Eric Olsen

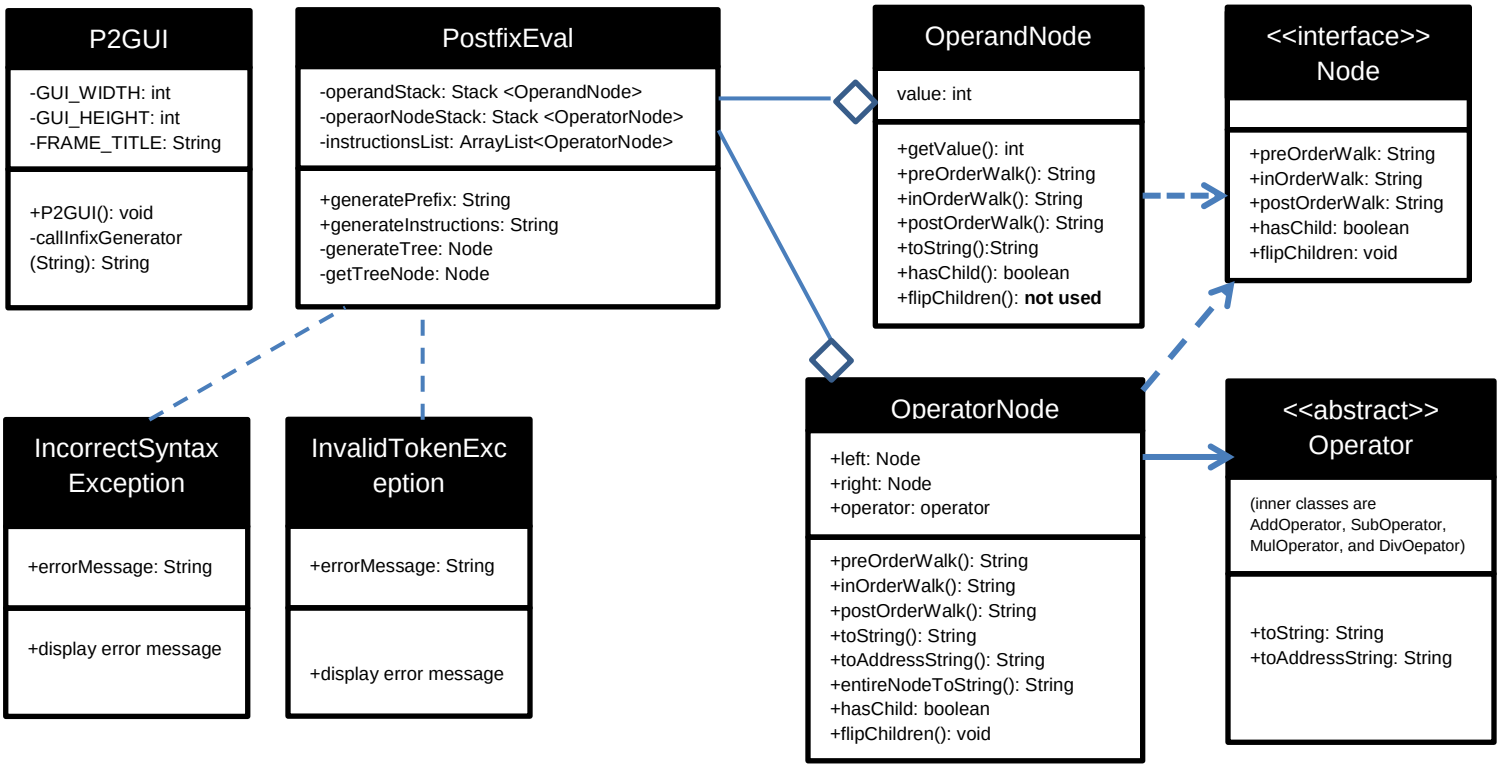
Assumptions:

The task for this assignment is to build a Graphical User Interface (GUI) that accepts a postfix String input, creates a data tree structure and then creates two outputs. The first output will display the input postfix expression in a parenthesized prefix state. The second output will be a saved file of the same expression in three address format. This file will be saved locally to the same folder holding the java source code. We need to make two assumptions in order to keep the scope of this project to manageable levels. Our first assumption is that this program will only utilize the four basic arithmetic operators. The second assumption is that the user is responsible for inputting the postfix expression in the proper format. There will be some degree of error detection when we calculate a tree with missing nodes; however, this will not be a caught exception. It will merely be a cautionary message to the user.

Main Design Decisions:

This program has a complicated class structure, but uses recursion for the bulk of its calculations. There are seven total classes. The first class is 2GUI, which, for the most part, is self-explanatory. The PostfixEval Class contains two public methods. The method generatePrefix(String) receives the GUI's postfix String input, creates a data tree, and returns a String value of the postfix's prefix representation. The generateInstructions method takes an ArrayList of OperatorNodes, generated during the postfix parsing, and returns a three address String representation of those nodes. This will be described in more detail later.

The tree data structure is composed of OperatorNode and OperandNode classes. Every OperatorNode has an assigned pointer to its left and right child node. The Operator class is also extended into inner classes for each of the four basic arithmetic operations. The OperatorNode class implements the Node interface, so the same series of methods can be directed to both operators and operands. This is our basis for using this program's us of recursion. OperandNodes differ in that they are always leaf nodes. OperandNodes hold integer values and implement all but one of the Node interface methods. The UML diagram depicts the overall class hierarchy for this program. I now realize that including methods was not required. Pardon my abuse of this paper's white space.



Now I will walk through PostfixEval's two public methods in greater detail. The first invoked method is the generatePrefix method. This method receives and tokenizes the postfix String in the same manner that it did in project 1. More specifically, it creates a substring of the character at String index zero, instantiates a character, shortens the tokenized string by one index, and logically tests the character. It also keep track of the last character so multiple digit integers, separated by spaces, can be processed. The very beginning of this method also clears all the field data structures used in the previous calculations. Without doing this the operatorNode stack continually grows causing some very confusing results back in the GUI.

This method differs from project 1 in that one stack contains OperandNode objects and the other OperatorNode objects rather than primitive data types. There is also an instructionsList ArrayList that adds operator nodes for the generation of 3 address instructions back in the GUI. When tokenizing input, this method always pushes every integer into the Operand stack no matter what. When the evaluated character matches one of our four arithmetic operators, one of three selection statements execute.

First, if the operandStack contains two or more operands, we create an OperatorNode with the top two operands as leaf children. We pop the top two operands, instantiate an OperatorNode, and push this node into the operatorNodeStack. Second, if there is only one operand in the operandStack, we pop the operandStack, operatorStack, and instantiate a new OperatorNode with the operand being the left child. We can assume that the operand will always be the left child only because we're working with postfix notation. Third, if we encounter an arithmetic operator and there are no operands in the stack, we conclude that we are not dealing with leaf nodes. We therefore pop the operator stack, copy the value of the next Operator node with the peek method, push the popped value back into the operator stack, and instantiate our new OperatorNode with three operator elements. This new operator node is then pushed into the top of the operator node stack. The end result of this tokenized input is a stack composed of Operator Nodes and an empty OperandStack.

This method's work is half done at this point; we still need to generate the tree structure and return a prefix representation. At this point we call a private method called generateTree(). This method takes the OperatorNode stack and pops each value. The overall class of the tree is a Node, but we can also safely assume that the root node is an operator. We consequently instantiate the tree's root node as a new operator object. We then continually pop the operatorNodeStack until we have instantiated operand leaf nodes. Our primary way of recognizing leaf nodes is the hasChild boolean method. When this method returns false, we know we're dealing with an operand.

Knowing exactly how many nodes to instantiate at run time is a little difficult to design. Ultimately we achieve this by utilizing a private method called getTreeNode(). This method is a recursive call which relies on the hasChild() method. When this method is false it returns the a leaf node (operandNode). When it is true it recursively calls the generateTree method. Getting this to work was painful, but also incredibly simple once complete. As an additional note, it was necessary to create a method in the Node interface called flipChildren due to the nature of popping stacks. Since the right child is always popped before the left, it was necessary to flip the tree's children whenever the stack was greater than one operatorNode.

Generating the three address instructions took some trial and error. In the end it was easiest to implement by adding operatorNodes to an ArrayList during the initial processing of the postfix string. Getting more specific, this method operates by iterating through the entire instructionsList and concatenating a single String value. It uses one outer for loop and two inner for loops. The outer for loop steps forward through the entire list from the first operatorNode entry to the last. Each iteration it creates a new register value. The for loop then utilizes the hasChild method which checks to see if each child node is or is not an operand. Since operands have no children, they return false. encounters an operator, it starts an inner for loop which iterates backwards toward the root to detect the first previous match of the detected operator. This approach was very much iterative rather than recursive.

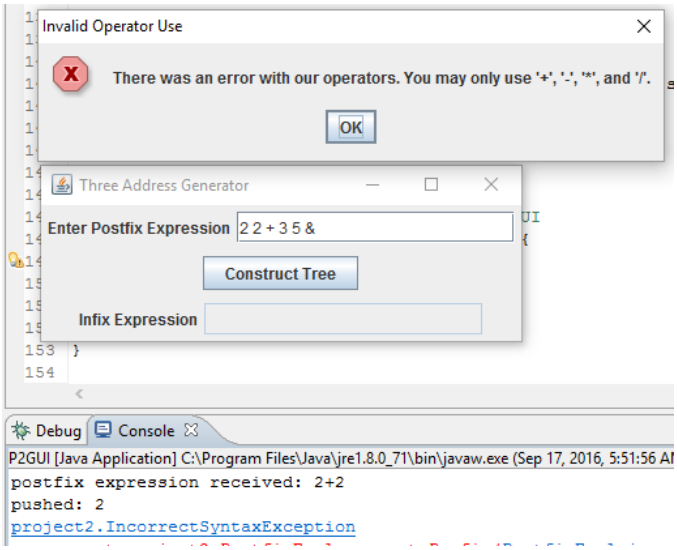
Error Handling:
As stated in the assumptions, not all possible errors are handled in this program. For example, when a user provides too few operators to complete the postfix String input, the program does not correct the user. However, when generating the tree, the PostfixEval class does do a final check on the OperatorNode stack to see if it's empty. If it is empty, we conclude that it processed all nodes in the tree. If it is not, we conclude that too few operators were provided and a warning

message appears. See test case 10 for an example. As of submission, providing too many operators will generate a run-time exception as the PostfixEval attempts to pop an empty operator stack, see test case 11. With more time, I would correct this, but for now I will just write it off as a program assumption. The IncorrectSyntaxException and InvalidTokenException are demonstrated below in test cases 9 and 10.

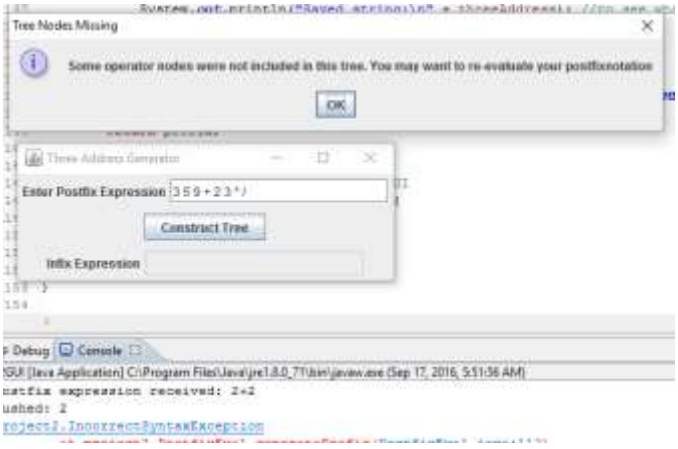
Program Testing:

Test Case #	Input	Description	Expected Output	Actual Output	Pass / Fail
1	3 5 9 + - 2 3 * /	Best case scenario, assignment example	((3 - (5 + 9)) / (2 * 3))	((3 - (5 + 9)) / (2 * 3)) Correct file generated	Pass
2	2 2+	easy arithmetic	2+2	(2+2) Correct file generated	Pass
3	125 5 -	Multiple digit processing	(125 – 5)	(125 – 5) correct file generated	Pass
4	25 25+35/	tree with two operator nodes	((25 + 25) / 35)	((25 + 25) / 35) Correct file generated	pass
5	20 20+30 60*/	balanced tree, height 3, with div as root	((20 + 20) / (30 * 60))	((20 + 20) / (30 * 60)) Correct file generated	pass
6	2 2+3 6*+8 7+1 2/*-	balanced tree, height 3	(((3 * 6) + (2 + 2)) - ((1 / 2) * (8 + 7)))	(((3 * 6) + (2 + 2)) - ((1 / 2) * (8 + 7))) Correct File generated	pass
7	2+2	Test for one version of improper postfix notation	caught exception	IncorrectSyntaxException caught	pass
8	a * d	Test for invalid tokens	Caught exception	InvalidTokenException caught	pass
9	2 2 +3 5 &	Test for invalid operator	caught exception	InvalidTokenException caught	pass
10	3 5 9 + 2 3 * /	Test for having too few operators	warning message displayed	(((2 * 3) / 3) user warned of a wrong answer	pass
11	3 5 9 + - / 2 3 * /	Test for too many operators	Run-time exception	EmptyStackException	fail

Test Case 9:



Test Case 10:



Lessons Learned:

This entire project was a painful lesson on how valuable recursive method calls are. An iterative approach to generating a tree data structure would have probably spanned over one hundred lines of code; the recursive method was about ten. Visualizing the method calls was difficult, so drawing a picture on a white board was really what helped me see the juggling of data structures required. I think I could have simplified several parts of this code and generated the 3 address instructions while simultaneously generating the tree. However, just getting this project to a passable state was an enormous effort. In future endeavors, I will try to use fewer overall data structures so I can increase the overall efficiency of the application.