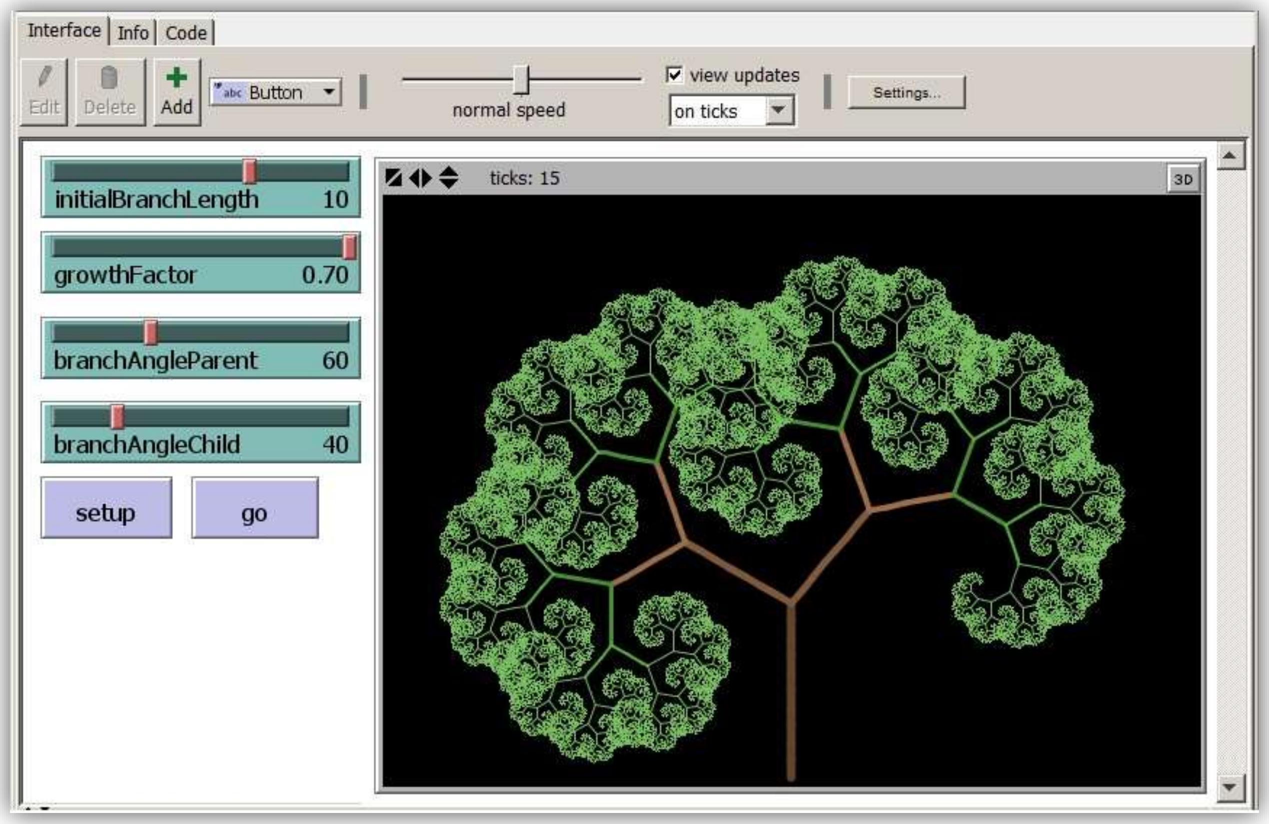
CS108L Computer Science for All
Module 8: Recursion and the Fractal Tree



initialBranchLength 10
growthFactor 0.70
branchAngleParent 60
branchAngleChild 40

Fig 1: Program in its final stage of development.

Model Overview:

In this NetLogo model, the goal is to create a **recursive program** that draws a tree-like **fractal pattern**. We define a recursive program as one that contains a procedure that calls itself! The two images on the next page are famous examples of recursion that occurs outside of computer science: a mirror reflecting itself, and M.C. Escher image of a pair of hands drawing each other. Recursion is a powerful concept that frequently stretches our brains! For more details on recursion, see this week's videos.

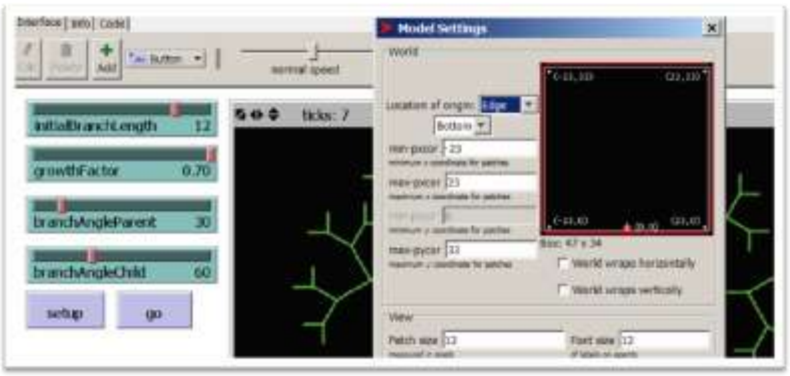
Fig2: “*Holding infinity in the palm of my hand*”, by Samizdat – the title being a reference to a line in the poem “To See a World...” by William Blake.

Fig3: “*Drawing Hands*” is a lithograph by the Dutch artist M. C. Escher first printed in January 1948.

Note: A real world set of mirrors reflecting to infinity is a great way to give a physical interpretation of recursion.

Getting Started:

Before trying to create the complete model, start a new Netlogo program with the sliders and buttons shown. Use the world settings shown below with the origin set to the bottom edge, the min-pxcor set to -23, the max-pxcor set to 23 and the max-pycor set to 33. Our NetLogo world is a rectangle in this Lab!



initialBranchLength 12
growthFactor 0.70
branchAngleParent 30
branchAngleChild 60

Location of origin: Edge
Bottom
min-pxcor -23
max-pxcor 23
max-pycor 33
Patch size 12
Font size 12

Fig 4.

The Basic Structure:

The basic structure of the program contains a minimum of three procedures. You can create more procedures if it helps you structure your programming better. The three procedures are: **setup**, **go** and **grow**. The **grow** procedure is the recursive procedure. So since it is recursive, somewhere within the grow procedure it will call itself!

```
to setup
   ;; clear-all, create turtle, ...
end

to go
  grow initialBranchLength
end


to grow [branchLength]
  ;; do stuff (may be multiple lines of code)

  if (branchLength >= 1)  ;;end recursion when <1
  [
    ask turtles
    [
      ;; do stuff (may be multiple lines of code)
    ]

    tick  ;; update the display before recursive call to "grow"

    grow (branchLength * growthFactor)
  ]
end
```

This program structure contains each of the key parts found in most recursive programs:

1) **Something that gets the recursion started:** In the example above, this is done in the go procedure. Inside this simple procedure we are calling the grow procedure.

   Note that there is a variable name following grow when we call the procedure

```
grow initialBranchLength
```

This allows us to send a value (an input value) into the grow procedure when we call it. The grow procedure needs some number "passed in" to do its work. For example, if the slider is set to 5, the above line really is just grow 5. In this case we are passing in the initial branch length that we set using a slider. If we look at the beginning of the grow procedure below

```
to grow [branchLength]
```

We see that there is a variable name in square brackets following the word grow. That variable name, branchLength, is a local variable that holds whatever number is assigned to it. In the example given, where we have called it with a 5, branchLength is the value 5 everywhere in that procedure. The local variable, branchLength, effectively carries the passed in value (5) to all the places that the variable is used inside the procedure.

Each call to "go" draws one branch that has a length equal to this input value.

2) **The Recursive Call:** On line 24, grow calls itself. Notice that the first call, the one that drew the initial branch is not quite finished. We haven't made it to the end of the procedure before we call grow again!

3) **The End Condition:** Recursive programs need to have some way of ending otherwise they will continue indefinitely! In this case, each time grow is called, it is called with a smaller and smaller input branch length. The recursion only continues when the branch length is greater than or equal to 1 patch (see line 14). Notice: We can only make it all the way through the grow procedure when the local variable, the parameter **branchLength** in this case, is less than 1.

The overall approach to creating this model:

1) Build the basic model that creates a simple branching tree
2) Add changing the branch angle of the parent
3) Add in changing the branch thickness
4) Add changing the branch color

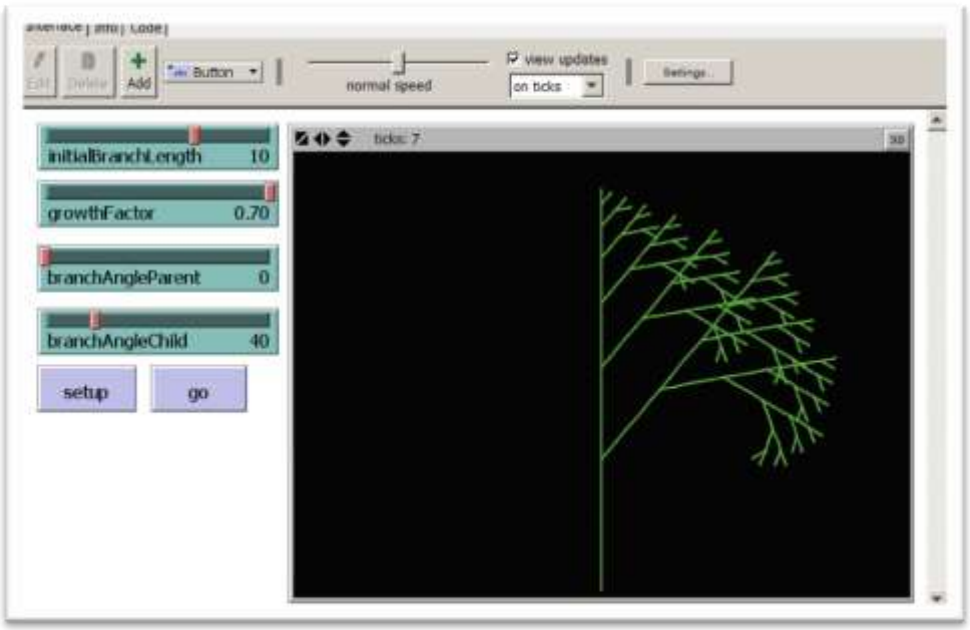
Creating the Simple Branched Tree

Setup:

In the setup procedure you want to start with just one turtle, heading towards the top of the screen with its pen down. You also want to hide the turtle (hide-turtle).

Grow Procedure “Do Stuff”:

The screen capture in Figure 5 shows a run of the model before it is fully complete. If we look at the figure we see that we start with 1 branch, the have 2 branches. Then each branch splits in two so we have 4 branches. After that the 4 branches become 8 branches and so on. Each of these branches are drawn by different turtles. Thus, one turtle becomes 2, 2 becomes 4, 4 become 8, 8 becomes 16, 16 becomes 32, ... Of course, if this does not stop, then it will not be long before there are too many turtles for NetLogo to handle.



initialBranchLength 10
growthFactor 0.70
branchAngleParent 0
branchAngleChild 40

Figure 5

We multiply the turtles by asking the turtles to hatch 1 turtle each time the grow procedure is called. The “`ask turtles`” in the grow procedure then works on both the parent and the baby turtle. Each time the grow procedure is called it will ask ALL the turtles to perform the same commands.

Since there is a picture being drawn, “do stuff” needs to do more than make baby turtles. In particular, it must draw a branch with length equal to the input value of “grow” and it must make a turn or two.

To understand how to make this work, think about the basic structure of Netlogo's "hatch":

```
;;Any changes to a turtle (color, heading, location, etc) made before "hatch" are
;;   inherited by (copied into) the new turtle.

hatch 1
[
   ;;Any changes to a turtle made within a hatch block only effect the
   ;;     new turtle that was just hatched.
]
;;Any changes to a turtle made after a hatch block only effect the parent, not the
;;     turtle just hatched.
```

In the example shown in Figure 5, the first turtle created in the setup, the grandparent of all the turtles, starts at the bottom center and moves straight up the center to near the top. This grandparent of all turtles follows the following rules:

1) It never makes a turn.
2) In each call to "grow" it moves forward one branch length.
3) In each call to "grow" its branch length is shorter than it was in the previous call.
4) In each call to "grow" it has one child who branches off to its right.
5) It stops, or "bottoms out" when its branch length is smaller than one patch (< 1).

Now these rules are the same for all the baby turtles too. So what's true for the great grandparent of all turtles are also true for each of the turtles in each of the following generations!!! EXCEPT for the turtle's direction. Each child turtle starts its life facing a different direction from its parent. It turns right the slider value of "branchAngleChild" to the heading of its parent, and each child's first branch has a length that is the same as the length its parent will branch on its next call to "grow". Otherwise, the life of a child is the same as the live of its parent.

At this point, you should stop reading and get coding. With what is given above, you should be able to create a model that works up through the capture shown in Fig. 5.

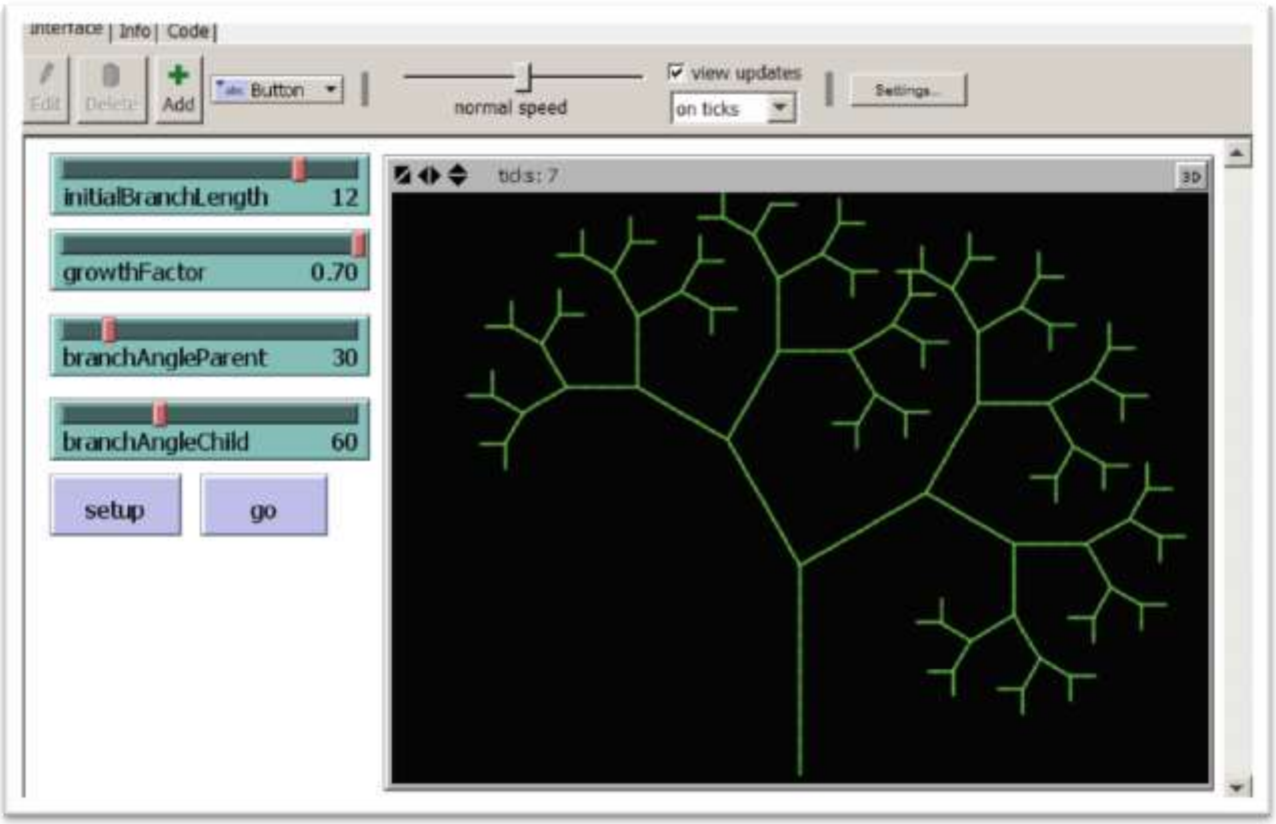
Running With More Generations:

After you get the model above up and running, change the recursion stop condition so that the model runs for more generations. That is, the version above only branches when the branch length is at least one patch. Change this to half a patch, a quarter patch, a tenth patch, …

When should you stop? Well, with each branch, the distance gets smaller, so eventually, the branches will be smaller than a pixel and different branches will blur together – not looking very good. Also, with each branch, the number of turtle's doubles, so each extra branch will take twice the runtime as the previous branch.

Implementing the Branch Angle of the Parent:

In the version above, the parent always went straight. In the final version, the parent should only move straight when the slider value of "branchAngleParent" equals 0. When this slider is set to a different value, the parent should branch left by the specified angle.



initialBranchLength 12
growthFactor 0.70
branchAngleParent 30
branchAngleChild 60

Fig 6

Implementing Changing Branch Thickness:

After all of the above parts are working, implement the branch thickness that is thickest for the "trunk" and gets smaller as the tree branches. There are a number of different ways this can be done. If you like, you may choose to create an additional slider to allow the user to make adjustments to the way line thickness is used. For example, selecting the thickness of the starting line and/or the rate at which the line gets thinner. Another way is to create a global variable, say "**BranchThickness**". If you create a global variable you must initialize it in the "setup" procedure to a value that is large enough so that it would be the trunk, say 7 or so. Also you have to set your pen-size to "**BranchThickness**". Then, as long as thickness is greater than 1 long, (if **BranchThickness > 1**) each time "grow" is called you reduce the thickness of the branch by 1 for all the turtles.

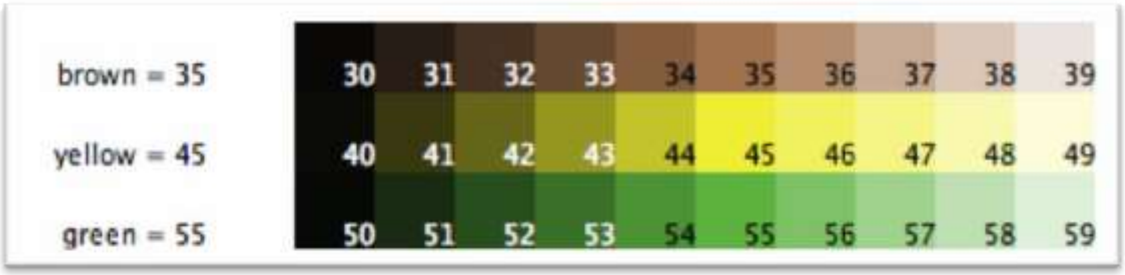
Implementing Changing Branch Color:

The requirements are:

1) All lines of the same length (drawn in the same generation) have the same color.

2) The final 14 or so generation image has at least three different line colors.

As with implementing the changing branch thickness, there are many different ways you may choose to implement these requirements. One way is to use the value of the global "**BranchThickness**" variable created for the branch thickness together with some knowledge of how NetLogo represents colors. In particular, NetLogo's default color table is given in the on-line documentation. In particular, I wanted to use browns and greens, so the part of the table I used was:



brown = 35	30	31	32	33	34	35	36	37	38	39
yellow = 45	40	41	42	43	44	45	46	47	48	49
green = 55	50	51	52	53	54	55	56	57	58	59

A segment of Netlogo's Default Color Table

For the program used to generate Figure 1, when the branch thickness is greater than 3, a shade of brown was used – the thicker the branch, the darker the brown. To get this, set the color to **(39 – BranchThickness)**. This worked since:

1) The largest value used for **BranchThickness** is 7 and 39 – 7 equals 32 – a nicely dark shade of brown.

2) This equation was used when **BranchThickness** was greater than 3, so the lightest color this ever picks is 39 – 4 or 35, which is lighter than the dark brown, but not too light.

When **BranchThickness** is not greater then 3, a similar equation is used to set the color to one of the shades of green.