CSC 301 Sections 402, 411, 701, 710 Fall 2017
Homework Assignment 2
Due as specified on D2L

Note 9/22: I am providing you with partially complete code for this assignment. I created a jar file containing this code, but D2L will not currently let me upload it because it is a jar file with (what it thinks) suspicious contents. So I posted the jar file here: http://condor.depaul.edu/slytinen/301f17/hw2.jar.

Directions: Please complete the problems below. The assignment will be graded on a scale of 0 to 3. You must complete 3 methods, each of which is worth 1 point. I will provide template code shortly, as well as a main method which invokes each of your methods on a variety of test cases. You will receive full credit for a problem if your method works properly for all or most of the test method calls. You may receive ½ point for a problem if your code compiles and works in some cases. No partial credit will be given for code that does not compile.

As you work on this assignment, you may feel free to discuss it at a conceptual level with other students in the course. However, when you write your code, **you must work on your own.** As you write your code, you may ask for help from me and/or a CDM tutor, but **not** from other students in the course, friends, etc. Needless to say, you **may not** copy a solution from the Web. Copying code from another student or from the Web is plagiarism.

To complete this assignment, you will complete the **ExpressionTree** class. A template for this class, and code that tests it, will be found in the form of a jar file which you can download from http://condor.depaul.edu/slytinen/301f17/hw2.jar. An **ExpressionTree** object is meant to represent an arithmetic expression. For our purposes, an arithmetic expression consists of one of the following:

- a number; or
- an arithmetic expression, followed by an operator (we'll only use +, –, and *), and a second arithmetic expression.

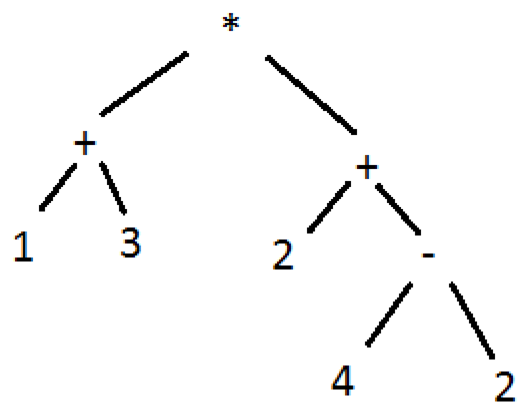
Note that the definition is recursive; that is, an arithmetic expression can be part of a larger expression. You may assume that each item in the expression is separated by other items with a whitespace. Each arithmetic expression, with the exception of a number, must be surrounded with parentheses. Here are some examples of valid expressions:

```
( 1 + 2 )
10
( ( 2 * 4 ) - 3 )
( ( 1 + 2 ) + ( ( 6 - 4 ) * 5 ) )
```

These are invalid expressions:

```
1 + 2
( 1 + 2 + 3 )
```

The expression should be represented as a **binary tree**. The operator of the expression is the root, and the left and right children are the expression to the left of the operator (the left child) and the expression to the right of the operator (the right child). For example, the tree which should represent ((1 + 3) * (2 + (4 – 2))) is the following;



The **ExpressionTree** class has 4 public methods: 2 public **constructors** (overloaded), a **toString** method, and an **evaluate** method. There are also 2 helper (private) methods: a third constructor, and a method called **getExpr**. You must write **(1) the toString method; (2) the evaluate method; and (3) the getExpr method.** Here is a description of each of the 6 methods in the **ExpressionTree**.

I. **Constructor #1**: I am providing you the complete code for this method. This constructor is passed 3 values: a **String** and two **ExpressionTree** objects. The String is stored in the **item** instance variable, and the 2 ExpressionTree objects are stored in the **left** and **right** instance variables, respectively. You may not need to call this constructor in the code that you write, but I will use it in my test code.

II. **Constructor #2:** I am providing you the complete code for this method. This constructor is passed 1 value, which is a **String**. The String represents an arithmetic expression. This constructor breaks apart a String into "tokens" (substrings) as delimited by whitespaces, using the **split** method of the **String** class. The tokens are then placed in a Queue, in which the first token is at the front of the queue. In Java, the class that we are encouraged to use for Queues is called **ArrayDeque**. In the **ArrayDeque** class, the enqueue operation is called **offer**, and the dequeue operation is called **poll**.

III. **Constructor #3:** I am providing you the complete code for this method. This version of the constructor is passed an **ArrayDeque**, and forms an expression tree by repeatedly polling tokens from the ArrayDeque until an arithmetic expression can be formed from those tokens.

IV. The **toString** method: **You must write this method.** It is passed no parameters, and returns a String, which represents the expression tree rooted by the given **ExpressionTree** object**.** For example, referring back to the diagram above, if **t** is the name of the root, then **t.toString()** should produce "((1 + 3) * (2 + (4 – 2)))". In general, if **s** is a String, then if we call

new ExpressionTree(s).toString();

the result is the original string **s**.

V. The **getExpr** method. **You must write this method.** It is passed the queue created by the constructor. When completed, it must create a tree representation of the next sequence of tokens on the queue which form an arithmetic expression. It should do this by dequeuing Strings from the queue (the Java method is called **poll**) until it has reached the end of the next arithmetic expression on the queue. For example: if the Queue contains (from front to back)

```
["(", "1", "+", "2", ")"]
```

Then the Queue should be polled 5 times, since the 5th poll will return the ")" indicating the end of the expression. In general, there may be additional items on the Queue after the first regular expression is found (after the `")"`) **.**

VI. An **evaluate** method, which returns an int (the value of the expression). For example, **evaluate** from the above tree should return 16.