
fd :side
```

This is the same as writing **fd 200**, but we have named the variable and can change it later.

Remember to use **"** when you're setting a variable and **:** when you want to use it.

```
make "side 100
fd :side rt 45
fd :side rt 45
fd :side rt 90
fd :side rt 45
fd :side rt 45
fd :side
```



6. Doing arithmetic

You can do normal arithmetic using numbers or variables.

Like most computer programs, you must write `*` for multiply (or times) and `/` for divide.

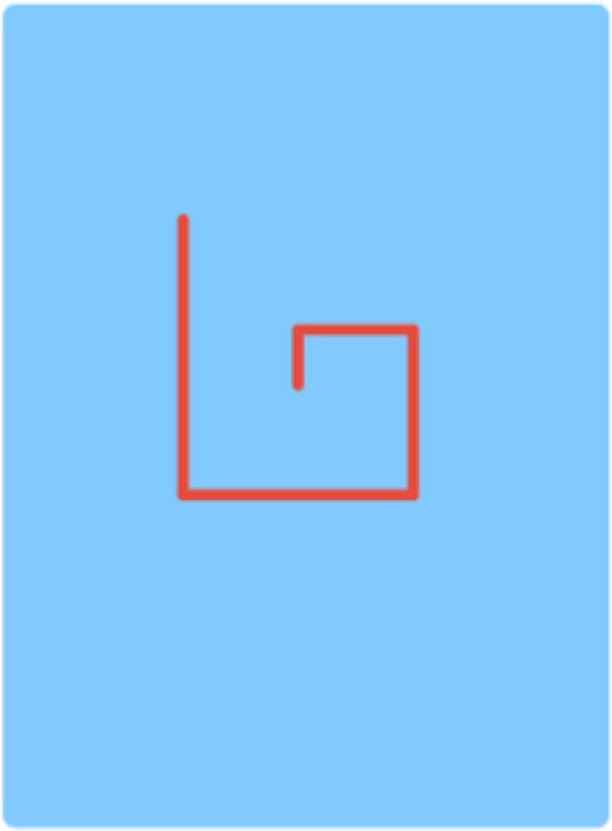
You can use `+` and `-` and round brackets `( )` like you normally would.

For example:

```
make "h 30
make "w 3*:h
fd (:w - 2*:h) rt 90
fd (:w - :h) rt 90
fd :w rt 90
fd :w + :h  rt 90
fd :w + 2*:h rt 90
```

You can get a random number between 0 and 100 by using the keyword `random`. For example:

```
fd random
```



7. Loops

Loops can be programmed by use of the `rpt` (repeat) command which tells the turtle to execute the commands inside the square brackets a specific number of times.

```
rpt 10 [ put your commands here and they
will be executed 10 times! ]
```

You can draw a square like this:

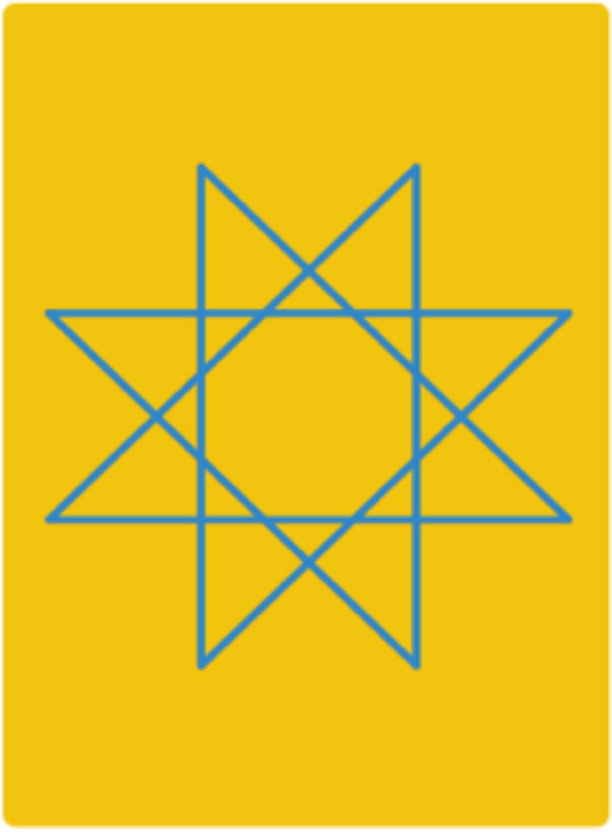
```
rpt 4 [ fd 100 rt 90 ]
```

You must start a repeat block with `[` and you must end it with `]`.

Can you work out how to draw the star on the right?

Can you work out what the following code will draw?

```
make "a 72
make "n 5
rpt :n [fd 100 rt :a]
```



8. Loops continued

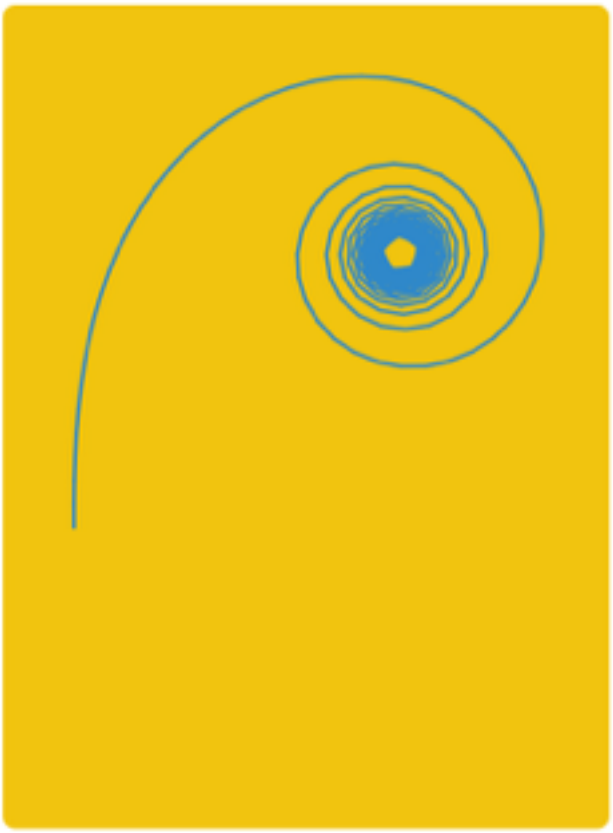
By turning right a small amount and repeating many hundreds of times we can create curves.

Interesting effects can be created by modifying the value of a variable inside a loop.

For example, in the program below, the variable **a** is increased by one in each step of the loop, which is repeated 300 times to create a spiral shape.

```
make "a 3
rpt 300 [
  fd 20
  rt :a/4
  make "a :a + 1
]
```

The spaces at the start of the lines are optional but can make your program easier to read



9. Loops continued

You can even program a loop inside another loop.

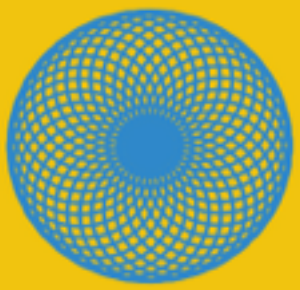
This program ...

```
rpt 360 [
 fd 1
 rt 1
]
```

... will create a circle shape, because turning right by 1 degree 360 times makes a whole turn.

Repeating that 36 times, turning 10 degrees each time creates the attractive pattern on the right:

```
rpt 36 [
rpt 360 [
 fd 1
 rt 1
]
rt 10
]
```



10. Procedures and functions

We know that the turtle knows the meaning of the words `make`, `penup`, `color` etc ...

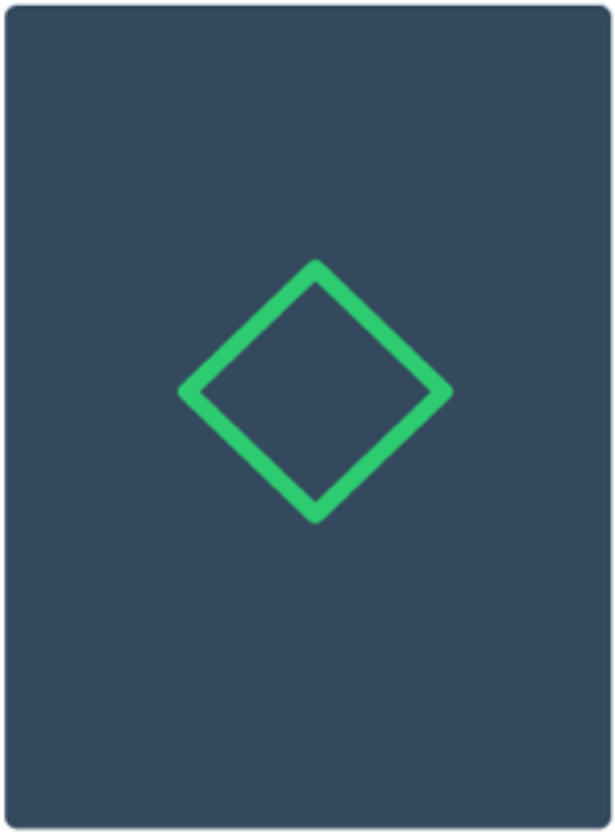
A function or procedure is often thought of as telling the turtle the meaning of a new word, for example we could draw a square by defining a procedure called 'drawsquare' and then 'calling' that procedure.

You use the `to` keyword to define a procedure, and `end` to mark the end:

```
make "side 150
to drawsquare
  rpt 4 [
    fd :side
    rt 90
  ]
end

drawsquare
```

The lines between `to` and `end` define the procedure and the last line 'calls' or 'executes' the function.



11. Procedures and functions continued

Functions can optionally take some variables as input. For example, you might want to re-use a block of commands more than once, with different lengths or angles each time.

List the variables one by one after the name of the function as shown:

```
to drawpoly :len :n
  rpt :n [
    fd :len
    rt (360/:n)
  ]
end
drawpoly 120 3
drawpoly 120 4
drawpoly 120 5
drawpoly 120 6
drawpoly 120 7
```

The procedure above has two arguments, called `len` and `n`, and it is then called five times with `len` equal to 120 and `n` equal to 3, 4, 5, 6 and 7 respectively.

12. Recursion

Recursion happens when, inside a function, you call the same function, normally with different input.

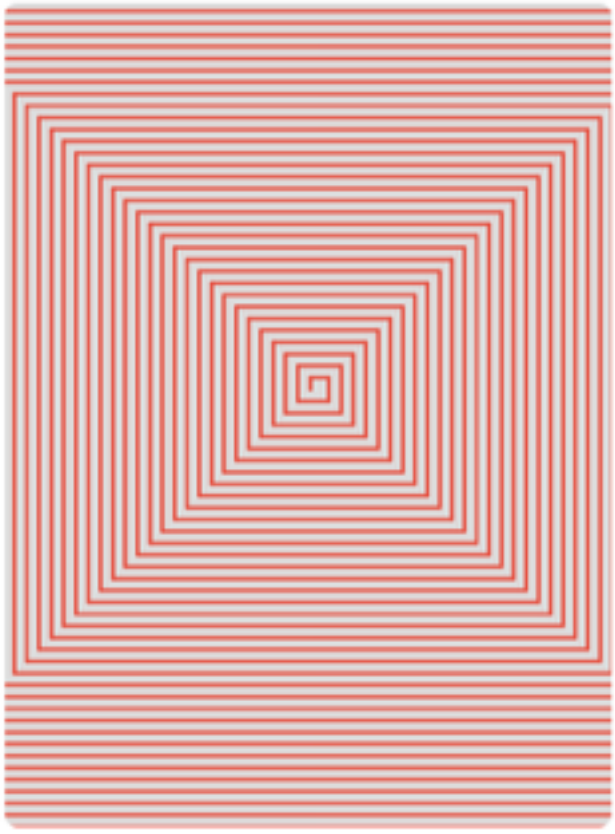
For example the program below defines a procedure which accepts one variable called `dist` and repeatedly calls itself, increasing the value of `dist` by 5 each time:

```
to drawspiral :dist
  fd :dist
  rt 90
  drawspiral (:dist+5)
end

drawspiral 10
```

The result is an increasing series of line segments, with lengths 10, 15, 20, 25... etc - forming a spiral.

Eventually the program will crash because it calls itself too many times!



13. Variable scope

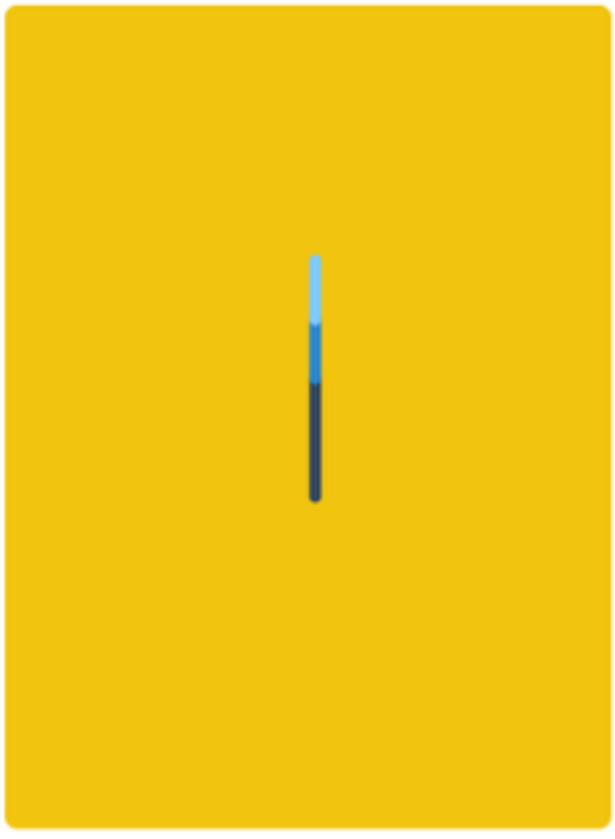
The program below illustrates the concept of 'scope' in Logotacular.

Every variable exists in a 'scope' depending on where you defined it, so you can have two variables with the same name that exist in different 'scopes'. For example:

```
make "j 50
to fn
  fd :j
end
fn
```

A variable called `j` is set to 50 outside of the function. When called, the function will check if a variable called `j` has been defined again inside the function itself.

In this case it hasn't, so the function uses the variable `j` from outside. We can say that `j` has 'global scope' because any code (in a function or not) can use it.



14. Variable scope continued

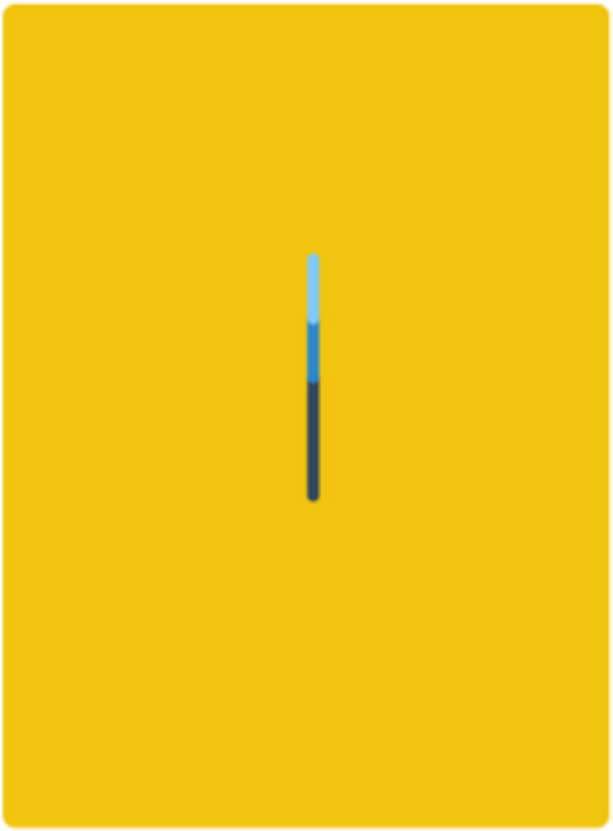
You can also define variables inside functions, and then they will exist inside that function only:

```
make "i 50
to fn
 make "i 100 fd :i
end
fn
fd :i
```

A variable called `i` is set to 50 outside the function. But when the function is called it immediately sets another variable called `i` to 100, and goes forward 100 units.

On the last line we go forward 'i' units again, but this time it will go forward 50 units.

This is because there are actually two different variables called `i`, with different scope, and the one that equals 100 exists temporarily, only inside the function.



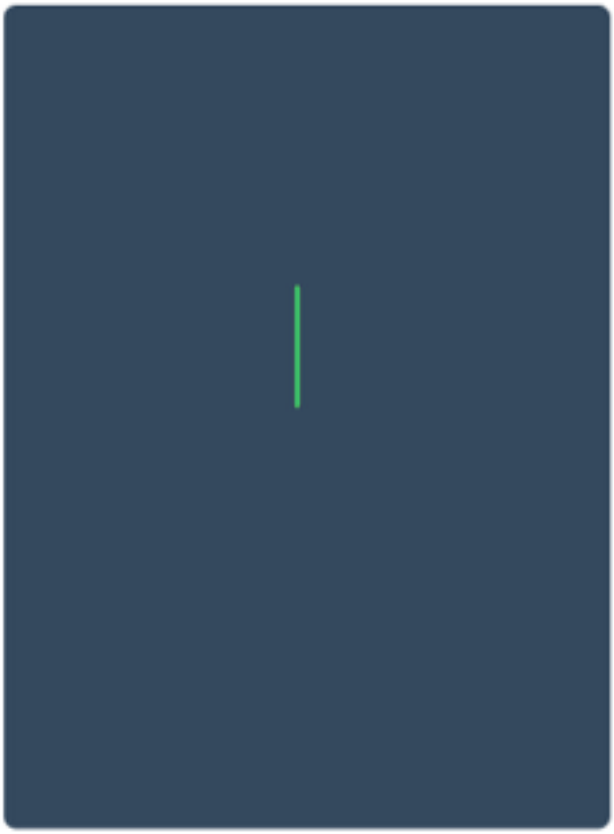
15. Stopping

You can use the keyword `stop` to stop a function or loop executing.

```
rpt 4 [
 fd 100 rt 90
 stop
]
```

Without the `stop` statement the program above would draw a square.

With the `stop` statement it will go forward once and turn 90 degrees, and then the loop will stop.



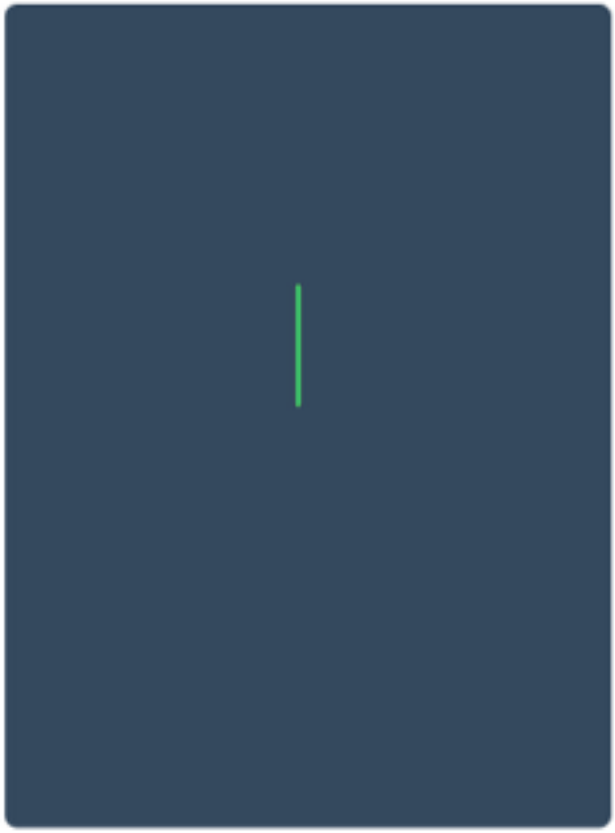
16. Stopping continued

In this second program the function `drawoneline` is called, and the turtle goes forward.

It reaches the `stop` statement and the function stops executing immediately.

```
to drawoneline
 fd 100
 stop
 rt 90
 fd 100
end
drawoneline
```

As a consequence, this code also just draws one 100 unit line.



17. Program control

Sometimes you only want to execute a line of code under certain circumstances - perhaps after checking the value of a variable for example. Logo uses the keywords `if` and `ifelse`. For example:

```
if :a < 1 [ fd 100 ]
```

If the variable `a` is less than one then go forward.

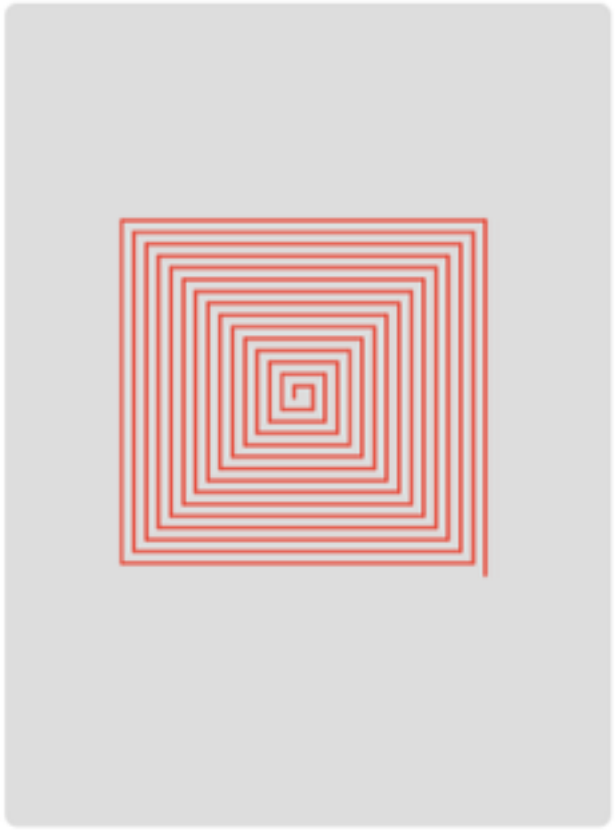
```
ifelse :a >= 1 [ fd 100 ] [fd -100]
```

If `a` is greater than or equal to one then go forward 100 units, otherwise go backwards 100 units.

You can check if variables are equal, less than, less than or equal to, greater than or greater than or equal to using the usual operators: `=`, `<`, `<=`, `>`, `>=`.

You can even combine this with the `stop` statement to choose when to stop a loop or function:

```
to drawspiral :dist
  if :dist>300 [stop]
  fd :dist rt 90
  drawspiral (:dist+5)
end
```



18. Program control continued

Combining recursion with the `stop` statement is a powerful way to create interesting designs. The function `tree` will draw a tree `d` branches high. It is called with an input value of '5' for a tree 5 branches high.

The function goes forward and turns left 20 degrees, then it draws a tree with one fewer branches, turns right 40 degrees and draws another tree with one fewer branches.

```
make "s 80
to tree :d
 if :d=0 [stop]
 fd :s rt 340
 tree (:d-1)
 rt 40
 tree (:d-1)
 rt 160
 fd :s
 rt 180
end
tree 5
```

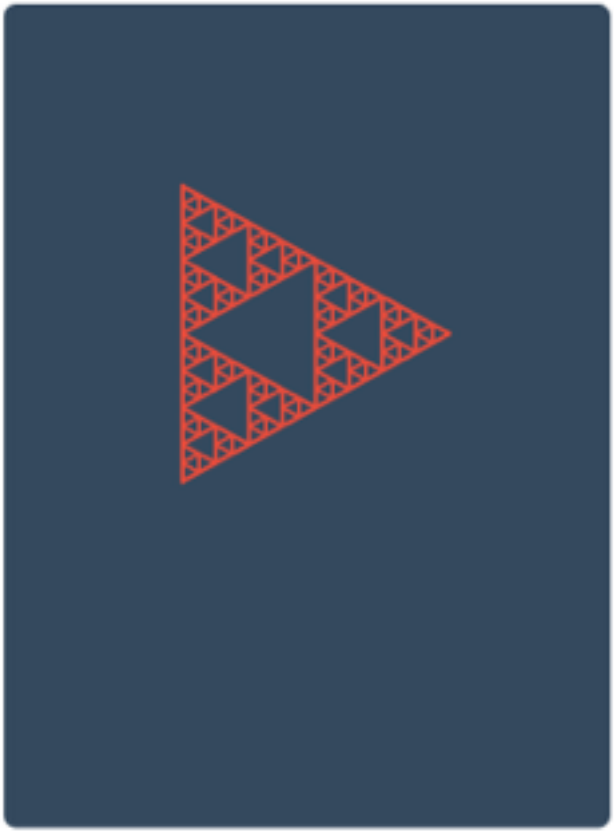
To make that work we need the last three lines of the function - which move the turtle back to the start, facing in exactly the same direction as it started.

19. Program control continued

The tree we drew is an example of a fractal, or a shape demonstrating 'self-similarity'. It consists of smaller copies of itself repeating at each step.

Another famous fractal you can draw using Logotacular is the Sierpinski gasket or Sierpinski triangle, which consists of an equilateral triangle divided into smaller copies of the same triangle recursively.

```
to sier :n :len
if :n=0 [stop]
rpt 3 [sier :n-1 :len/2
fd :len
rt 120
]
end
sier 5 250
```



20. Errors and debugging

Errors are of two kinds: they can be in your code itself or they might only happen when you run your code.

The first kind of errors include those such as missing out a bracket or forgetting to put **end** at the end of a function definition. The second kind of errors don't appear until you press 'Play'. For example, you might try to use a variable or call a function that you haven't defined, or you might divide by zero by mistake.

This program has an error of the first kind:

```
rpt 10 [fd 10
```

The program below has two errors of the second kind. Firstly **b** is not defined, and secondly you cannot divide by **a** since it is zero.

```
make "a 0
fd :b/:a
```

In Logotacular an arrow appears where it thinks the error is. It isn't always 100% accurate, sometimes the error is actually on a previous line which makes the indicated line incorrect.

You can click on the arrow for more information about the error.