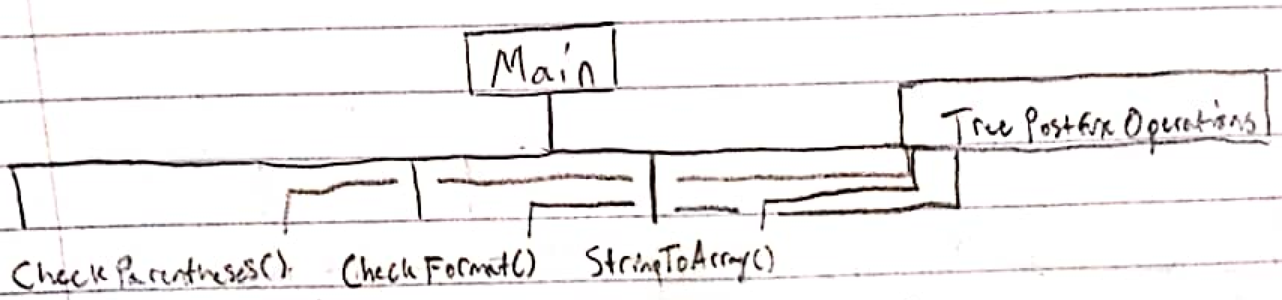
# Homework 04 - Expression Tree

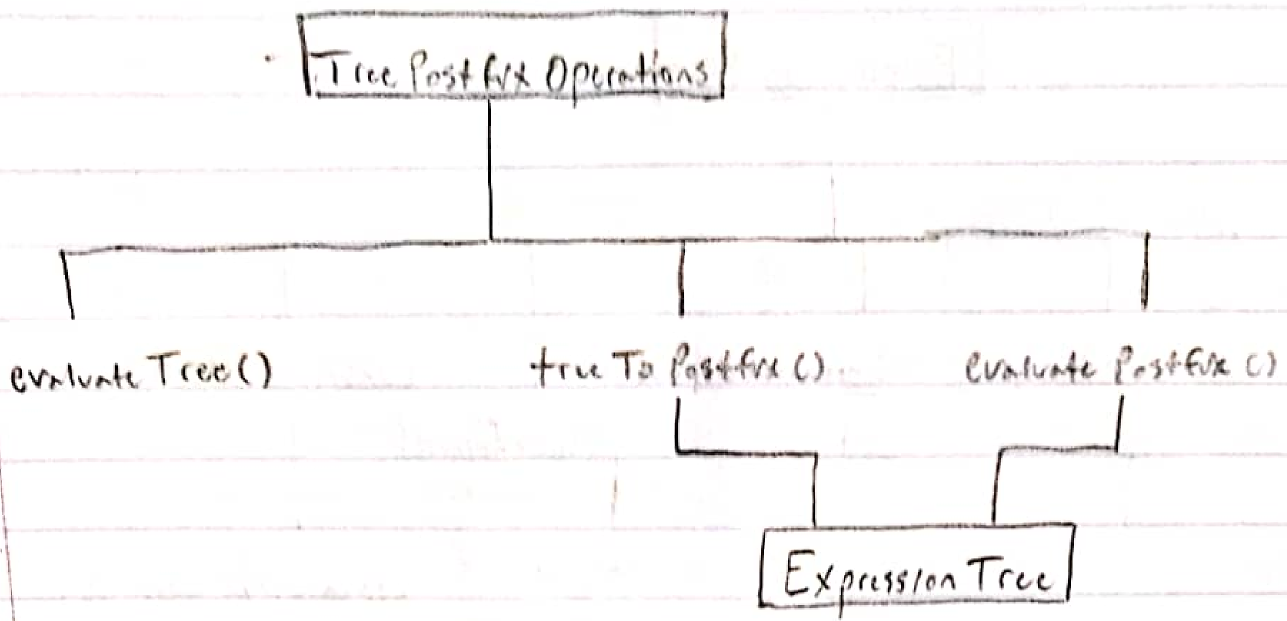
## Dependencies



- CheckParenthesis() can be tested by passing various valid and invalid expressions. If the boolean returned by the method does not match expectations, the method is incorrect.
  - Cases: Expression with no parentheses
    Expression with mismatched parentheses
    Expression with invalid open/close parentheses

- CheckFormat() can be tested by passing various valid and invalid expressions as above. Expression cannot contain spaces, but main() removes spaces before invoking the method. Boolean returned should match expectation
  - Cases: Expressions with letters or invalid symbols
    Expressions beginning or ending with operators
    Expressions containing two consecutive operators
    * Assignment description requests for unsigned integers, thus using signed values or floating points will break the method.

- StringToArray() This method parses an expression by building a buffer until a symbol is encountered. Passing a valid expression to the method and uncommenting the for loop at the bottom can validate correctness.
  * As above, signed values will break this method.

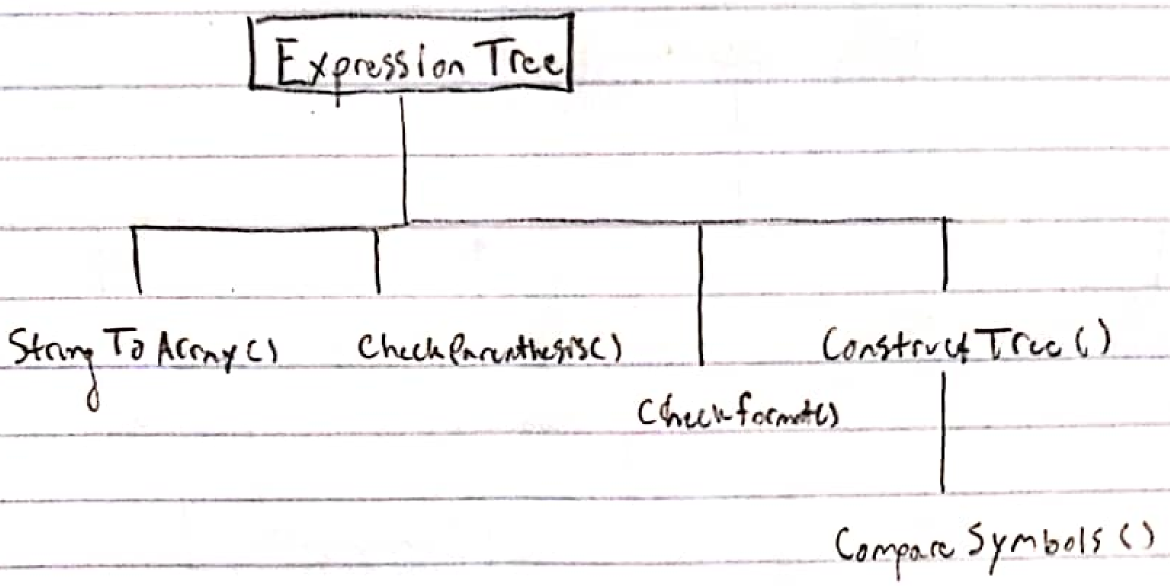


- evaluateTree() is the only static method in this class. It can be validated by invoking the method on a known, valid expression tree, or by constructing a valid tree using the ExpressionTree class. It is likely best to verify that ExpressionTree outputs a valid tree and to test using this output.

- treeToPostfix() Requires an object of type ExpressionTree. Given a valid expression tree, we can observe the output from the print statement in the method to confirm if the postfix notation can properly construct the input tree.

- evaluatePostfix() Requires an object of type Expression Tree. As above, given a valid expression tree, we can observe the return value from the method for correctness. The procedures can be compared to the output from evaluate tree for correctness. The postfix expression may also be evaluated by hand for correctness.

* All of the above methods include print statements which may be used to verify correctness.



- <u>CompareSymbols()</u> Checks whether the two given inputs are a matching set of parenthesis or operators of equivalent precedence. It can be validated completely by passing 6P2 symbols from the pool of (, ), %, *, +, - and evaluating the boolean that the method returns.
  - A short script can be written to evaluate these cases rather than perform this test manually 30 times.

- <u>ConstructTree()</u> May be evaluated by passing a valid array of expression
  - Construct a single node with a number input
  - Construct a single node with a simple expression "3/4"
    - Test this case using all operators and with parenthesis
  - Construct two nodes using the above steps and verify correctness
  - By induction, if the above cases are true any following cases should also hold true.

- other methods tested in Main class

Testing order

1.) All methods in main may be tested independently and are required for constructing a valid expression tree. Thus, they should be tested first.

2.) compareSymbols() may be tested for correctness independent of any other methods, but requires an object of type ExpressionTree. This object does not require any valid nodes to test the method for correctness.

3.) constructTree() requires either a valid array with an expression or for the above methods to be valid. Therefore, it should be tested after these methods.

4.) evaluateTree() may be tested on any valid expression tree whose root is assigned to TreePostfixOperations object's root. However, it is more likely that we should test the above first.

5.) treeToPostFix() also may be evaluated with a valid ExpressionTree. However, the above methods should be evaluated first.

6.) evaluatePostfix() can be validated by constructing an object of type ExpressionTree and assigning a valid array in postfix notation to the variable postfix. However, this is bad practice, so the above should be tested first.

7.) The main method runs test cases on the completed program which can be used to validate the functionality of all parts of the program.