CS112 Lecture: Recursion

Last revised 3/20/08

Objectives:

1. To introduce the notion of recursion

Materials:

1. Von Koch curve images
2. Blue J projects: modified version of NestedRects/NestedRectsController, Expression representation, Word Frequency program, Factorial program, Fibonacci program
3. Handout of code for WordFrequency classes
4. Projectable of a recursive solution to project 1 (misuse of recursion)

I. Introduction

A. Today we will look briefly at a concept that will be dealt with in much more depth later in the curriculum (in CS212): Recursion.

B. We say that something is recursive if it is defined partially in terms of itself.

1. For example, many natural phenomena can be described in terms of fractals. A fractal is a recursive structure.

   PROJECT: Series of seven resolutions of Von Koch curve. Note how, at each step, a line is replaced by a copy of the curve at 1/3 the size. (Open in Preview; select Show Drawer in View; expand disclosure triangle)

2. For example, the factorial function in mathematics is defined recursively:

   0! is defined to be 1
   n! - for n > 0 - is defined to be n * (n-1)!

   Example: 3! is 3 * 2! = 3 * 2 * 1! = 3 * 2 * 1 * 0! = 3 * 2 * 1 * 1 = 6

3. The book gave examples of recursive pictures such as nested rectangles, brocolli, and parsley.

   RUN Modified NestedRectsController main() method [which draws a single picture and pauses at each step in the drawing process]

C. Any recursive definition must include one (or more) base cases plus one (or more) recursive cases. (Notice in our definition: "partially in terms of itself").

1. For recursive pictures (such as the Von Koch curve or the examples in the book), the base case is often when the level of detail in the picture reaches the limits of the resolution of the display device.

2. For factorial, the base case is 0!, which is defined simply as 1.

II. Defining Recursive Structures

A. A recursive structure is a structure which is self-similar - i.e. it consists of parts which are just like the whole.

   1. Show Koch curve again, showing how this works out

   2. The textbook picture examples exhibit this, too.

B. This can be useful in cases other than pictures, too. Consider, for example, arithmetic expressions in a language such as Java.

   For simplicity, we will confine ourselves to fully-parenthesized expressions composed of constants, the variable X, and the arithmetic operators +, - (unary and binary), *, and /.

   1. Example: (3 * (- X))

      a) Base case: a constant, by itself, is an arithmetic expression.

         Example: 3 is an arithmetic expression.

      b) Another base case: a variable, by itself, is an arithmetic expression.

         Example: X is an arithmetic expression.

      c) Recursive case: the negation of an arithmetic expression, enclosed in parentheses, is an arithmetic expression.

         Example: (- X) is an arithmetic expression because X is an arithmetic expression.

      d) Another recursive case: two arithmetic expressions combined by an arithmetic operator, enclosed in parentheses, is an arithmetic expression

         Example: (3 * (- X)) is an arithmetic expression consisting of the two arithmetic expressions 3 and (- X) combined by *. (We have already shown that 3 and (- X) are arithmetic expressions).

e) Of course, this can be carried out to any level of complexity. For example:

• ((3 + (2 * X)) / (X + 1)) is an arithmetic expression consisting of (3 + (2 * X)) and (X + 1) combined by /

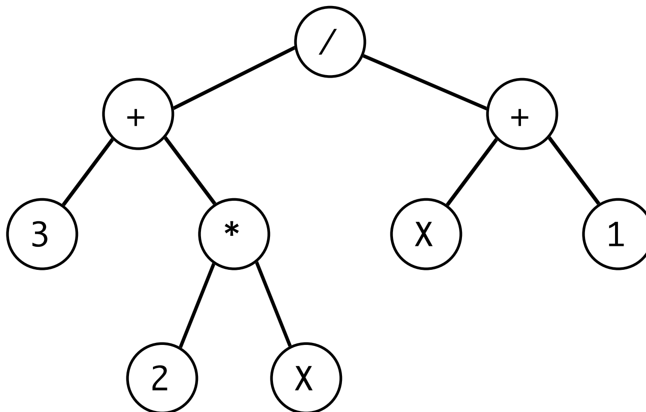
• (3 + (2 * X)) is an arithmetic expression consisting of 3 and (2 * X) combined by +

• (2 * X) is an arithmetic expression consisting of 2 and X combined by *

• (X + 1) is an arithmetic expression consisting of X and 1 combined by +

• Of course, 3, 2, 1, and X are all base case arithmetic expressions

2. An arithmetic expression defined this way can be picture by a structure known as an <u>expression tree</u>. (You will see this more in CS212). For example, ((3 + (2 * X)) / (X + 1)) can be represented as follows:



3. One of the beauties of recursive structures is that they can often be represented quite simply in a programming language. For example, .let's consider how we might represent arithmetic expressions - using an approach similar to that developed for the recursive structures examples in the book. (This is actually not the only way to do it, but is a good approach).

4. What we need to define

a) An interface (called, say, Expression) which defines the capabilities of any kind of expression

SHOW Code for Expression

b) A class that implements this interface for each kind of expression. In this case, we need four classes:

(1) Constant - a non-recursive case - SHOW Code

(2) VariableX - a non-recursive case - SHOW Code

(3) Negation - a recursive case - similar to the next one

(4) BinaryOperation - a recursive case with four subclasses (Sum, Difference, Product, Quotient) - SHOW Code

C. Another example: a list of word frequencies.

1. Consider the following problem - given a text, report the number of occurrences of each word in the text, or of particular words.

   Example:

   Show file demo.txt

   Demo using `java AnalyzeWordFrequency < demo.txt`

   Demo again using `java AnalyzeWordFrequency was < demo.txt`

2. There are actually quite a few ways to solve this problem, but we will consider one approach here - using a recursive list.

   Note: The example we are developing here is not actually the most efficient or easiest way to solve this problem!. It is meant as an example to illustrate recursion, and happens to be somewhat similar to a problem we will be using in lab.

   a) The basic idea - maintain a list of words and frequencies. Record each word in the text whenever it appears. Maintain the list in alphabetical order.

   e.g. for demo.txt, we would record

   the
   quick
   brown
   ...

b) We will define the list using an interface

   HANDOUT code

   Discuss methods of interface WordFrequencyList. (Note: handout includes interface and classes implementing it, but not main program, which uses file input and String processing methods and other things we have not discussed yet)

   Discuss methods defined by the interface, and how each is used in solving the problem

c) We need to define two classes implementing the interface - one representing an empty list, and one for a non-empty list

   (1) Discuss code for each method of EmptyWordFrequencyList

   (2) Discuss code for NonEmptyWordFrequencyList

      (a) Variables

      (b) Each method

      (c) Note, in particular, how recording the occurrence of a words is handled. Three possibilities:

         i) Word occurs here

         ii) Word belongs later in the list

         iii) Word belongs before the current word

            Note: the approach taken here is to always create a new list, rather than modifying the existing list. The latter would be more efficient in this case, but this example has been constructed in a way that will be helpful to you as you work on Lab 10.

III. Defining Recursive Methods

A. The examples we have considered thus far have involved recursive structures - a structure in which some elements contain one or more similar (but smaller) structures of the same type.

B. In the case of operations defined recursively (like factorial), it is sometimes useful to use a recursive method. A recursive method is one that can call itself.

Recursive methods are useful when implementing a recursive definition

1. Example: a method to calculate the factorial could be defined as follows:

```
/** Calculate the factorial of a non-negative number
 *  @param n the number (must be non-negative)
 *  @return its factorial
 */
public static int factorial(int n) {
     if (n == 0)
          return 1;
     else
          return n * factorial(n-1);
}
```

Demo (static method of class Factorial)

What will happen if we give it a negative argument? (Which is illegal, according to the prologue comment)

ASK

Demo with -1

2. Another example: the fibonacci numbers

Definition: the first two fibonacci numbers are 1. Each subsequent number is the sum of the two preceeding fibonacci numbers

a) Example: first few fibonacci numbers are 1, 1, 2, 3, 5, 8, 13 ...

b) These numbers show up in many interesting places

ASK class for examples

c) Develop code for a recursive method in class

d) Demo: (static method of class Fibonacci)

IV. Cautions

A. Recursion is a very powerful tool - but, like all powerful tools, needs to be used with caution. (Compare to a chain saw!)

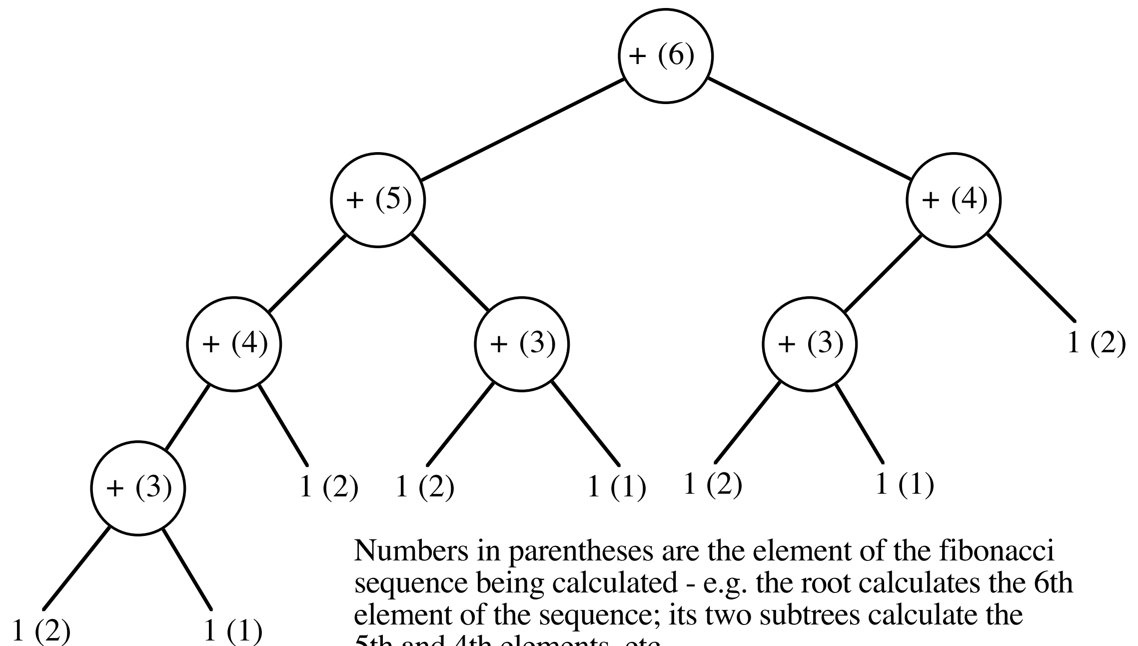
B. Some problems that can arise from mis-using recursion

1. Sometimes the recursive solution to a problem, though simple, is actually quite inefficient when compared to a more complex non-recursive solution

   Example: In the case of the fibonacci numbers, the recursive solution, though very simple, is actually quite inefficient for large values of n. In particular:

   a) The number of additions performed by the recursive solution turns out to be one less than the fibonacci number itself.

      Example: consider the calculation of fibonacci(6). The recursive computation can be represented by the following expression tree:



Numbers in parentheses are the element of the fibonacci sequence being calculated - e.g. the root calculates the 6th element of the sequence; its two subtrees calculate the 5th and 4th elements, etc.

      The inefficiency arises because many computations are repeated - e.g. the calculation of fibonacci(3) is done 3 times!

   b) For this particular problem, there is a much more efficient - but less easy to understand! - solution that takes time proportional to n:

      Show and demo non recursive version in class Fibonacci.

c) The advantage can be very significant - the fibonacci numbers grow exponentially, so - for example - the 46th element of the series is 1836311903, and is the largest member of the series that can be represented by a java int.

Demo calculation of 46th recursively and non-recursively

2. It turns out that recursion is equivalent to looping, in the sense that any code that can be written using a loop can be rewritten recursively. (The reverse is also true, but sometimes non-trivial!) Sometimes, one can solve a problem recursively when a loop would actually be simpler and clearer.

   Example: Project 1 - the robot steeplechase - is naturally solved using a loop. A couple of people in the class hit upon recursive solutions which, while correct, were very hard to understand. (In projecting the following example, I'm not intending to pick on anyone!)

```
public void runRace() {
      while (frontIsClear()){
          if (nextToABeeper()){
              pickBeeper();
              turnOff();
          }
          move();
      }
      climbUp();
      moveAcross();
      climbDown();
}

// climbUp and moveAcross() omitted

public void climbDown() {
      while(frontIsClear()){
          move();
      }
      turnLeft();
      runRace();
  }
```

   PROJECT

C. In general, recursion is used appropriately when

1. It is used with a structure that is itself recursive

2. It is used with a concept that is itself defined recursively (e.g. factorial or fibonacci).

3. In both cases, though, one should consider whether there might be a non-recursive solution that is either simpler or - though more complex - much more efficient.

   Examples

   a) In the case of the word frequency problem, there happens to be a non-recursive solution that is both simpler (and also happens, in this case, to be slightly more efficient).

      PROJECT Non-recursive solution - note that everything is in this one class (which is now slightly longer)

   b) In the case of the fibonacci numbers, there is a non-recursive solution that, though much more complex, is also much more efficient, as we have seen.

V. We will save further discussion of Recursion for CS212