/\*\*

\* This program will implement a GUI application for reading in a postfix expression, and then \* translate it into infix form, and output its Three Address code into a file.

\*

\* Name: Nabeel Hussain

\* Class: CMSC350

\* Professor: Didier Vergamini

\* Project 2

\* Date: 11/18/2016

\*

\* **@author** Nabeel Hussain

\*/

This program for project 2 involves implementing a Three Address Generator, in which the user enters a postfix expression to be translated into infix form. The result is shown using a GUI, and the expressions Three Address code is outputted into a file. The program translates the postfix expression using an operand stack to hold the nodes of the expression Tree being built.

My program has six classes.

The first class is Stack.java, which implements the stack data structure. It will be used to create my expression tree to hold the operator and operand nodes of the postfix expression being translated.

The second class is Node.java, which will be an abstract class that defines a node of the expression tree.

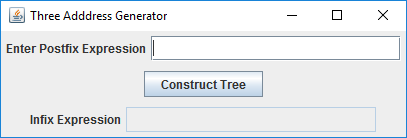
The third class is OperandNode.java, which extends the Node class. It defines the components of an operand node, which will be used to hold the integers in the expression tree.

The fourth class is OperatorNode.java, which extends the Node class. It defines the components of an operator node, which will be used to hold the + - \* and / operators in the expression tree.

The fifth class is ExpressionTree.java, which builds the expression tree containing the components of the postfix expression read in. It holds the methods that will be used by my GUI class to perform the Three Address Generator. This will include a method to separate the individual tokens of the postfix expression read in, so the expression tree can be properly built. It will also contain two Boolean methods to distinguish whether a token in the expression is an operator or operand. It will also contain a method to create the expression tree and return the infix form of the postfix expression. Lastly, it will also include a method for generating the three address code of the expression and writing it into a file.

The sixth class is used for the GUI, which will contain the buttons and a text field output to meet the functional requirements for this assignment.

My GUI:



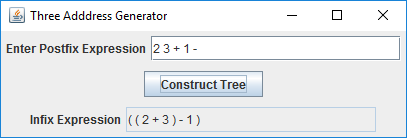
**Test Plan:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Input** | **Expected Output**  (Displays a fully parenthesized infix expression) | **Did Test Pass?** |
| 1 | **Postfix Expression = 2 3 + 1 -**  \*Spaces between each token\* | ( ( 2 + 3 ) - 1 ) | Y |
| 2 | **Postfix Expression = 7 2 5 +3 2 \*+ 1- /**  \*Leading blank space\*  \*Random or no spacing between operands and operators\*  \*All operators are used\* | ( 7 / ( ( ( 2 + 5 ) + ( 3 \* 2 ) ) - 1 ) ) | Y |
| 3 | **Postfix Expression = 72 5 +32\*1-**  \*Blank space at the end of the expression\*  \*Integers should always be separated by a space, in order to distinguish between multi digit numbers. \* | ( ( ( 72 + 5 ) \* 32 ) - 1 ) | Y |
| 4 | **Postfix Expression = 2 3 &**  \*Contains an invalid token in the expression\* | JOption Pane Show Message Dialog:  "Invalid token &" | Y |
| 5 | **Postfix Expression = 7 2 5 +3 2 \*+ 1- \***  \*Tests whether the three address generator properly  displays the correct code in the output file \* | ( 7 \* ( ( ( 2 + 5 ) + ( 3 \* 2 ) ) - 1 ) )  **ThreeAddressCode.txt**  Add R0 2 5  Mul R1 3 2  Add R2 R0 R1  Sub R3 R2 1  Mul R4 7 R3 | Y |

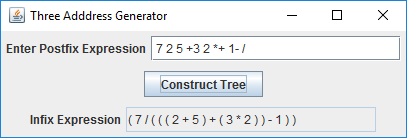
\*Additional test cases are in my **ExpressionTreeTest.java** Junit test files

**Screen shots of successful compilation and running for all test cases**

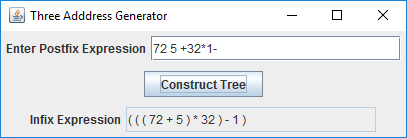
**Test Case 1:**



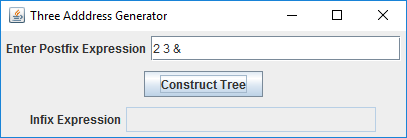
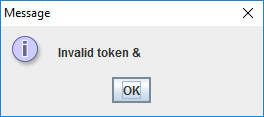
**Test Case 2:**



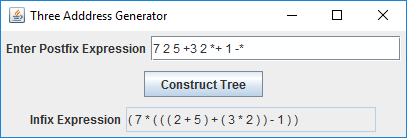
**Test Case 3:**

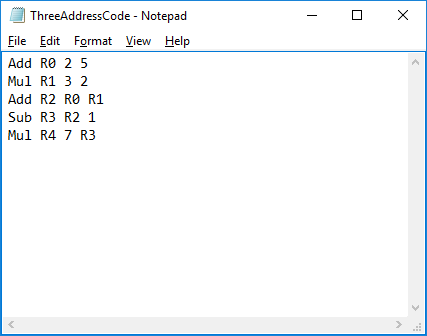


**Test Case 4:**

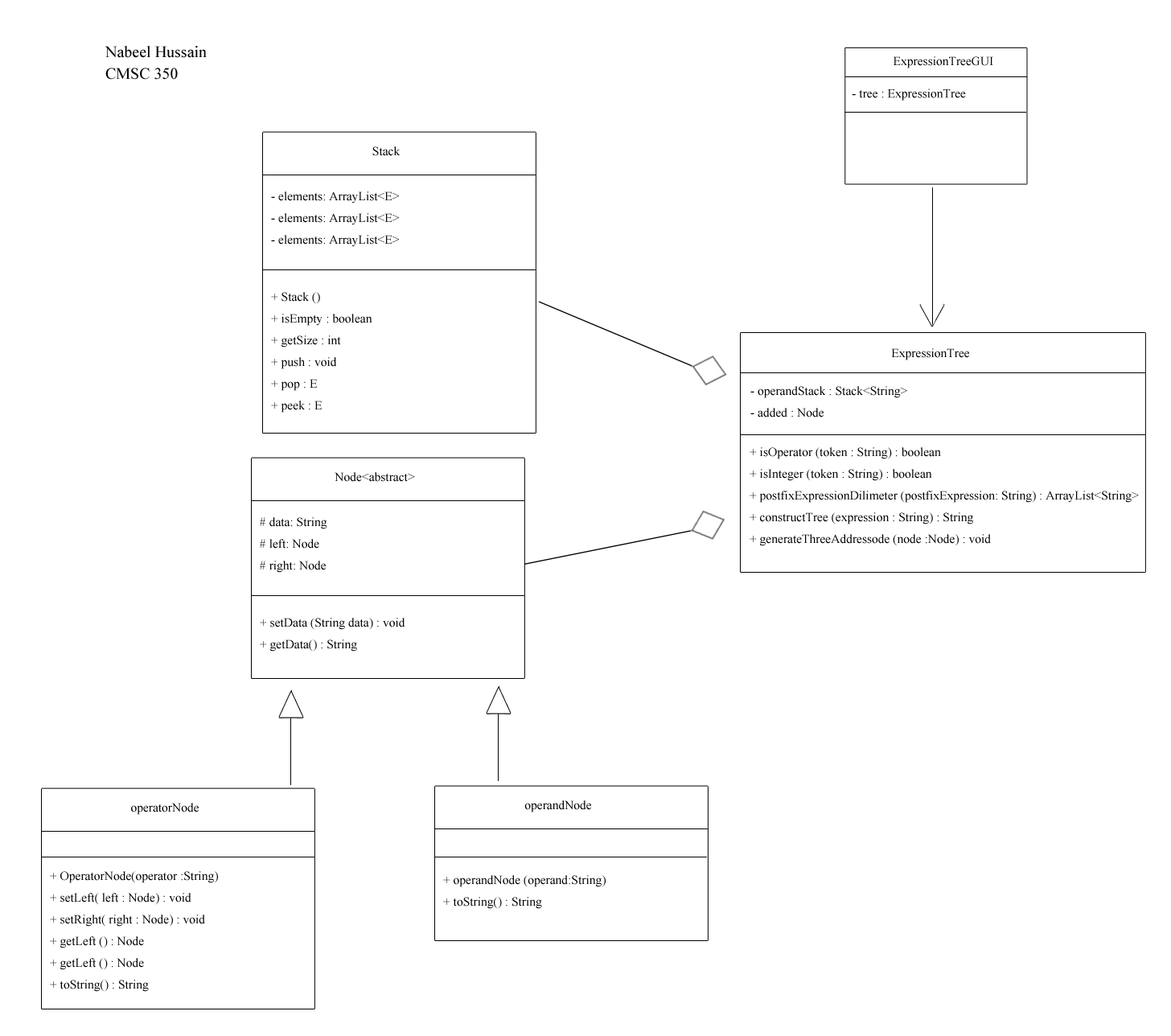


**Test Case 5:**





**UML Diagram:**



**Lessons Learned:**

While working on Project 2, there were many new concepts and techniques that I learned about building binary expression trees, as well as how to go about traversing them. I learned that one way to build a tree, is by using Stacks to hold the nodes of the tree, and then pushing and popping them while iterating through it, to create the desired tree.

I also learned about three address code and how to generate it for a binary expression tree, by doing a post order walk. This was by far the most challenging part of the assignment for me, and it took me quite a while to fully understand how to go about implementing the code. I had to re- read and go over the three address code example from our reading multiple times, before I was able to properly understand what to do. However, after looking at how to perform postorder travel examples online, I was able to figure it out after much trial and error.

Overall, this assignment was the most challenging and time consuming for me so far, but it definitely helped me understand and reinforce all the material I learned from this week’s module much better.