# HW3

**Due: Sun. Jun. 10th by 11:55 pm**

This assignment will be marked out of 30 points. You also have an opportunity to earn 5 bonus points. *This is an individual assignment. Please review the rules on academic misconduct on the course D2L page.*

This assignment consists of two parts. The first part consists of problems that will give you more practice working with binary trees. The second part consists of a coding exercise that explores the use of trees in parsing and evaluating arithmetic expressions.

# Part I

1. (5 points)
   Prove the following:
   - A *binary* tree with $n$ nodes has $n + 1$ `null` links.
   - In a non-empty binary tree, the number of full nodes plus one is equal to the number of leaves. (A *full node* is a node that has two children.)
2. (5 points)
   Design a linear time algorithm to test whether a binary tree is a binary search tree. Provide pseudocode for your algorithm and justify that your algorithm is linear in the number of nodes.
3. (5 points)
   Analyze the worst case and best case running times of inserting $n$ elements into an initially empty binary search tree. For each case, provide details of your analysis starting from counting relevant operations to an accurate asymptotic characterization.

# Part II

Write a parser that parses an arithmetic expression given in infix notation into a binary expression tree. Your parser should support infix expressions consisting of binary operators {`'+'`, `'-'`, `'*'`, `'/'` }, numbers (in integer format), as well as parentheses (which may be nested). Once an expression has been parsed into an expression tree, your program will additionally be able to perform preorder and postorder traversals on the tree, and evaluate the expression.

You can perform the parsing either using a stack (worth 15 points) or via recursive descent (worth 15 points + 5 bonus) using an expression grammar for infix expressions. Both strategies will be discussed in class and tutorials.

## Stack-based Parsing (15 points)

Please review Section 4.2.2 in Weiss which presents a stack-based algorithm for parsing parentheses-free expressions given in postfix notation. Note that you will first need to apply Dijkstra's shunting yard algorithm to convert the given infix expression to postfix.

- You may use the `Stack` class from the Java Collections Framework.
- Implementation of Dijkstra's shunting yard algorithm and the subsequent stack-based parsing algorithm must be your own. You are also *required* to use the provided `ExpressionTree` and `ExpressionTreeNode` classes (see below).
- Your implementation should check for errors and throw an exception if there are any syntax problems in the input expression.

## Recursive Descent (15 + 5 points)

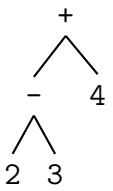
An expression grammar for infix expressions is given below.

```
E --> T { ( "+" | "-" ) T }
T --> F { ( "*" | "/" ) F }
F --> num | "(" E ")"
```

This grammar generates expressions consisting of the binary operators `+`, `-`, `*` and `/` as well as parenthetical expressions.

**Notation:** In the above grammar, braces `{}` are used to indicate productions that may repeat 0 or more times, `|` separate alternatives, while parentheses `()` are used for grouping. The only terminal symbol is a `num`. *For this assignment, you can assume that* `num` $\in \mathbb{Z}$.

Note that `*` and `/` have higher precedence compared to `+` and `-`. Operators with the same precedence are associated left-to-right (left-associativity), e.g. the expression `"2 - 3 + 4"` yields the following tree.




Before proceeding with your implementation, please make sure you understand the rules of the grammar and know how to apply them for parsing via recursive descent.

## Implementation

Write a Java class called `ExpressionTree` that parses expressions either using a stack or by recursive descent. In either case, your class should have the following public methods.

```
public class ExpressionTree {

  private ExpTreeNode root = null;

  // Builds empty tree
  public ExpressionTree() {
  }

  // Parse an expression from a string
  public void parse( String line ) throws ParseException {
  }

  // Evaluate an expression
  public int evaluate() {
    return 0;
  }

  // Return a postfix version of the expression
  public String postfix() {
    return "";
  }
```

```
  // Return a prefix version of the expression
  public String prefix() {
    return "";
  }
}
```

A skeleton of this class as well as the class `ExpTreeNode` are available on D2L.

### Additional Requirements

Include as many private methods as necessary to parse a given expression. You can assume that properly formatted input expressions will have tokens separated by whitespace. However, you should check for expressions that have syntax issues. If an input expression contains a syntax error, the `parse` method should throw a `ParseException`. In the thrown exception, include a description of the nature of the error as well as the location of the error in the input string.

When converting expressions to prefix or postfix, please separate tokens using whitespace. Prefix and postfix expressions should be parentheses free.

### Documentation and Testing

For all private methods that you add, please provide Javadoc comments to explain what the methods are doing.

Write a program to test your parser. Test with both small size expressions as well as expressions with mixed operators, nested parentheses etc. A test program that is compatible with your implementation will be made available closer to the due date.

# Submission

## Part I

Please prepare a PDF file called `part1.pdf` (scanned or typeset) that is packaged with the rest of the files for Part II. *If you are scanning your solution, please submit the scanned pages as a single PDF file.*

## Part II

Please package your Java class files in a compressed archive called `hw3.zip`. Please remove any driver programs (main methods) that you used to test your implementations. Please make sure that your name appears at the beginning of each file. If you need to cite any other sources that you used, please include them either at the beginning of the file `ExpressionTree.java` or in a separate file called `README`.

Please do not place the files in different folders. Your archive should only contain the following Java files: `ExpTreeNode.java`, `ExpressionTree.java`. In addition, your archive should also contain the file please `part1.pdf` and optionally a `README` file if applicable.