# Assignment 4: Expression Tree

**Given: Monday, Nov 4th, 2018**

**Due Date: Sunday, Nov 18th, 2018**

**Time Due:** 11:59 pm (5% deducted if assignment is submitted anytime on Saturday)

**Hand in:** A folder (named *A3 LastName FirstName*), submitted to the I: drive. Failure to properly do this will result in loss of marks.

**Objectives:**

* Basic java object oriented assignment
* To gain experience working with recursion
* To become familiar with Expression Trees

## Assignment Overview

The purpose of this assignment is to design and build a program that will read in a user provided boolean expression and store the expression in an expression tree. It will have the ability to traverse the tree and print out postfix, infix, and prefix versions of the expression. It will also be able to simplify and evaluate the tree. Valid operators in a boolean expression will include the binary operators: & (AND), | (OR) and ^ (XOR) and the unary operator: ! (NOT), and operands will include the characters '0' (False) and '1' (True) or single character variables.

The assignment is broken down into three phases. In phase 1 you will read in a postfix boolean expression and build an expression tree from this expression. You will then traverse the tree and print out postfix, infix, and prefix versions of the expression. You will then evaluate the tree. In phase 2 you will read in an infix boolean expression and convert it to a postfix version from which you will build the tree. In phase 3 you will add the ability to simplify the expression tree. Completely solve each phase before moving on to the next phase and save a working version of each phase just in case.

For this assignment remember that your design does count for marks. You should try your best to do a good object oriented design. Try to break the program down into different classes that properly fit for each component of the assignment.

## Phase 1 Specifications

**Basic tasks checklist:**

1. Prompt the user to enter a postfix boolean expression and read it in (remember to do proper error handling)
2. Build an expression tree from the expression.
3. Traverse the tree and print out postfix (using postOrder traversal), infix (using inOrder traversal and adding brackets where appropriate), and prefix (using preOrder traversal) versions of the expression.
4. Evaluate the tree and print out the boolean value of the tree

**Detailed Task breakdown:**

1. Prompt the user to enter a postfix boolean expression and read it in

* A postfix expression is one in which the operator follows all of its operands. For example: ab&, a! and 10| which are postfix versions of the infix expressions: a & b, !a and 1 | 0.
* You must deal with the case where the expression entered is not a valid postfix expression. Parentheses are not needed and should not be allowed. Here is an algorithm to validate a postfix expression (from <http://stackoverflow.com/questions/789847/postfix-notation-validation>):

1. Initialize a counter to 0.
2. When you see a literal, increment the counter.
3. When you see a binary operator, decrement the counter twice, then increment it.
4. When you see a unary operator, decrement the counter, then increment it.
5. At the end of the string, if the counter is 1, and if it never went below 0, the string is a valid postfix expression.
6. Build an expression tree from the expression.

You should use a suitable implementation of a stack for this. In the java documentation for class Stack, it says “A more complete and consistent set of LIFO stack operations is provided by the [Deque](https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html) interface and its implementations, which should be used in preference to this class. For example:

Deque<Integer> stack = new ArrayDeque<Integer>();”

1. Traverse the tree and print out postfix (using postOrder traversal), infix (using inOrder traversal), and prefix (using preOrder traversal) versions of the expression

* You should use recursion for this.

1. Evaluate the tree and print out the boolean value of the tree

* You should use recursion for this.
* If there are variables in the expression, prompt the user for the value of each variable in order to evaluate the tree. You should only prompt the user once for each distinct variable. And you should create the tree with the variables and not simply replace the variables in the expression with the values and create the tree from this modified expression.

Example runs (with user input highlighted):

Expression Tree Test

Enter equation in postfix form

11^

Prefix : ^11

Infix : (1^1)

Postfix : 11^

Evaluated Result: false

Expression Tree Test

Enter equation in postfix form

ab&

Prefix : &ab

Infix : (a&b)

Postfix : ab&

Enter a value ('0' or '1') for variable: a

1

Enter a value ('0' or '1') for variable: b

1

Evaluated Result: true

Expression Tree Test

Enter equation in postfix form

1|0

Postfix : 1|0

Invalid postfix expression.

**Once the above tasks are complete and working properly move onto phase 2**

## Phase 2 Specifications

**Basic tasks checklist:**

1. Read in a user provided infix boolean expression and convert it to a postfix version from which you will build the tree.

**Detailed task breakdown:**

1. Read in a user provided infix boolean expression and convert it to a postfix version from which you will build the tree.

* Use the shunting-yard algorithm to convert the infix expression: <https://en.wikipedia.org/wiki/Shunting-yard_algorithm>
* The precedence of the boolean operators from highest to lowest will be: !, &, ^, |
* For error handling, you don't have to validate that the infix expression is valid but only that the resulting postfix expression is valid (which you should have implemented in phase 1). Although you are welcome to add additional validation if you wish

Example runs (with user input highlighted):

Expression Tree Test

Enter equation in infix form

!(1|a)

Prefix : !|1a

Infix : (!(1|a))

Postfix : 1a|!

Enter a value ('0' or '1') for variable: a

0

Evaluated Result: false