Recursive Structures and Processes

Curran Kelleher

August 24, 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." Functions can call other functions, and also themselves. 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 Java programming language. 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 "!".

$$3! = 3 * 2 * 1 \tag{1.1}$$

$$5! = 5 * 4 * 3 * 2 * 1 \tag{1.2}$$

$$X! = X * (X - 1) * (X - 2) * (X - 3) * ... * 3 * 2 * 1 \tag{1.3}$$

$$0! = 1 \tag{1.4}$$

So how can we compute this!? Here is a recursive function that does it: $n! = (n - 1)!$ for $n > 1$, and $n! = 1$ for $n \leq 1$. Here is that same function coded in Java, with a `main` method that calls it with the numbers 0 through 10, printing the output to the console:

```
static int factorial(int n) {
    if (n > 1)
        return n * factorial(n - 1);
    else
        return 1;
}
public static void main(String[] args) {
    for (int i = 0; i <= 10; i++)
        System.out.println("factorial(" + i + ") = " + factorial(i));
}
```

Consider the case that the `factorial` function were defined only as `return n * factorial(n-1)`, then the function would call itself infinitely and never return. This is infinite recursion, 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 $n \leq 1$ 1 - it returns 1.

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

```
factorial(0) = 1
factorial(1) = 1
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.

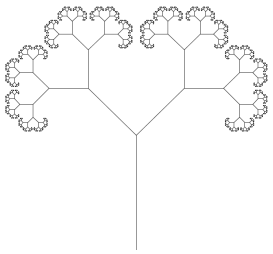
```
static int fibonacci(int n) {
    if (n > 1)
        return fibonacci(n - 1) + fibonacci(n - 2);
    else
        return 1;
}
public static void main(String[] args) {
    for (int i = 0; i <= 10; i++)
        System.out.println("fibonacci(" + i + ") = " + fibonacci(i));
}
```

The output of this program is the Fibonacci sequence:

```
fibonacci(0) = 1
fibonacci(1) = 1
fibonacci(2) = 2
fibonacci(3) = 3
fibonacci(4) = 5
fibonacci(5) = 8
fibonacci(6) = 13
fibonacci(7) = 21
fibonacci(8) = 34
fibonacci(9) = 55
fibonacci(10) = 89
```

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

1.2.3 A Tree Fractal



Now we'll begin to generate pictures using recursion. This how fractals are generated. The program defines a recursive process, and the resulting tree is a recursive structure. Curiously, such recursive structures and processes are found everywhere in nature.

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 `trunkHeight`, and the number of levels of recursion (the number of times the tree branches into new trees begore it bottoms out) is determined by `depth`.

```
static DrawingPanel panel = new DrawingPanel();

static double angleFactor = Math.PI / 4;
static double sizeFactor = 0.592;
static double trunkHeight = 0.4;
static int depth = 12;

public static void growTree(double x1, double y1, double rootLength,
        double rootAngle, int depth) {
    double x2 = x1 + Math.cos(rootAngle) * rootLength;
    double y2 = y1 + Math.sin(rootAngle) * rootLength;
    panel.add(new Line(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);
    }
}

public static void main(String[] args) {
    growTree(0.5, 0, trunkHeight, Math.PI / 2, depth);
    panel.showInFrame();
}
```

First a bit of code explaination. An open source Java visualization library called JyVis provides the drawing API (Application Programming Interface) that these examples use. This API provides the `DrawingPanel` and `Line` classes. `Line`s and other objects can be added to a `DrawingPanel` using the `add` method. The method `showInFrame` in `DrawingPanel` creates and displays a full-screen window on the screen. This library is used instead of directly using Java's graphics API so that the code focuses mainly on the algorithms, and is not cluttered with the peripheral details of graphics and user interface code. The important part of the code to note is the recursion: if the maximum depth has not been reached, then branch to the left and branch to the right, passing [the current depth]-1 as the new depth (so it will eventually bottom out).

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 and other fractal shapes. 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?

1.2.4 The Koch Curve

The Koch Curve (also called Koch Snowflake, or Koch Star) first appeared in a paper by Swedish mathematician Helge von Koch. The Koch Snowflake has finite area but infinite perimiter. 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.

To generate the Koch curve, start with a line. Divide that line into 3 segments, and change the middle segment into an equilateral triangle with no bottom. This is the rule. After one iteration of the rule we are left with 4 line segments. Now apply this rule to each of the four line segments, which results in 16 line segments. This is the second iteration of the rule. Apply the rule to those 16 line segments, this is the third iteration of the rule. The Koch Curve is what results after iterating the rule an infinite number of times.

The Koch Curve is a purely theoretical object, because it's definition never bottoms out. Here is a program which approximates the Koch Curve by iterating the rule 7 times:

```
static DrawingPanel panel = new DrawingPanel();
static int depth = 7;

public static void createCurve(double x1, double y1, double rootLength,
        double rootAngle, int depth) {
    double unitX = Math.cos(rootAngle), unitY = Math.sin(rootAngle);
    double dx = unitX * rootLength, dy = unitY * rootLength;
    if(depth>0){//(cx,cy)
        //           *
        //          /|\
        //         / h \
        //        /  |  \|---d---|
        //*-------*   *   *-------*
        //(x1,y1) (ax,ay) (bx,by) (x2,y2)
        double ax = x1 + dx * 1 / 3, ay = y1 + dy * 1 / 3;
        double bx = x1 + dx * 2 / 3, by = y1 + dy * 2 / 3;
        double d = rootLength / 3, h = Math.sqrt(3) / 2 * d;
        double cx = x1 + dx / 2 - unitY * h, cy = y1 + dy / 2 + unitX * 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
        panel.add(new Line(x1, y1, x1 + dx, y1 + dy));
}
public static void main(String[] args) {
    createCurve(.01, .5, .98, 0, depth);
    panel.showInFrame();
}
```

Notice the essential part: if the maximum depth has not been reached, then recurse four times - once for each new line segment. Only when the recursion bottoms out is a line drawn.

1.2.5 The Koch Curve as a Lindenmeyer System

The Koch Snowflake can also be described by a Lindenmeyer System (L-System). L-Systems create fractals by re-writing an initial string (by replacing certain characters with certain strings) many times, then interpreting the resulting string graphically. An L-System consists of an initial string, replacement rules (the grammar), and an interpretation of characters whereby each character changes the rotation or position of the "turtle" and/or draws something (or does nothing).

For example, here is an L-System which creates the Koch Curve:

- Initial string: "F"
- Grammar: "F" → "F+F–F+F"
- Interpretation:

- "F": move the turtle foreward one unit (with it's current rotation) and draw a line from the previous position to the current position.
- "+": increase the rotation of the turtle by $\pi/3$
- "-": decrease the rotation of the turtle by $\pi/3$

Here is Java code which implements this L-System, and interates it (printing the string) when you click on the screen:

```
static DrawingPanel panel = new DrawingPanel();

static int depth = 0;
static double turtleX, turtleY, turtleTheta = 0;
static double increment = .98 / Math.pow(3, depth);

static void createCurve() {
    String string = "F";
    for (int i = 0; i < depth; i++)
        string = string.replace("F", "F+F--F+F");

    System.out.println(string);

    turtleX = .01; turtleY = 0.5; turtleTheta = 0;
    increment = .98 / Math.pow(3, depth);

    for (char c : string.toCharArray())
        if (c == 'F') {
            double previousX = turtleX, previousY = turtleY;
            turtleX += Math.cos(turtleTheta) * increment;
            turtleY += Math.sin(turtleTheta) * increment;
            panel.add(new Line(previousX, previousY, turtleX, turtleY));
        } else if (c == '+')
            turtleTheta += Math.PI / 3;
        else if (c == '-')
            turtleTheta -= Math.PI / 3;
}
public static void main(String[] args) {
    createCurve();
    panel.showInFrame();
    panel.addMouseListener(new MouseAdapter() {
        public void mouseClicked(MouseEvent arg0) {
            depth += 1;
            panel.clearObjects();
            createCurve();
            panel.updateDisplay();
        }
    });
}
```

This is the program output after 2 clicks:

```
F
F+F--F+F
F+F--F+F+F+F--F+F--F+F--F+F+F+F--F+F
```

1.2.6 The Sierpinski Triangle

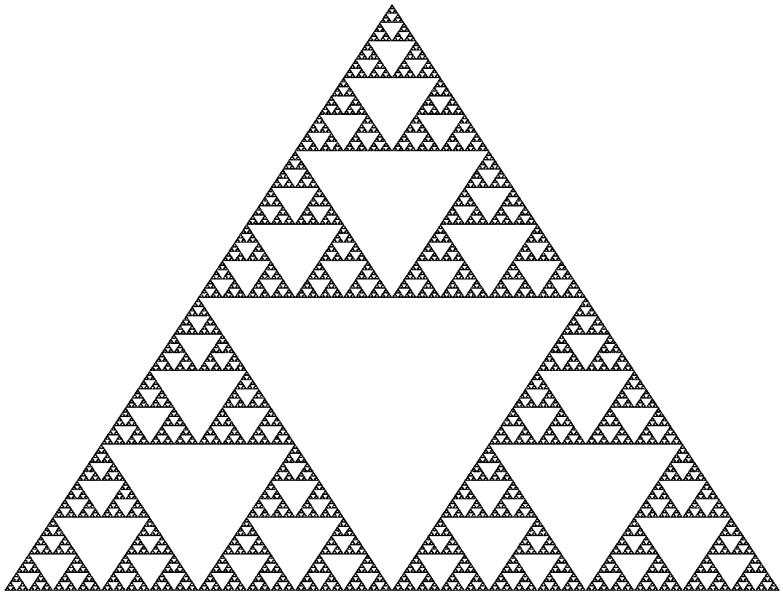
The Sierpinski Triangle (or Sierpinski Gasket) was described by Wacł+aw Sierpiński in 1915. To get the Sierpinski Triangle, divide a triangle into three smaller triangles using the midpoints between the vertices of the original triangle, and repeat this rule to the resulting triangles ad infinitum. Here is Java code which implements this method of construction:

```
static DrawingPanel panel = new DrawingPanel();
static int depth = 7;

public static void makeTriangles(double x1, double y1, double x2,
        double y2, double x3, double y3, int depth) { //    2
    double x1_2 = (x1 + x2) / 2, y1_2 = (y1 + y2) / 2;//   / \
    double x1_3 = (x1 + x3) / 2, y1_3 = (y1 + y3) / 2;//  /   \
    double x2_3 = (x2 + x3) / 2, y2_3 = (y2 + y3) / 2;// 1-----3

    if (depth > 0) {
        makeTriangles(x1, y1, x1_2, y1_2, x1_3, y1_3, depth - 1);
        makeTriangles(x1_2, y1_2, x2, y2, x2_3, y2_3, depth - 1);
        makeTriangles(x1_3, y1_3, x2_3, y2_3, x3, y3, depth - 1);
    } else {
        double[] xPoints = { x1, x2, x3 }, yPoints = { y1, y2, y3 };
        panel.add(new Polygon(xPoints, yPoints));
    }
}

public static void main(String[] args) {
    makeTriangles(.1, .1, .5, .9, .9, .1, depth);
    panel.showInFrame();
}
```



1.2.7 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 probabalistically 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 fractal images.

The Sierpinski Triangle

The Sierpinski Triangle emerges from an extremely simple IFS, often called the Chaos Game. In the Chaos Game, there are three points - vertices of a triangle. An initial point is chosen (it could be any point) and plotted on the screen. Then one of the three vertices is chosen at random, the point is moved halfway between it's current position and the chosen vertex, and it is plotted. This process is repeated indefinitely, and the Sierpinski Triangle emerges. Here is Java code for the Chaos Game:

```
static Random randomNumberGenerator = new Random();
static double[] xCoords = { .1, .5, .9 };
static double[] yCoords = { .1, .9, .1 };

public static void draw(PixelPlotter plot) {
    double x = xCoords[0], y = yCoords[0];
    while (true) {
        int next = randomNumberGenerator.nextInt(3);
        x = (x + xCoords[next]) / 2;
        y = (y + yCoords[next]) / 2;
        plot.plotPoint(x, y);
    }
}

public static void main(String[] args) {
    PixelPlotter plot = new PixelPlotter();
    plot.showInFrame();
    draw(plot);
}
```

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. The system is comprised of the following four functions, which map from any point inside the outermost black rectangle...

1. to a point on the green part of the stem (1% of the time)

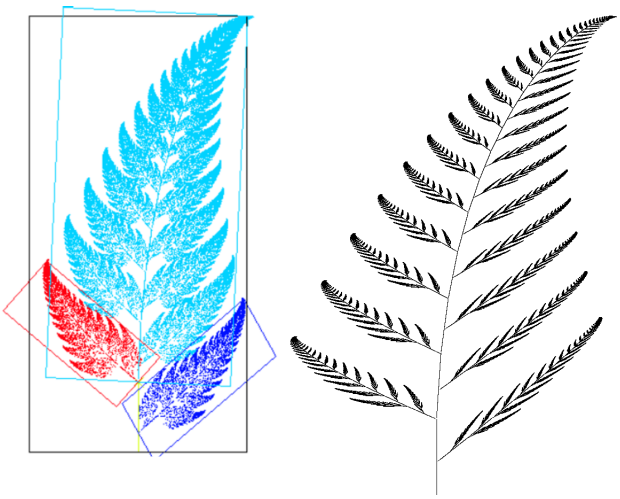


Figure 1.1: The rectangles used in the coordinate transformations (approximately), and the image generated by our program.

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)

Here is Java code which implements this IFS:

```
//                  maps up and spiraling
//              maps to right branch     |
//         maps to left branch     |     |
//    maps to the stem|     |     |     |
static double[] a = { 0,   .2, -.15,  .85 },
                b = { 0, -.26,  .28,  .04 },
                c = { 0,  .23,  .26, -.04 },
                d = {.16, .22,  .24,  .85 },
                f = { 0,  1.6,  .44,  1.6 },
    probabilities = {.01, .07,  .07,  .85 };

public static void draw(PixelPlotter plot) {
    plot.window.set(-5, 5, -1, 10);
    double x = 0, y = 0;
    while (true) {
        int i = chooseIndex();
        x = a[i] * x + b[i] * y;
        y = c[i] * x + d[i] * y + f[i];
        plot.plotPoint(x, y);
    }
}

private static int chooseIndex() {
    double randomNumber = Math.random();
    double sumOfProbabilities = 0;
    int i = 0;
    for (; randomNumber >= sumOfProbabilities; i++)
        sumOfProbabilities += probabilities[i];
    return i - 1;
}

public static void main(String[] args) {
    PixelPlotter plot = new PixelPlotter();
    plot.showInFrame();
    draw(plot);
}
```

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 probabalistically which of the four mappings to apply.

1.2.8 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 excape. If z never escapes the circle, then it is in the Mandelbrot

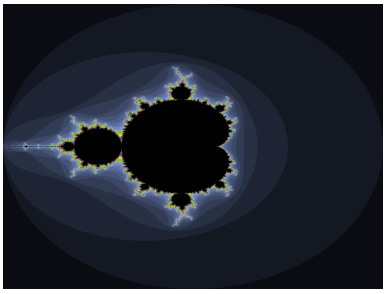


Figure 1.2: The image generated by our Mandelbrot program. Black points are in the Mandelbrot set. The other points are colored based on the number of iterations it took for z to escape the circle of radius 2 centered at the origin.

set, and is colored black. We assume that if z is still inside the circle after `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.

```
static PixelPlotter plot = new PixelPlotter();
static int maxIterations = 100;
static ComplexNumber z = new ComplexNumber();
static ComplexNumber c = new ComplexNumber();

public static void draw() {
    plot.window.set(-2.2, 2.2, -2, 2);

    for (int xPixel = 0; xPixel < plot.window.getWidth(); xPixel++)
        for (int yPixel = 0; yPixel < plot.window.getHeight(); yPixel++) {
            c.real = plot.window.getXValue(xPixel);
            c.imaginary = plot.window.getYValue(yPixel);
            int iterations = calculateIterations(c);
            Color c = calculateColor(iterations);
            plot.plotPixel(xPixel, yPixel, c);
        }
}

static int calculateIterations(ComplexNumber c) {
    z.real = z.imaginary = 0;
    int iterations = 0;
    while (circleContains(z) && iterations < maxIterations) {
        z = z.squared().plus(c);
        iterations++;
    }
    return iterations;
}

private static boolean circleContains(ComplexNumber z2) {
    return Math.sqrt(z.real * z.real + z.imaginary * z.imaginary) < 2;
}

private static Color calculateColor(int iterations) {
    if (iterations == maxIterations)
        return Color.black;
    else {
        int red = (iterations * 10) % 255;
        int green = (iterations * 12) % 255;
        int blue = (iterations * 17) % 255;
        return new Color(red, green, blue);
    }
}

public static void main(String[] args) {
    plot.showInFrame();
    draw();
}
```

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.