Recursive Structures and Processes

Curran Kelleher

July 7, 2007
Chapter 1

Recursive Structures and Processes

"Every computer program is a model, hatched in the mind, of a real or mental process. These processes, arising from human experience and thought, are huge in number, intricate in detail, and at any time only partially understood. They are modeled to our permanent satisfaction rarely by our computer programs. Thus even though our programs are carefully handcrafted discrete collections of symbols, mosaics of interlocking functions, they continually evolve: we change them as our perception of the model deepens, enlarges, generalizes until the model ultimately attains a metastable place within still another model with which we struggle." - Alan J. Perlis

1.1 Computer Programming

The above quote is from the forward to the book Structure and Interpretation of Computer Programs (SICP) by Harold Abelson and Gerald Jay Sussman. This book has become a highly revered text in the field of computer science. Alan Perlis sums up quite nicely the nature of the beast: computer programming is amazing, but we can never achieve the perfection that we instinctively seek.

As Perlis says, computer programs are “mosaics of interlocking functions.” When a function calls itself, it is called recursive. The nature of programming is itself recursive. Programming languages are defined by grammars which are recursive, which is why programming has infinite possibilities. On a higher level, the programmer is always seeking to abstract and generalize essential pieces, so that they can be re-used in different contexts
instead of re-written. The programmer tries to repeat this process on the
new code (recursion), always attempting to find the most elegant solution
to the problem at hand. Eventually the program arrives at a “metastable
place within still another model with which we struggle.” If and when in
the future that model itself is transcended, the new model will be inside yet
a higher meta-model (recursion), and so on. The higher up this pyramid of
models, the more perfect the program seems. However, this pyramid has no
end. This is why perfection is impossible.

1.2 Examples

We will learn about recursion by example, using the Groovy programming
language (yes, it really is Groovy). These examples are intended to be
investigated by interested people, and often in their construction efficiency
has been sacrificed for clarity. You will get a sense of how they work, but you
will probably not fully grok them unless you rewrite them yourself. (grok: to
understand something so well that it is fully absorbed into oneself) Nothing
is fully grokked unless you do it yourself, so I encourage you to re-implement
these examples. Make them better, faster, interactive, more interesting,
more general, more beautiful, and share your discoveries with the world.

Computer programming is complex, no doubt, and if you are unfamiliar
with programming you may initially feel overwhelmed by these examples.
The purpose of these examples is to teach you about recursion, so the most
important thing to keep in mind is not the details of each example, but
rather the common theme of all of them - recursion.

1.2.1 Factorial!

We will begin with the example of the factorial, denoted “!”.

\[
3! = 3 \times 2 \times 1 \quad (1.1) \\
5! = 5 \times 4 \times 3 \times 2 \times 1 \quad (1.2) \\
X! = X \times (X - 1) \times (X - 2) \times (X - 3) \times ... \times 3 \times 2 \times 1 \quad (1.3) \\
0! = 1 \quad (1.4)
\]

So how can we compute this!? Here is a recursive function that does it:
First a bit of code explanation. The keyword `def` means define. In this case, we are using it to define a function. `def` is also used to define variables. The `(n)` means that the function takes a single argument, and inside the function it will be referred to as `n`. The statement `for(i in 1..n)` means to do something `n` times, giving the variable `i` the current value (0, 1, 2, 3, ... n-1, n). The keyword `return` returns a value from the function. The statement `factorial(i)` calls the function `factorial`, passing whatever is in the variable `i` as the argument (which is referred to as `n` inside the function). Surrounding text with "" makes it a string literal. The + operator, when applied to strings, concatenates them together (adds the second string to the end of the first one). The function `println` prints a string to the console.

Lets look at the algorithm. It evaluates the recursive definition of !, which is $n! = (n-1)!$ for $n > 1$, and $n! = 1$ for $n \leq 1$. If the function were defined only as `return n * factorial(n-1)`, then the function would call itself infinitely and never return. It must “bottom out” in order to do anything. This is true for all recursive functions. In this case, the function bottoms out when `n` is $n \leq 1$ - it returns 1.

When this program is executed, we get the following output.

```
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(6) = 720
factorial(7) = 5040
factorial(8) = 40320
factorial(9) = 362880
factorial(10) = 3628800
```

1.2.2 Fibonacci Numbers

The Fibonacci sequence can be generated by a recursive function very similar to the recursive factorial function. The only difference is the rule, which
for the Fibonacci sequence is \( f(n) = f(n - 1) + f(n - 2) \) for \( n > 1 \), and \( f(n) = 1 \) for \( n \leq 1 \). This kind of recursive definition appears frequently in mathematics, and is also called a recurrence relation.

```python
def fibonacci(n):
    if (n > 1):
        return fibonacci(n-1) + fibonacci(n-2)
    else:
        return 1

for i in range(10):
    print "%s\n" % fibonacci(i)
```

The output of this program is the Fibonacci sequence:

- \( \text{fibonacci}(0) = 1 \)
- \( \text{fibonacci}(1) = 1 \)
- \( \text{fibonacci}(2) = 2 \)
- \( \text{fibonacci}(3) = 3 \)
- \( \text{fibonacci}(4) = 5 \)
- \( \text{fibonacci}(5) = 8 \)
- \( \text{fibonacci}(6) = 13 \)
- \( \text{fibonacci}(7) = 21 \)
- \( \text{fibonacci}(8) = 34 \)
- \( \text{fibonacci}(9) = 55 \)
- \( \text{fibonacci}(10) = 89 \)

Do you find it curious that two things that on the surface seem vastly different are in fact almost the same?

### 1.2.3 Recursive Transition Networks

On page 132 of Gödel, Escher, Bach, you will find diagrams representing two recursive transition networks (RTNs), which define a grammar. When this grammar is iterated through randomly, it can generate sentences. Here is a program which carries out those RTNs, choosing transitions randomly whenever there are several options.
Here is some sample output:

small small bagel inside the strange cow
a small cow that small great cow gobbled
the horn
cow
large small bagel that runs small large horn
small horn
strange bagel with horn

1.2.4 A Tree Fractal

Now we’ll begin to generate pictures using recursion. This how fractals are generated. This program grows a tree recursively, by adding two branches to the end of the current branch until it bottoms out. Bottoming out happens when the tree has branched a certain number of times. The size of the two new branches relative to the current one is determined by the `sizeFactor` variable. The angle at which the two new branches will branch out relative to the current one is determined by the `angleFactor` variable. The height of the first tree
is determined by \textit{trunkHeight}, and the number of levels of recursion (the number of times the tree branches into new trees before it bottoms out) is determined by \textit{depth}.

```java
class Tree extends JVDrawingPanel{
  def angleFactor = Math.PI/4
  def sizeFactor = 0.58
  def depth = 11
  def trunkHeight = 0.4

  Tree(){
    growTree(0.5, 0, trunkHeight, Math.PI/2, depth)
  }

  def growTree(x1, y1, rootLength, rootAngle, depth){
    def x2 = x1 + Math.cos(rootAngle) * rootLength
    def y2 = y1 + Math.sin(rootAngle) * rootLength
    add(new JLine(x1, y1, x2, y2))
    if(depth>0){
      growTree(x2, y2, rootLength*sizeFactor, rootAngle+angleFactor, depth-1)
      growTree(x2, y2, rootLength*sizeFactor, rootAngle-angleFactor, depth-1)
    }
  }
}
JV.createWindow(new Tree())
```

First a bit of code explanation. A Java visualization library provides the drawing API (application programming interface) which includes \textit{JVDrawingPanel} and \textit{JLine} to draw the lines, and the function \textit{JV.createWindow()}, which creates and displays a window on the screen. This library is used instead of directly using Java’s graphics API so that the code focuses mainly on the algorithm, and is not cluttered with the peripheral details of graphics and user interface code.

Branching structure (like in this tree) exists in most plants, meaning that a recursive algorithm is executing inside plants as they are growing. Aristid Lindenmayer noticed this, and developed the notion of a L-system or Lindenmayer system, which can algorithmically generate virtual plants. Brian Goodwin explores somewhat how these recursive branching algorithms actually work at the molecular level in plants and other organisms in his book “How the Leopard Changed Its Spots.”

How many branches are there for a given depth?

### 1.2.5 The Koch Snowflake

The Koch Snowflake (also called Koch Curve, or Koch Star) first appeared in a paper by Swedish mathematician Helge von Koch. The Koch Snowflake has finite area but infinite perimeter. Coastlines exhibit a similar property. If randomness is introduced to the generation of the koch curve, the curves that are generated resemble coastlines or the cracks in rocks or
pavement. When this randomized Koch curve is generalized into 3 dimensions, fractal surfaces are formed that resemble real mountains.

// (cx,cy) // * // /| \ // / h \ // / | \|---d---| // *.*.*. * *.*.*. //(x1,y1) (ax,ay) (bx,by) (x2,y2)

class Koch extends JVDrawingPanel{
  def depth = 6
  Koch(){ createCurve(0.02, 0.5, 0.96, 0, depth) }

  def createCurve(x1, y1, rootLength, rootAngle, depth){
    def xHat = Math.cos(rootAngle), yHat = Math.sin(rootAngle)
    def dx = xHat * rootLength, dy = yHat * rootLength

    if(depth > 0){
      def ax = x1+dx*1/3, ay = y1+dy*1/3
      def bx = x1+dx*2/3, by = y1+dy*2/3
      def d = rootLength/3
      def h = Math.sqrt(3)/2*d
      def cx = x1+dx/2-yHat*h, cy = y1+dy/2+xHat*h

      createCurve(x1,y1,d,rootAngle,depth-1)
      createCurve(ax,ay,d,rootAngle+Math.PI/3,depth-1)
      createCurve(cx,cy,d,rootAngle-Math.PI/3,depth-1)
      createCurve(bx,by,d,rootAngle,depth-1)
    }
    else
      add(new JVLine(x1, y1, x1+dx, y1+dy))
  }
}

JV.createWindow(new Koch())

1.2.6 Iterated Function Systems

Iterated Function Systems (IFS) take a single point and move it around repeatedly, plotting a point on the screen for each move. The point is moved according to a mapping function, which is chosen probabilistically from several possible mapping functions. The Sierpinski Triangle and Fern are notable IFS examples. Electric Sheep, the famous evolving fractal screen saver developed by Scott Draves, uses an IFS with non-linear mapping functions (with some other fancy tricks) to generate it’s beautiful images.

The Sierpinski Triangle

The Sierpinski Triangle (or Sierpinski Gasket) was described by Wacław Sierpiński in 1915. The Sierpinski Triangle can be obtained a number of
different ways. Pascal’s Triangle, when the odd numbers are colored, approaches the Sierpinski Triangle as it is expanded infinitely.

Fern

The fern IFS is a system of four mapping functions. Each of these functions is a mapping from the outermost rectangle to another, smaller rectangle.
Figure 1.1: The rectangles used in the coordinate transformations (approximately), and the image generated by our program.

The four mapping functions map from any point inside the outermost black rectangle

1. to a point on the green part of the stem (1% of the time)
2. to a point in the red rectangle, the lowest left branch (7% of the time)
3. to a point in the dark blue rectangle, the lowest right branch (7% of the time)
4. to a point in the light blue rectangle, which spirals everything upwards and smaller (85% of the time)
Is this algorithm recursive? The code itself is not recursive, it is just repetitive. However, the mappings are recursive, because they are applied to themselves eventually. The filling in of the fractal object happens because of the randomness introduced by selecting probabilistically which of the four mappings to apply.

### 1.2.7 The Mandelbrot Set

The Mandelbrot Set is probably the most famous fractal of all. It is defined by the equation $z = z^2 + c$, where $c$ is the starting point in the complex plane, and $z$ is initially zero. This function is iterated many times. If $z$ eventually (after many iterations) “escapes” the circle (of radius 2 centered...
at the origin), then the initial point, $c$, is not in the Mandelbrot set, and that point gets assigned a color based on the number of iterations it took for $z$ to escape. If $z$ never escapes the circle, then it is in the Mandelbrot set, and is colored black. We assume that if $z$ is still inside the circle after $\text{maxIterations}$ iterations, then it will never escape (so $c$ is in the set). This assumption is not always valid, but we must make it to avoid infinite looping.
1.3 Recursion is Everywhere

Recursion is all around us. It is in our computers, in our cells, in the plants we eat, in our brain, in the land we walk on, in Gödel’s incompleteness theorem, in Escher’s art, and in Bach’s music.

```java
class Mandelbrot extends ImagePanel{
    def z = new ComplexNumber(0,0)
    def c = new ComplexNumber(0,0)
    def maxIterations = 7
    def drawMandelbrot(){
        window.set(-2,2,-2,2)
        for(xPixel in 0..window.width){
            for(yPixel in 0..window.height){
                c.real = window.getXValue(xPixel)
                c.imaginary = window.getYValue(yPixel)
                def iterations = calculateIterations(c)
                def color = calculateColor(iterations)
                image.fillPixel(xPixel,yPixel,color)
            }
        }
    }

    int calculateIterations(ComplexNumber c){
        z.real = z.imaginary = 0
        def iterations = 0
        while(circleContains(z) && iterations < maxIterations){
            z = z*z+c
            iterations++
        }
        return iterations
    }

    boolean circleContains(ComplexNumber z){
        Math.sqrt(z.real*z.real+z.imaginary*z.imaginary)<2
    }

    Color calculateColor(int iterations){
        if(iterations == maxIterations)
            return Color.black
        else{
            int red = (iterations*3+200)%255
            int green = (iterations*4)%255
            int blue = (iterations*5)%255
            return new Color(red,green,blue)
        }
    }
}

def mandelbrot = new Mandelbrot()
mandelbrot.showInWindow(300,300)
mandelbrot.drawMandelbrot()
```