John Kucera

Prof. Ioan Salomie

CMSC 350 Data Structures and Analysis

6 February 2020

Week 4: Project 2 Solution Description

**3-Address Instructions GUI**

1. **Assumptions, Main design decisions, and error handling**

Like the previous GUI (P1GUI), I have assumed that anything can be entered. This includes extra spaces, characters, or any other possible tokens. I have also assumed that the only input to be accepted are operators + - \* /, digits 0 to 9, and a single space between each character. With that in mind, any additional white space in between will raise an Invalid Token Exception in addition to entering any tokens that are not among those I just mentioned. The integers are to be unsigned and any decimal points in the input will count as invalid tokens.

Assuming anything can be entered, I have tried to catch each possible error that could be entered through the GUI. This includes:

1. EmptyStackException - for cases where the parentheses and operators might be misplaced
2. NullPointerException – for cases where the input is empty
3. InvalidTokenException – for cases where the input expression includes tokens that are NOT + - \* / 0-9 and an empty space between each character
4. IOException – for cases where file writing is interrupted

The single spaces between each character are handled in ExpressionTree.java where the characters get tokenized and the spaces are bypassed. Additional space in between will raise an Invalid Token Exception. Additional white space after the expression will be bypassed.

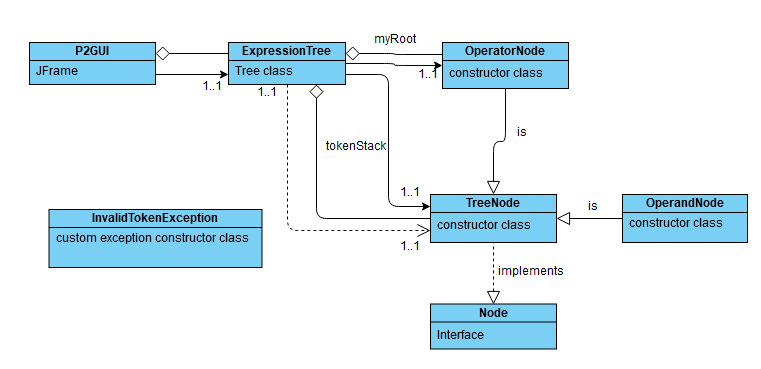
As for the main design decisions, I decided not to create an Operator abstract class (with AddOperator, SubOperator, MulOperator, DivOperator subclasses) because I could easily handle the differentiation of each operator with a simple switch statement. I did, however, include a Node.java interface, which allows all Nodes to have a value. TreeNode.java class implements Node and contains a constructor, allowing TreeNode objects to have a value, a left node, and a right node. TreeNode.java contains the inOrder() and postOrder() methods for (respectively) generating the infix expression and the 3-address instructions. It is also in TreeNode.java that the words for the operators (Mul, Div, Add, Sub) are generated in a switch statement.

TreeNode class is extended by OperatorNode.java and Operand.java classes. The difference between these is that Operator Nodes always have a value, a left node, and a right node, while an Operand Node will only have a value; Operand Nodes have their left and right defined as null because all Operands are leaves on the Expression tree. Finally, ExpressionTree.java is the class that constructs the Expression Tree, pushes the tokens into the stack in the proper order, and initiates the postOrder() and inOrder() methods on the root of that tree. Size of the tree is also calculated in ExpressionTree.java.

The way the program flows is like this: a postfix expression is entered in the GUI and the “Construct Tree” button is pressed. The postfix expression is passed through readInput(), the primary method of ExpressionTree, and the tree is constructed as the characters are tokenized and OperandNode/OperatorNode objects are generated and pushed onto a stack of type TreeNode. The last Node in the stack will be the root of the tree. Then, postOrder() is run on the tree using that root, which generates the 3-address instructions and prints them to Output.txt. The root is also used to calculate the number of nodes in the tree with a recursion method, which is then printed to Output.txt as well. Finally, the root is popped and used to run inOrder(), where the resulting infix expression will be generated. The infix expression is returned to the GUI and displayed there.

I designed Output.txt to NOT be cleared whenever a new GUI is started up. All new 3-address instructions will be appended to the file with no clearance of previous instructions.

1. **UML Class diagram**

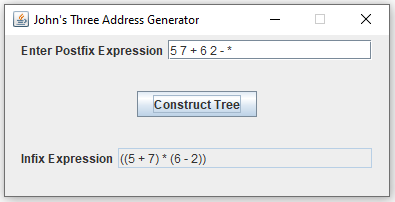


1. **Test Cases**

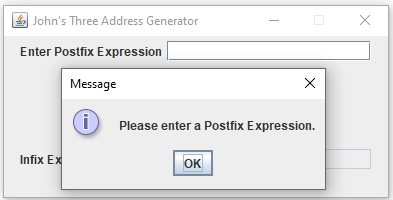
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **What aspect is tested** | **Input** | **Expected Output** | **Actual Output** | **Pass/**  **Fail** |
| If the user can close the GUI | Click “X” button on frame | GUI is closed and exited | GUI is closed and exited | PASS |
| Successful output of infix expression (Test 1) | 3 5 9 + - 2 3 \* /  press Construct Tree | (in GUI)  ((3 - (5 + 9)) / (2 \* 3)) | (in GUI)  ((3 - (5 + 9)) / (2 \* 3)) | PASS |
| Successful output of 3-address instructions and number of nodes (Test 1) | 3 5 9 + - 2 3 \* /  press Construct Tree  (with 5 empty spaces after expression) | (in Output.txt)  Add R0 5 9  Sub R1 3 R0  Mul R2 2 3  Div R3 R1 R2  Total nodes in the tree: 9 | (in Output.txt)  Add R0 5 9  Sub R1 3 R0  Mul R2 2 3  Div R3 R1 R2  Total nodes in the tree: 9 | PASS |
| Successful output of infix expression (Test 2) | 5 7 + 6 2 - \*  press Construct Tree | (in GUI)  ((5 + 7) \* (6 - 2)) | (in GUI)  ((5 + 7) \* (6 - 2)) | PASS |
| Successful output of 3-address instructions and number of nodes (Test 2) | 5 7 + 6 2 - \*  press Construct Tree  (with 3 empty spaces after expression) | (in Output.txt)  Add R0 5 7  Sub R1 6 2  Mul R2 R0 R1  Total nodes in the tree: 7 | (in Output.txt)  Add R0 5 7  Sub R1 6 2  Mul R2 R0 R1  Total nodes in the tree: 7 | PASS |
| Successful output of infix expression (Test 3) | 1 2 3 + - 4 5 6 7 \* + / -  press Construct Tree | (in GUI)  ((1 - (2 + 3)) - (4 / (5 + (6 \* 7)))) | (in GUI)  ((1 - (2 + 3)) - (4 / (5 + (6 \* 7)))) | PASS |
| Successful output of 3-address instructions and number of nodes (Test 3) | 1 2 3 + - 4 5 6 7 \* + / -  press Construct Tree  (with 10 empty spaces after expression) | (in Output.txt)  Add R0 2 3  Sub R1 1 R0  Mul R2 6 7  Add R3 5 R2  Div R4 4 R3  Sub R5 R1 R4  Total nodes in the tree: 13 | (in Output.txt)  Add R0 2 3  Sub R1 1 R0  Mul R2 6 7  Add R3 5 R2  Div R4 4 R3  Sub R5 R1 R4  Total nodes in the tree: 13 | PASS |
| InvalidToken Exception, additional white space in between | 3 5 9 + - 2 3 \* /  press Construct Tree  (spaces between – and 2 and between \* and /) | Message Dialog Box: Invalid Token: | Message Dialog Box: Invalid Token: | PASS |
| InvalidToken Exception, letters | 3 5 b + - 2 3 \* /  press Construct Tree | Message Dialog Box: Invalid Token: b | Message Dialog Box: Invalid Token: b | PASS |
| InvalidToken Exception, misc. characters | 3 5 9 > - 2 3 \* /  press Construct Tree | Message Dialog Box: Invalid Token: > | Message Dialog Box: Invalid Token: > | PASS |
| No Input | No input  press Construct Tree | Message Dialog Box: Please enter a Postfix Expression. | Message Dialog Box: Please enter a Postfix Expression. | PASS |
| Misplaced operators/ operands (test 1) | 2 \* 3  press Construct Tree | Message Dialog Box: Your Postfix Expression is invalid due to misplaced Operators or Operands. Please try again. | Message Dialog Box: Your Postfix Expression is invalid due to misplaced Operators or Operands. Please try again. | PASS |
| Misplaced operators/ operands (test 2) | / 9 4  press Construct Tree | Message Dialog Box: Your Postfix Expression is invalid due to misplaced Operators or Operands. Please try again. | Message Dialog Box: Your Postfix Expression is invalid due to misplaced Operators or Operands. Please try again. | PASS |

1. **Screenshots of program execution**

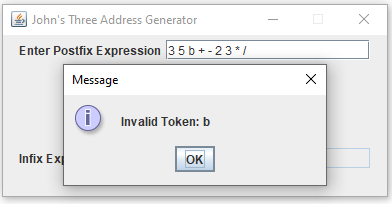
GUI, evaluating an infix expression:



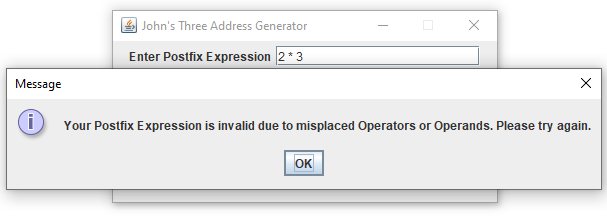
Empty input error:



Invalid Token Exception error:



Misplaced operators/operands error:

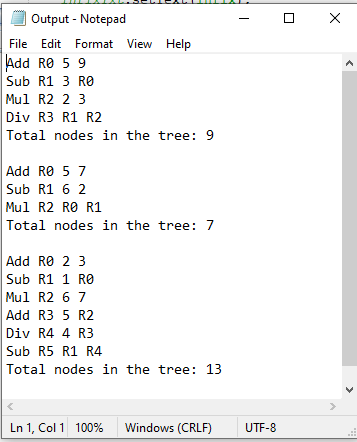


Output.txt: Uses 3-address instructions from these three postfix expressions:

3 5 9 + - 2 3 \* /

5 7 + 6 2 - \*

1 2 3 + - 4 5 6 7 \* + / -



1. **Lessons learned**

This was my first experience with Trees in computer science and I had a bit of a difficult time grasping the concepts involved with traversing through the tree. I fully understood the meaning behind the structures of binary trees, but the traversing methods (In Order vs. Pre Order vs. Post Order) got me stuck when I realized I did not know when to use which. They seemed pretentious at first, but when it came to using them for this particular project, I slowly got to understand why it is important that there are different directions of traversal. In fact, the fact that binary trees can be traversed in these different directions is what makes trees so unique. I understood this a little late, but definitely made it easier to write the methods for generate my infix expression and 3-address instructions.

One thing I understood a lot more while writing this program is that stacks are flexible in the way that they can hold objects of different types as long as inheritance is in play. For example, I was wondering at first how I would put operator nodes and operand nodes (not strings this time) in the same stack. Operator nodes have left and right nodes, but operand nodes are leaves. I realized I had to make a TreeNode class (and Node interface) to be extended by OperatorNode and OperandNode. I could then make a stack that holds TreeNode objects, which could consist of either OperatorNode or OperandNode. I never had this sort of mindset in the previous project where I was only making a stack of String objects. In turn, this also helped me understand the concept of inheritance a lot better.