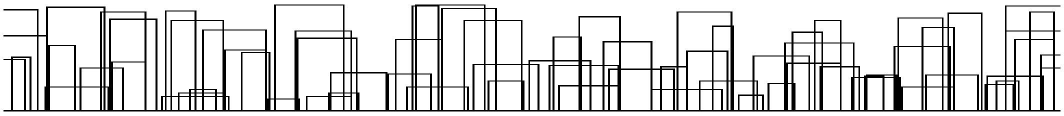


1. Turtles that Climb Trees

Introduction

You probably noticed the Fern branch and Tropical tree drawn on the very first page of this book. Here they are once more:

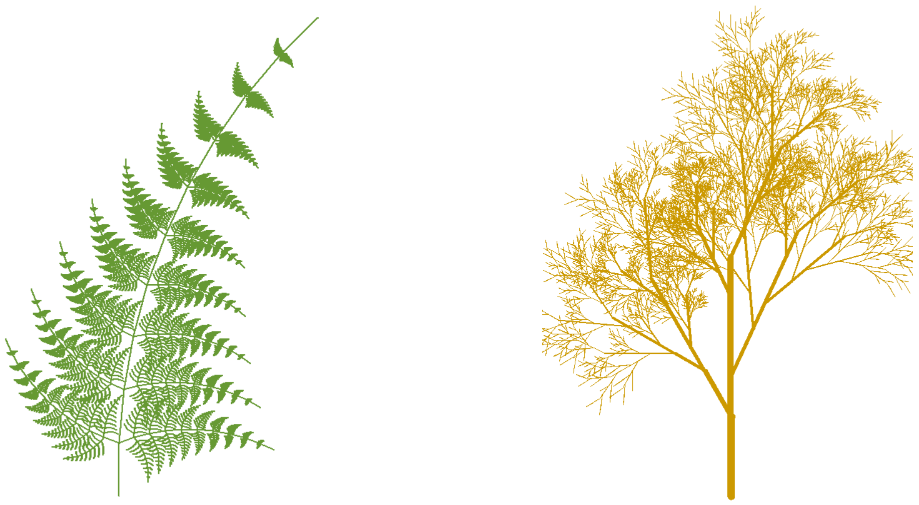


Figure 1-1

Believe it or not, these figures have been drawn using Logo programs – programs containing not more than 15 instructions each! They are based on an interesting idea called *Recursion*.

Continue reading this chapter to learn about *recursion* and how do draw these sort of designs in Logo.

Mystery of Recursion

In order to draw trees like the ones shown above, we will learn a couple of new Logo procedures. But, in addition, we will also explore the interesting and mysterious concept in Programming called *Recursive Procedures*.

Definition of Recursive Procedure:
A recursive procedure calls itself.

The definition is deviously simple! Let's try it out through an example. Below, see an ordinary procedure in which the Turtle just draws a line and turns right.

```
ERASE "Foo
TO Foo :len
FD :len RT 90
END
```

? Foo 100

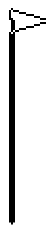


Figure 1-2

To make this procedure *recursive*, we will call it *from within itself*.

```
ERASE "Foo
TO Foo :len
FD :len RT 90
Foo :len
END
```

? Foo 100

When you run this instruction, you will notice two things: (1) A square is drawn; (2) It appears as if Logo is hung – our program never stops!

Do you understand why our program never stops? If we list the events we might understand why:

1. Line 1 of `Foo` draws a line and turns right.
2. Line 2 calls `Foo`.
3. Line 1 of `Foo` draws a line and turns right.
4. Line 2 calls `Foo`.
5. Line 1 of `Foo` draws a line and turns right.
6. Line 2 calls `Foo`.
7. …

As you can see, this series of events will never end. After 4 calls a square will be drawn, and after that the Turtle will keep running over that square indefinitely. What we have is a program that will run forever!

(Note: Press Ctrl-Q (or Alt-S on UCB Logo 6.0) to terminate this program)

Insight: A recursive program runs forever.

Play Time:

Write your favorite procedures and make them recursive. See what happens.

Dissecting Recursion

Let’s now make two small changes (shown in boldface below) to our recursive procedure `Foo` and run it again.

With these changes, you will see the Turtle drawing a pattern (a rectangular spiral) as shown below.

```
ERASE "Foo
TO Foo :len
FD :len RT 90
wait 30
Foo :len+5
END

? Foo 10
```

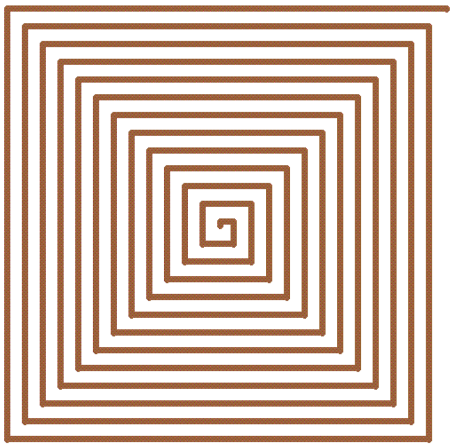


Figure 1-3

The <u>first change</u> is the insertion of the `WAIT` statement *before* the recursive call. `WAIT` causes the program to run in slow-motion, which helps us understand how recursion works.

The `WAIT` procedure asks Logo to simply do nothing for a specified duration.

Using the WAIT procedure:
The basic format of `WAIT` is as follows:

```
? WAIT time
```

The input "`time`" indicates the duration (in 60th parts of a second) during which Logo does nothing, thus delaying execution of subsequent commands.

<u>Example</u>: The following command will cause Logo to pause for one second.

```
? WAIT 60
```

The <u>second change</u> affects `Foo`'s input – it is no longer fixed. We start the program with some value (such as 10), and with every subsequent call to `Foo`, we increase the line length by 5 (from 10 to 15, to 20, to 25, and so on). The `WAIT` command allows us to see this process in slow motion.

Play Time:

1. The turning angle in the spiral above is 90. Try changing it slightly (say to 91 or 89) and see what you get. Also try other angles, say 60 or 120. You might get the patterns shown below:

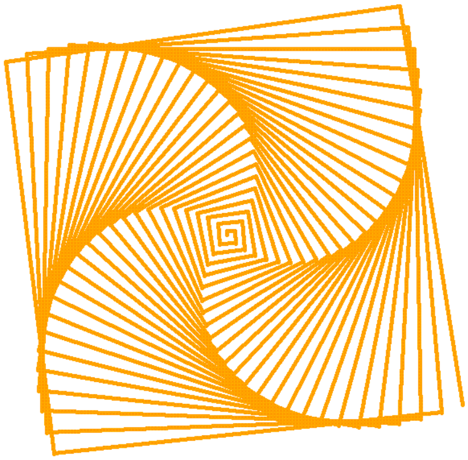


Figure 1-4

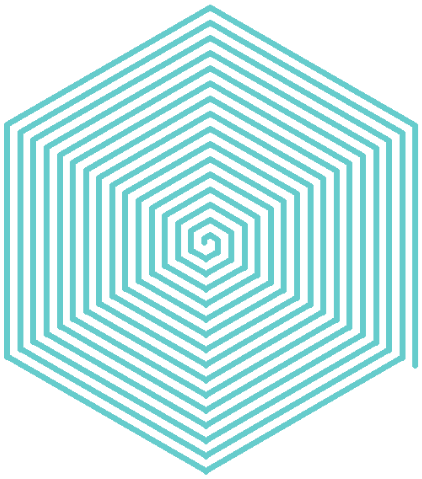


Figure 1-5

2. Write a recursive procedure to draw the pattern shown here. (Hint: Start with the biggest square, and in each recursive call, reduce the size of the square in a *geometric* fashion. For example, each square could be 90% of the previous square.)

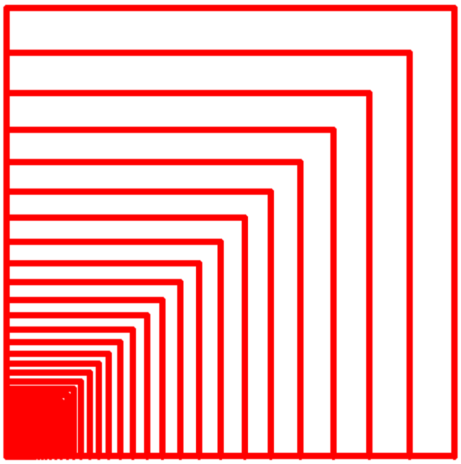


Figure 1-6

IF Infinity Then What?

Writing a program that never terminates is interesting but not very convenient. We would like to write recursive programs that do interesting things and terminate (i.e. stop) when their job is done.

For this purpose, we will learn a new Logo procedure called `IF`.

The `IF` procedure allows a program to ask (itself) a **yes**/**no** question and then take action depending on the answer. The question can only have two possible answers: **Yes** or **No** (or **True** or **False**). If the answer is **Yes**, `IF` goes ahead and runs the instructions in its second input (which is a list); otherwise the program just moves on to the next instruction.

Example: The following procedure will print "Input is too small" if the input is less than 100. Otherwise nothing will be printed.

```
ERASE "iftest
TO iftest :size
IF (:size < 100) [PRINT [Input is too small]]
END
```

```
? iftest 50
Input is too small
? iftest 150
?
```

Insight: The YES/No question in IF allows a Logo program to *conditionally* run a set of instructions. That is why we call the question itself as a *condition*.

Using the IF procedure:

The basic format of `IF` is as follows:

```
IF (condition) [list of instructions]
```

The input "`condition`" is a YES/NO question. The second input is a list of Logo instructions.

In other words, the `IF` command examines a *condition*, and if the condition is TRUE, it runs the instructions supplied in its list input.

In our recursive `Foo` procedure, we could insert the `IF` instruction.

```
ERASE "Foo
TO Foo :len
IF (:len > 200) [STOP]
FD :len RT 90
wait 30
Foo :len+5
END
```

The `IF` in `Foo` asks the question: "Is 'len' greater than 200?" If the answer is YES, Logo runs the `STOP` command. If the answer is NO, Logo moves on to the next instruction.

```
? CS Foo 10
```

If you run this program, it will stop when 'len' becomes greater than 200.

The STOP command:

This command stops the currently running procedure and returns to the calling procedure.

- Self-study: The STOP command only causes the current procedure to stop and return to the calling procedure. How does then our entire program stop?

Controlling recursion using the concept of Level:

There is another way to make recursion finite – that is, make it stop after some time. We can simply decide how *deep* the recursion should go. See the modified Foo procedure below:

```
ERASE "Foo
TO Foo :len :level
IF (:level > 50) [STOP]
FD :len RT 90
Foo :len+5 :level+1
END

? Foo 10 1
```

Trace: 'level' is 1 when we call Foo. It becomes 2 when Foo is called the 2nd time. It becomes 3 when Foo is called the 3rd time. And so on. The IF command will cause this recursive chain to terminate when level become 51.

Similar to before, the recursive call increases the length; and it also increases the value of 'level' by 1. Instead of checking the length, we check 'level' – which is an indication of how deep the recursion is – and when it reaches 50, we ask the program to stop recursing any further.

Play Time:

Modify your earlier recursive procedures such that they do not run forever. Use the IF command that will make them stop after doing some work.

Multiple Recursive Calls

See a simple recursive design below:

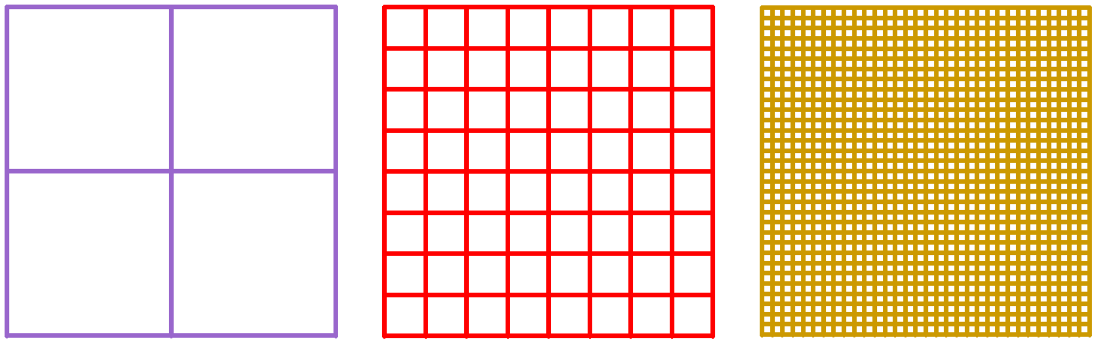


Figure 1-7

The design itself is quite simple, and can be created using other methods (i.e. using `REPEAT`), but it would be fun to draw it using recursion.

The idea is to draw 4 squares in a paned-window pattern. Using recursion, we can then go to each square and again draw the 2x2 pattern – one that fits exactly – inside it. This process can go on as *deep* as we wish. The program is shown below.

First, the ordinary non-recursive procedure that draws a 2x2 pattern.

```
ERASE "sq
TO sq :size
REPEAT 4 [square :size RT 90]
END
```

Now, the recursive procedure: we will use the jump procedure to jump to the center of each square and make the recursive call. As before, 'level' decides when to stop the recursion.

```
ERASE "sq
TO sq :size :level

;Instruction to check when to stop recursion
if (:level < 1) [stop]

REPEAT 4 [square :size RT 90] ;Draw the 2x2 pattern

jump :size/2 :size/2   ;Center of upper-right square
sq :size/2 :level-1    ;Recursive call
jump -:size/2 -:size/2 ;Back to the original place

jump :size/2 -:size/2  ;Center of lower-right square
sq :size/2 :level-1    ;Recursive call
jump -:size/2 :size/2  ;Back to the original place

jump -:size/2 :size/2  ;Center of upper-left square
sq :size/2 :level-1    ;Recursive call
jump :size/2 -:size/2  ;Back to the original place

jump -:size/2 -:size/2 ;Center of lower-left square
sq :size/2 :level-1    ;Recursive call
jump :size/2 :size/2   ;Back to the original place

END
? CS sq 200 1       ;The first one in Figure 1-7
? CS sq 200 3       ;The second one in Figure 1-7
? CS sq 200 5       ;The third one in Figure 1-7
```

Important: Note above that after each recursive call, the Turtle returns to its original position (as noted in the comments of the program).

Insight: In a recursive program, we can get different designs by controlling the depth of the recursion.

Logo Challenge:

- The figure below shows a recursive design. It's called the "Sierpinski Triangle" named after the Polish mathematician Waclaw Sierpinski. Using the concept of 'level', we have shown how this triangle would appear at 3 different levels. Can you work out the Logo program?

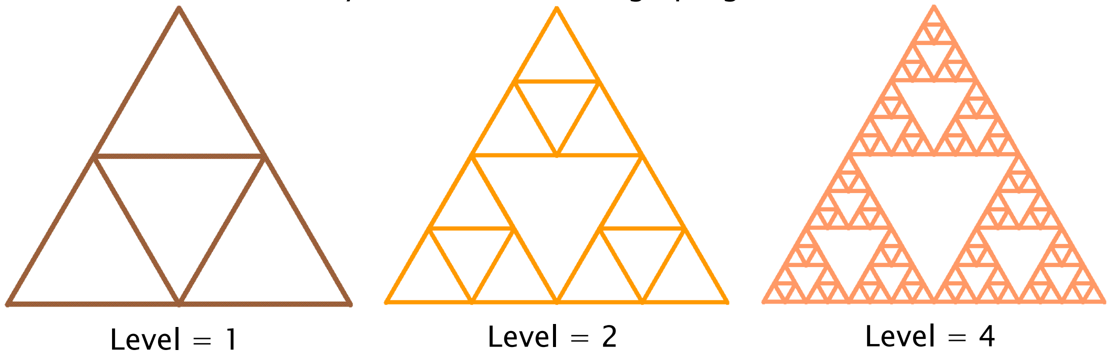
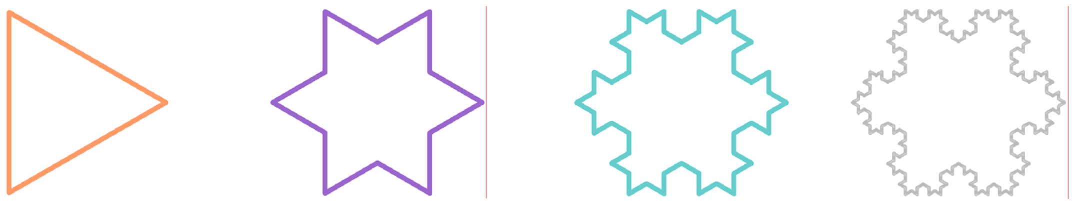


Figure 1-8

- The following series shows another interesting use of recursion – creation of snowflakes! Observe the various levels carefully and work out the recursive procedure.



Recursive Trees

Ok, after all this hard work and learning, we now have the required arsenal to take on the challenge of using recursion to draw trees.

A tree, as you know, is basically a network of branches. We will see if we can take a simple branch structure and use recursion to build a tree.

See the Y branch below:

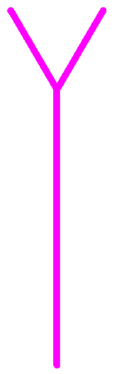


Figure 1-9

This 'basic tree' can be drawn using the following procedure:

```
Erase "tree
To tree :size
FD :size
LT 30 FD :size/2 BK :size/2 RT 30
RT 30 FD :size/2 BK :size/2 LT 30
BK :size
End
```

The <u>very important</u> part to remember about drawing such basic patterns for recursive trees is to *always* return the Turtle to the base (original position) and not leave it dangling on the tree somewhere!

Now, using recursion, we will attach similar patterns at the end of each branch of the Y. But, we will reduce 'size' for the recursive call. To avoid an infinitely running program, we will add a STOP condition based on 'level'. That way, we can control how *deep* the tree should be drawn.

```
Erase "tree
To tree :size :level
IF (:level < 1) [stop]
FD :size
LT 30 FD :size/2 tree :size/2 :level-1 BK :size/2 RT 30
RT 30 FD :size/2 tree :size/2 :level-1 BK :size/2 LT 30
BK :size
End
```

Let's now play with this program and try different values of 'level'.

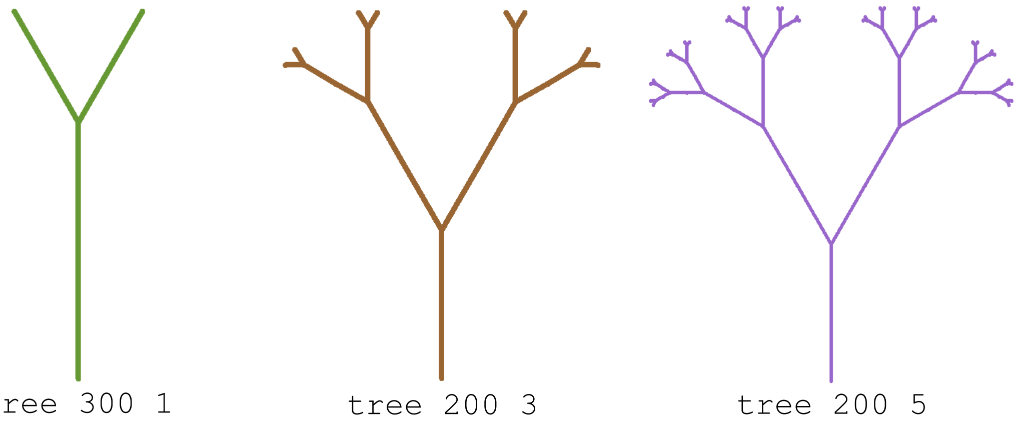


Figure 1-10

Insight: To write a proper recursive tree procedure, you must (1) Return the Turtle to its starting position, and (2) Insert recursive calls at places where you want branches to grow (usually at end-points).

Play Time:

We used certain proportion for each branch in the tree above, and also a certain amount of reduction in size each time the recursive call was made. These parameters obviously determine the appearance of the tree. Modify these values and see what sort of trees you get.

It’s a Forest out there

We really are done with learning how to use recursion to draw trees. If you have completely understood how recursive calls help the Turtle climb trees – in a manner of speaking – you are now well on your way to becoming a certified *tree artist*. All you need to do is *experiment with the technique*.

For example, the tree above looks very symmetric. Real trees (with some exceptions, of course) are actually quite asymmetric. How will you draw an asymmetric tree?

The answer is in simply redesigning our basic pattern. Here is an example of a quite different basic pattern. (Aren’t we creative geniuses?)

```
ERASE "tree
TO tree :s
FD :s/3
LT 30 FD :s/15 BK :s/15 RT 30
FD :s/6
RT 25 FD :s/15 BK :s/15 LT 25
FD :s/3
LT 25 FD :s/15 BK :s/15 RT 25
FD :s/6
RT 25 FD :s/15 BK :s/15 LT 25
BK :s
END
```

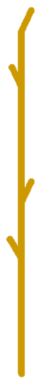


Figure 1-11

? tree 200

Granted that it looks rather ugly, but wait till we apply recursion on it. We will insert 4 recursive calls – one each at the 4 endpoints. We will reduce ‘size’ in each recursive call. The stop condition is based on the concept of ‘level’. Here is the recursive tree procedure. Changes are shown in bold.

```
ERASE "tree
TO tree :s :l
if (:l < 1) [stop]
FD :s/3
LT 30 FD :s/15 tree :s*2/3 :l-1 BK :s/15 RT 30
FD :s/6
RT 25 FD :s/15 tree :s/2 :l-1 BK :s/15 LT 25
FD :s/3
LT 25 FD :s/15 tree :s/2 :l-1 BK :s/15 RT 25
FD :s/6
RT 25 FD :s/15 tree :s/2 :l-1 BK :s/15 LT 25
BK :s
END
```

Let’s now play with this program and try different values of ‘level’.

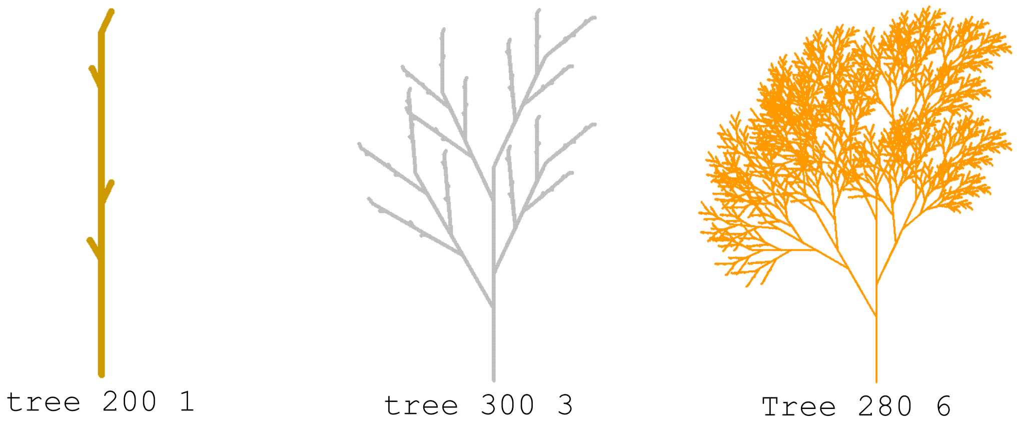


Figure 1-12

Play Time:

1. You can improve the tree above by making the *thickness* of the branches *decrease* as you go deeper in the recursion. You can achieve this effect by supplying an additional input for *pen thickness*, which should be reduced before passing on to the recursive call. With this idea, you can get a tree which would look as shown. Write a program for this tree.

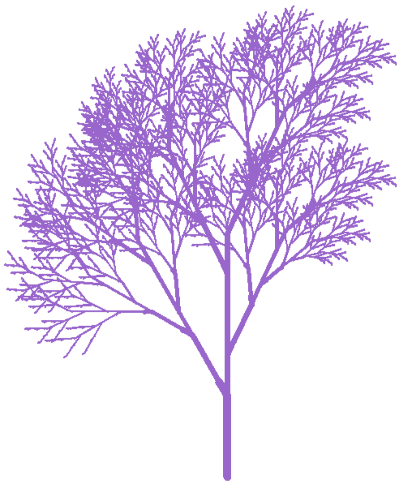


Figure 1-13

2. Write a recursive procedure to draw a symmetric tree (a sort of Christmas tree) as shown here. Use all ideas we have learnt so far, including the idea of decreasing branch thickness.

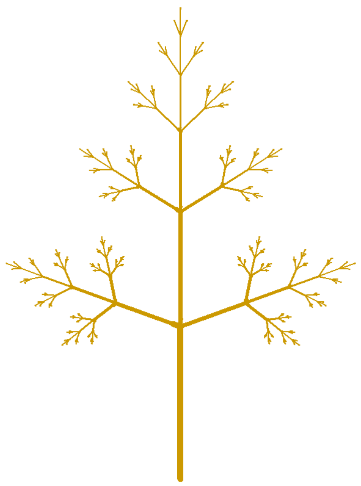


Figure 1-14

3. You can make the asymmetric tropical tree above even more interesting if you use `RANDOM` in choosing the size of its branches. Experiment with this idea.

4. Try the Fern plant recursive procedure. The following figure shows 3 different levels of this program.

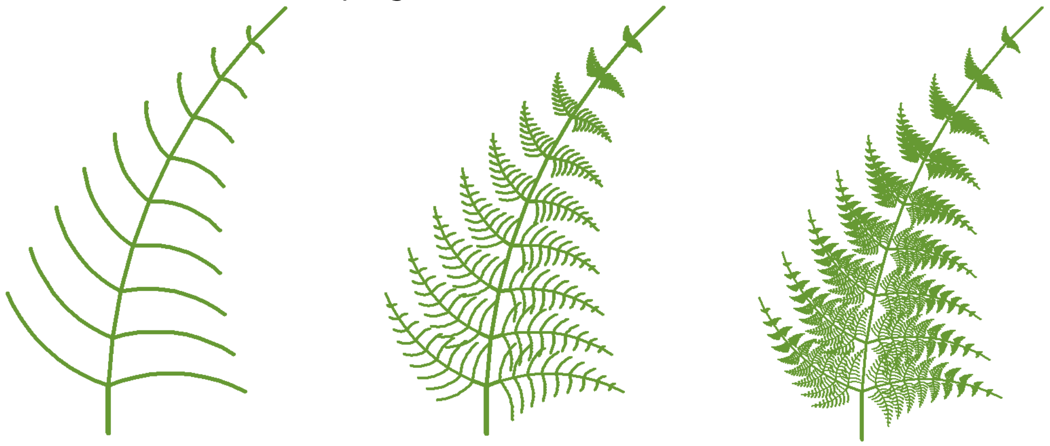


Figure 1-15

Logo Challenge

The following trees actually look different every time you call the instruction (even if all inputs are the same), and they also show fruits at some places. (Obviously, we have used RANDOM to make things happen differently every time.) Write a program to draw such trees.

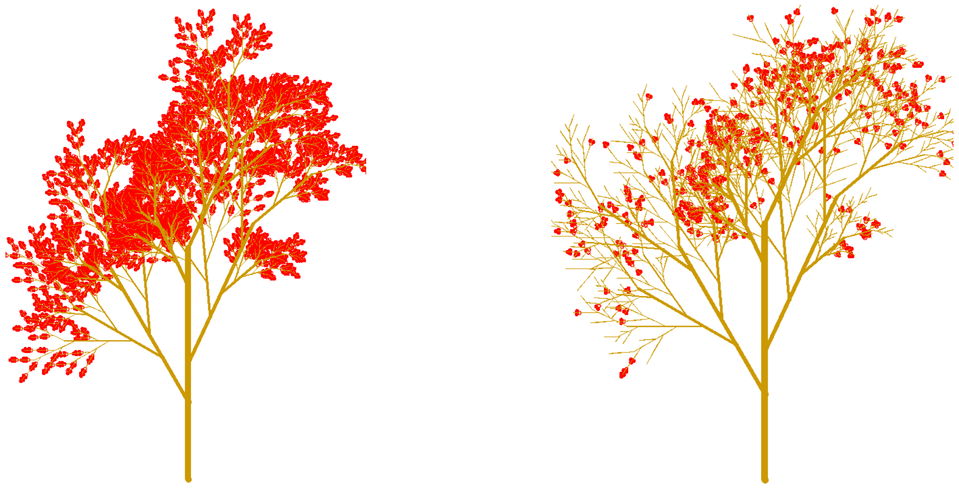


Figure 1-16

Key Observations

- Recursion is a deceptively simple and tricky idea.
- Sometimes you have to press Ctrl-Q (or Alt-S on UCB Logo 6.0) many times to stop an infinite loop.
- Some parameter(s) of your recursive procedure must change in a recursive call to make recursion meaningful and useful.
- The condition supplied to `IF` must be achievable; otherwise the program will never stop.
- A common bug in recursive programs is not moving the Turtle back to its original position after the recursive call.

What We Learnt in this Chapter

1. Recursion is the idea of a procedure calling itself.
2. `WAIT` helps in watching recursion work in slow motion.
3. `STOP` is a useful command to stop the current procedure and make it return immediately to the caller.
4. The `IF` command examines a condition, and if the condition is TRUE, `IF` runs the instructions supplied in its list input.
5. To control recursion, i.e. to make it finite, `IF` should be used to examine a condition that will eventually become TRUE. Using a 'level' input is a simple technique to achieve this.
6. A procedure can have multiple recursive calls.
7. A tree, with its network of branches, can be drawn using Recursion. The main challenge is to get the basic pattern worked out. The next step is to insert recursive calls at the proper points in the basic pattern.
8. The idea of recursion can be applied even to gradually reduce the thickness of branches.
9. The stop condition can be used to do something only at the terminal phase of the recursion, such as, adding a fruit at the end of a branch.

Suggestions to Teachers/Parents

Recursion is an extremely powerful idea. It will require a lot of exploration to really appreciate its power and to understand how best to utilize it. Think of patterns and designs (or look up on the Net) that can be drawn using recursion, and give them to the children for practice. There is a popular branch of Geometry called Fractals which relies heavily on recursion. Children would love to try out some of the fractal patterns.

Solutions to Selected Problems

Rectangular Spiral:

```
ERASE "r_spiral
TO r_spiral :len :step :t
FD :len RT 90 WAIT 60
r_spiral :len+:step :step :t
END
```

```
? CS r_spiral 5 10 500
```

Press Ctrl-Q (Alt-S on UCB Logo 6.0) to stop the program.

Tunnel of Squares:

```
ERASE "square
TO square :size
REPEAT 4 [FD :size RT 90]
END
```

To make this procedure *recursive*, we will call it *from within itself*.

```
ERASE "rsquare
TO rsquare :size
REPEAT 4 [FD :size RT 90]
wait 30
rsquare :size
END
```

Next, we will change the recursive call to provide a slightly smaller 'size' input.

```
ERASE "rsquare
TO rsquare :size
REPEAT 4 [FD :size RT 90]
wait 30
rsquare :size * 0.9
END
```

```
? rsquare 100
```

The program starts with a square of size 100, but with every subsequent call to `square`, we get smaller squares until they become so small we can't see them. The `wait` command allows us to see this process in slow motion.

Tree with variable thickness:

We can achieve this effect by supplying an additional input for pen thickness, which should be reduced before passing on to the recursive call. Setpensize requires its input to be a whole number. We can ensure this using the Logo procedure `ROUND` which takes a fractional number and rounds it to its nearest whole number. For example, `ROUND 2.6` is `3`; `ROUND 4.1` is `4`, and so on. Also, note that, you must *reset* the thickness at the end of the procedure, since it is changed in the recursive calls.

```
ERASE "tree
TO tree :s :l :t
if (:l < 1) [stop]
setpensize round :t
FD :s/3
LT 30 FD :s/15 tree :s*2/3 :l-1 :t*0.7 BK :s/15 RT 30 FD :s/6
RT 25 FD :s/15 tree :s/2 :l-1 :t*0.7 BK :s/15 LT 25 FD :s/3
LT 25 FD :s/15 tree :s/2 :l-1 :t*0.7 BK :s/15 RT 25 FD :s/6
RT 25 FD :s/15 tree :s/2 :l-1 :t*0.7 BK :s/15 LT 25 BK :s
setpensize round :t
END
```

Christmas Tree:

This one is symmetric. We simply have to decide how many branches to draw. At each successive level, we go on reducing the angle of the branch.

The algorithm is: 1) draw the branch, 2) turn left and draw the left sub-tree, 3) turn right (i.e. become straight again), and draw the middle sub-tree, 4) turn right and draw the right sub-tree, 5) turn left (i.e. become straight again), and return to the original position.

The basic pattern would be:

```
Erase "ctree
TO ctree :s :ang
FD :s
LT :ang FD :s/20 BK :s/20
RT :ang FD :s/20 BK :s/20
RT :ang FD :s/20 BK :s/20
LT :ang
BK :s
End
```

`? ctree 200 70`

With the stop condition, 3 recursive calls, and input for thickness:

```
Erase "ctree
TO ctree :s :ang :l :t
IF (:l<1) [STOP]
setpensize round :t
FD :s
LT :ang FD :s/20 ctree .4*:s :ang-12 :l-1 :t*.7 BK :s/20
RT :ang FD :s/20 ctree .7*:s :ang-12 :l-1 :t*.7 BK :s/20
RT :ang FD :s/20 ctree .4*:s :ang-12 :l-1 :t*.7 BK :s/20
LT :ang
BK :s
setpensize round :t
End
```

```
? ctree 250 70 1 6       ;First one in Figure 1-10
? ctree 250 70 3 8       ;Second one in Figure 1-10
? ctree 250 70 6 8       ;Third one in Figure 1-10
```

Fern Tree:

```
Erase "ftree
TO ftree :s :l :t
IF :l<1 [STOP]
SETPENSIZE ROUND :t
REPEAT 10 [
  FD :s/10
  RT 70 ftree 0.33*:s*(1-REPCOUNT/10) :l-1 :t*.7 LT 70
  LT 70 ftree 0.33*:s*(1-REPCOUNT/10) :l-1 :t*.7 RT 70
  RT 5
]
PU REPEAT 10 [LT 5 BK :s/10] PD
SETPENSIZE ROUND :t
END
? CS ftree 300 2 6  ;First one in Figure 1-15
? CS ftree 300 3 6  ;Second one in Figure 1-15
? CS ftree 300 4 6  ;Third one in Figure 1-15
```