Computer Science 115
Assignment 5
Due Friday April 1 at *4:30 pm*.

Objectives

- Extend the BinaryTree class with an ExpressionTree.
- Read through some solution code as exam prep and study material.
- Experience working on a Java project, by grouping some smaller projects.
- Play with Java Swing and AWT components (no coding required here).

Introduction

In this assignment, you will implement the engine of a calculator using a binary expression tree. Most of the work has already been done in the labs and in your previous assignments, so all you have to do is complete one algorithm.

Note that this is a shortened version of a previous CSC115 assignment. Because the final exam follows the end of classes with little time to study, please use this assignment as a study tool. Most of the code has been written, but it is reasonable to expect that you would be able to do this on your own.

Part 1 Preparation

1. Read the whole of this document.
1. Download the javaFiles.zip and unzip it in a working directory.
2. All the of the `*.java` files are complete and ready for running EXCEPT:
    - `ExpressionTree.java`
3. Run `javadoc *.java` on all the files in your working directory. Then double click on the index.html file. From the information, you should get a fair idea of how all the files fit together.
4. Compile the files. Of course, they don't work, but we've built the skeleton for you, so you know the syntax is sound.
5. Run the program by typing `java Calculator`
    - You will see the GUI, *Graphical User Interface*, come up on your screen. You can resize the window for comfort and visibility.
    - Enter an expression and try to create the tree. Of course you don't expect it to work yet. It is simply the *front end* that interfaces with the real code in the background.

Part 2 Understand the BinaryTree.

1. This is the `BinaryTree` you have been working on in Lab 9. You will have coded the various methods and traversals yourself as part of the lab. A fully functional binary tree will be posted in the toolbox on Friday. Take some time to compare your code to the one provided.
2. Test.

Part 3 Complete the Expression Tree.

The `ExpressionTree` is populated with tokens: either an operator or a number. The internal nodes contain operators and the leaves all contain numbers. Every subtree contains a complete expression that has a numeric value.

1. Scan the internal workings of this code. It has some complicated parts that you can ignore if you want. However, you should note the following:
   - The `ExpressionTree extends BinaryTree`
     - This means that all `public` AND `protected` attributes and methods are *inherited* by `ExpressionTree`.
   - The generic placeholder `E` has been replaced by `String`.
   - It has a single attribute, called `inputExp`, a `String array`.
     - This is initialized by the method called `parseInput`. It separates the input string into tokens. Effectively, it finds the operators {+,-,*,/,^} and puts them into their own index location in the array. It also separates parentheses. *if you decide to incorporate parentheses into your code, you can.* It will also separate the stuff between the operators by the fact that there are spaces between them.
     - Do not change `parseInput` unless you enjoy a lot of stress. Note that you are not expected to become familiar with how this method works.
   - The `evaluate` method follows the algorithm described in Assignment 4 except that the '*' operator is used for multiplication, and double values are allowed. The postOrderIterator is used to serve up the postfix expression from the tree. Have a look at the method and see how it differs from the one you implemented.
2. Complete the `populate` method. It is called by the constructor.
   - See the algorithm below for details.
   - You can expect any one of the following set of operators {+, -, *, /}. If you would like to add round parentheses or the exponent operator '^', you can add that if you like, but it is not required.
   - You are required to impose the regular arithmetic operator precedence.
     - $4 + 2 * 3 = 10 \neq 18$
3. Test.
   - You should be doing this as you progress. Since we are not overly concerned about dealing with incorrect expressions, test only valid strings.
   - You can run `Calculator` and it will provide feedback on your input and print out any Exception errors.

The Populate Algorithm:

There is no guarantee that the non-operators are actual numbers or that the expression is completely valid. However, unless there is a real obvious error (ie. an operator missing an operand), just make the tree. If the user screwed it up, then that will become apparent when it is evaluated.

However, if the user submits a string that messes up the parsing, then throw an `ExpressionTreeException` at this point, with a nice message, of course.

Algorithm for parsing an infix expression into a tree: (Note that we are not handling parenthesis)

- **Recursively** parse an array of tokens *in reverse order:* index [end ... start]
  - in a loop that looks for operators in this reverse order of precedence: {+,-,*,/,^}
    - at the index of found op:
      - create a node with the op as root and 2 children:
        - each child (subtree) is created from the recursive call to the method on the part of the array from start to index-1 and from index+1 to end
        - return this node as a root of a subtree at the current level of recursion.
  - The base case happens when you cannot find any operators in the sub-array. In that case, start = end and there is one non-operator token.
    - So, when no operator is left, and the sub-array is not empty, then create a node of the remaining token and return that.
- Make the parent class root point to the highest level TreeNode that was called (this part is already implemented in the constructor of the ExpressionTree).

Part 4 (Play time)

Now you can play with the GUI (by running Calculator.java). Start out softly, with correct expressions. Once you have tested them, try a few error-checking sequences. The messages from your ExpresionTreeException objects should show up in the "Errors" part of the GUI.

Submission

Submit your completed

- `ExpressionTree.java`.

If you have *created* or *modified* any `*.java` files that are required to run your program, you must submit these as well. Otherwise, the tester will be using the exact files given with this assignment.