CSC-361
Lab 6

Building Expression Trees

This lab will have you implement code to build an expression tree using a bottom-up parsing technique. This technique is similar to how many compilers encode expressions.

Background: Postfix Representation and Evaluation of Expressions

Postfix notation (aka reverse Polish notation) is a way of encoding a mathematical expression without the need for grouping operators (i.e., parentheses are not required to encode the meaning of the expression). In postfix notation, the operator comes after the arguments it operates on. For example, using standard (infix notation) we write $5 + 3$. However, in postfix notation, we write $5\ 3\ +$.

As a larger example, we would write the expression $3 - (4 + 6)$ in postfix notation as $3\ 4\ 6 + -$; note in the postfix expression we do not require parentheses to indicate order of operations. We would thus evaluate the postfix expression as:

$$3\ 4\ 6 + - \quad \rightarrow \quad 3\ 10 - \quad \rightarrow \quad -7$$

That example is not very illustrative of the actual mechanics. To evaluate such an expression, we would use a stack. Reading from left to right, when we encounter a numeric value, we push onto the stack. When we encounter a binary operator, we pop the stack twice, and evaluate with the operation on the two popped values. The result is then pushed back on the stack. Figure 1 steps through the use of the stack to evaluate the postfix expression $3\ 4\ 6 + -$.

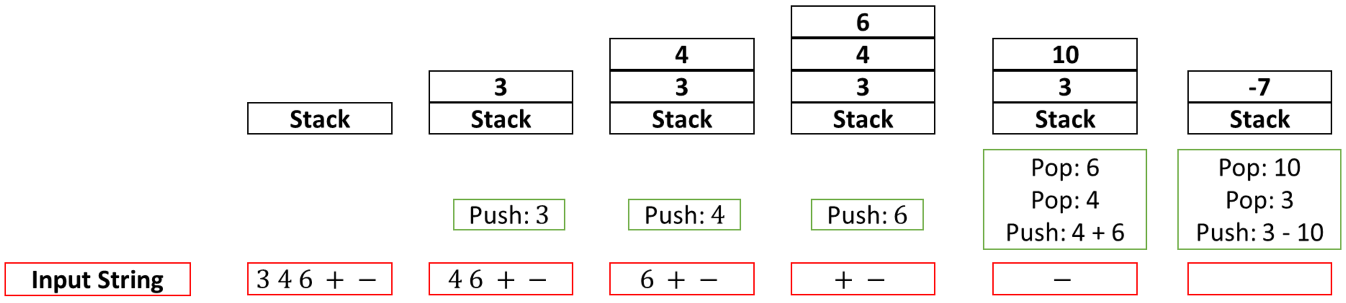


Figure 1: Stack-based evaluation of the postfix expression $\mathbf{3\ 4\ 6} + -$.

After the expression is evaluated, the result remains on the stack and can be popped accordingly. In this lab our goal is not evaluation, but construction of a tree-based representation of the input expression.

What You Need To Do (Step 1): Construction of an Expression Tree

We can use this stack-based technique to construct a tree representation of such mathematical expressions. We refer to these trees as *expression trees or abstract syntax trees* (ASTs). As an example, we refer to Figure 2 illustrating the AST-based representation of our expression $3\ 4\ 6 + -$.

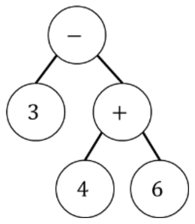


Figure 2: AST representation of the postfix expression $\mathbf{3\ 4\ 6} + -$.

In Figure 2, notice that *all* leaves of the tree are numeric values. That is, operators are always internal nodes in the tree. Second, associativity of expressions like $(4 + 6)$ is encoded in the parent-child relationship between nodes; that is, the $+$ node in the AST has two children, respectively: 4 and 6. Last, we observe that operations that occurred near the end of the input string are higher up the tree compared to other operations. Evaluating the expression using the tree then becomes a post-order traversal of the AST. For example, we cannot evaluate the subtraction operation in Figure 2 until we know the result of the addition operation lower in the tree.

Our Language

Our target language will consist of the binary operators and unary operators listed in Table 1. It is important to note that some of the unary operators will be input as prefix expressions while other unary operators are interpreted in a postfix manner.

Table 1: Operators and Operations for our Expression Language

Binary Operator	**Description**	**Postfix Example**	**Infix Evaluation of Example**
$+$	Binary Addition	$2\ 3\ +$	$2 + 3 \rightarrow 5$
$-$	Binary Subtraction	$2\ 3\ -$	$2 - 3 \rightarrow -1$
$*$	Binary Multiplication	$2\ 3\ *$	$2 * 3 \rightarrow 6$
$/$	Binary Float Division	$2\ 3\ /$	$2\ /\ 3 \rightarrow 1.5$
$**$	Binary Exponentiation	$2\ **\ 3\ =\ 8$	$2^3 \rightarrow 8$
Prefix Unary Operator	**Description**	**Prefix Example**	**Infix Evaluation of Example**
$+$	Unary Addition	$+4$	$+4 \rightarrow 4$
$-$	Unary Subtraction	-4	$-4 \rightarrow -4$
Postfix Unary Operator	**Description**	**Example**	**Infix Evaluation of Example**
A	Unary Absolute Value	$5\ 8\ -\ \mathrm{A}$	$\lvert 5-8 \rvert \rightarrow \lvert -3 \rvert \rightarrow 3$
S	Unary Square Root	$16\ \mathrm{S}$	$\sqrt{16} \rightarrow 4$

As an example, consider a postfix expression represented as an AST in Figure 3.

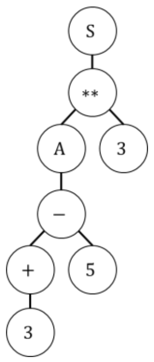


Figure 3: Expression tree representation of the expression $\mathbf{+3\ 5\ -\ A\ 3\ **\ S}$.

This expression contains a binary minus operator ($-$) as well as the binary exponentiation operator ($**$) and can be observed as the nodes with two children in Figure 3. However, it also boasts the unary absolute value operator (A) and unary square operator (S): two nodes in Figure 3 with one child. Last, Figure 3 employs a unary plus ($+$) with a child node of 3 at the bottom of the expression tree.

If we traverse this tree using a post-order traversal (evaluating from the bottom-up), we would arrive at the mathematical expression equivalent to $\sqrt{|+3-5|^3} \rightarrow \sqrt{8}$.

Our Language in Code

There are many ways to represent such an expression tree. Our implementation will use a hierarchy of node classes as listed in Table 2. All classes derive from the abstract `ASTNode` class.

Table 2: Classes used to define the AST nodes our expression trees.

Abstract AST Node	Subclasses	Instance Variables
`ASTnode`	`LiteralNode` `OperationNode`	
`LiteralNode`	`IntLitNode` `RealLitNode`	`_value: double`
`OperationNode`	`BinaryExprNode` `UnaryExprNode`	

`BinaryExprNode`	`BinaryPlusNode`
	`BinaryMinusNode`
	`BinaryTimesNode`
	`BinaryDivideNode`
	`BinaryPowerNode`
		`_left: ASTnode`
		`_right: ASTnode`

`UnaryExprNode`	`UnaryPlusNode`
	`UnaryMinusNode`
	`UnaryAbsoluteValueNode`
	`UnarySquareRootNode`
		`_expr: ASTnode`

The hierarchy is relatively intuitive in that all operations are instances of `OperationNode` and all numeric values (also known as literals) are instances of the `LiteralNode` class. All classes directly or indirectly inherit from `ASTnode`. As an example, consider the tree-based representation of the expression $+3\ 5\ -\ \mathrm{A}\ 3\ **\ \mathrm{S}$ from Figure 3 alongside the object-based representation. We observe in Figure 4 that expressions are mirrored in their tree- and object-based representations: `LiteralNode` objects are leaves and `OperatorNode` objects are internal nodes.

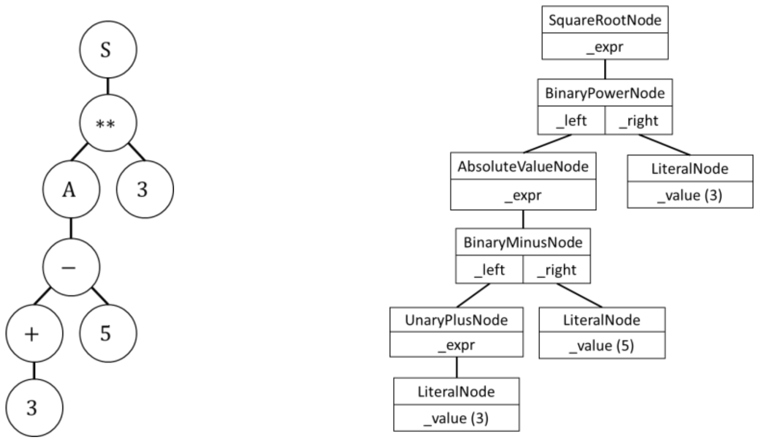


Figure 4: Expression Tree and corresponding ASTnode-based representation of $\mathbf{+3\ 5\ -\ A\ 3\ **\ S}$.

Our Input

Each input expression will be via a string. Each postfix operator and literal will be separated by a space (except in one case described below). For example, `"5 3 +"` will be interpreted as the expression `5 + 3`. As a more complex example,

`"6 2 34 5 + - *"` will be interpreted as `6 * (2 - (34 + 5))`.

There is one significant ambiguity that must be resolved with our language: distinguishing between prefix unary $+$ and $-$ operators and binary $+$ and $-$ operators. We will use the following rule to properly disambiguate these operators in our input strings: *prefix operators will not be separated by spaces*. For example, unary $+$ with 3 will be input as `"+3"` with no space between the characters. As a more complex example, all the $+$ and $-$ operators in the expression `"+-+-+--++-3"` are unary prefix operators. As a last example,

`"+-2.0 -+4 ** S"` will be interpreted as $\sqrt{(-2)^{-4}}$.

You may assume all input in your program conforms to the specification described in this document. That is, you do not have to worry about error-handling associated with malformed input strings.

What You Need To Do (Step 2): Verification

In order to verify the construction of a tree, we will implement common functionality for tree construction techniques called *unparsing*. The idea of unparsing is to traverse the tree and output the contents of the tree as a string so we can verify that our input matches the object-based representation. In this case, our goal will be to unparse the tree so that the input string will match *exactly* the output string. Since our original strings were encoded as pseudo-postorder expressions, we can acquire an unparsing using a (mostly) post-order traversal of the tree. The only case, when we use a pre-order traversal is for the prefix unary operators ($+$ and $-$).

Here is a sample of debugging output from a unit test for building a tree and unparsing:

```
In:  |1.0 A 2.0 S 3.0 A 4.0 S -+5.0 + - * / A ----12.3 + 4 **|
Out: |1.0 A 2.0 S 3.0 A 4.0 S -+5.0 + - * / A ----12.3 + 4 **|
```

This output was acquired from the following code that constructed a tree and unparsed it:

```java
if (DEBUG) System.out.println("In:  |" + in + "|");

// BUILD
ASTnode tree = ASTbuilder.build(in);

// UNPARSE
StringBuilder unparsed = new StringBuilder();
tree.unparse(unparsed);

// COMPARE
if (DEBUG) System.out.println("Out: |" + unparsed.toString() + "|");
```

Provided Source Code: Where to Start

`Constants` is a class that contains definitions of `final static` variables that are to be used project-wide. For example, do not use `"S"` in code when referring to square root. Instead, use `Constants.SQUARE_ROOT`.

`ASTnode` contains the class definitions of all classes in the AST (as described above and listed in Table 2). You will need to implement `unparse` methods for each of the applicable classes.

The sole purpose of the `ASTbuilder` class is to construct an expression tree from an input string as described above. Your main `build` method must be `static` as used in the code listing above. You are encouraged to implement many (static) support methods to facilitate parsing and construction of expression trees.

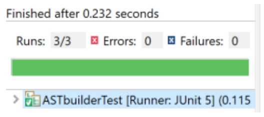
A set of unit tests has been provided in `ASTbuilderTest`; feel free to add tests, but do not modify any of the existing tests. You can (and should) note that the constant `DEBUG` can turn on / off output from testing. Please use it while debugging, but do not submit evidence that includes such output.

JUnit Testing

No output should be produced by your tests; we are seeking only a 'green' output indication in Eclipse. Make sure your screenshot shows success of all junit testing methods and not just the overall summary.

Submitting: Source Code

Your code should be well documented, including docstring comments of methods, blocks of code, and header comments in each file. Junit tests should have reasonable String messages output, if failure occurs. For example, the image to the right is bad.



```
Finished after 0.232 seconds
Runs: 3/3   Errors: 0   Failures: 0
> ASTbuilderTest [Runner: JUnit 5] (0.115
```

Header Comments

Your program must use the following standard comment at the top of *each source code file*. Copy and paste this comment and modify it accordingly into these files.

```
/**
 * Write a succinct, meaningful description of the class here. You should avoid wordiness
 * and redundancy. If necessary, additional paragraphs should be preceded by <p>,
 * the html tag for a new paragraph.
 *
 * <p>Bugs: (a list of bugs and / or other problems)
 *
 * @author <your name>
 * @date   <date of completion>
 */
```

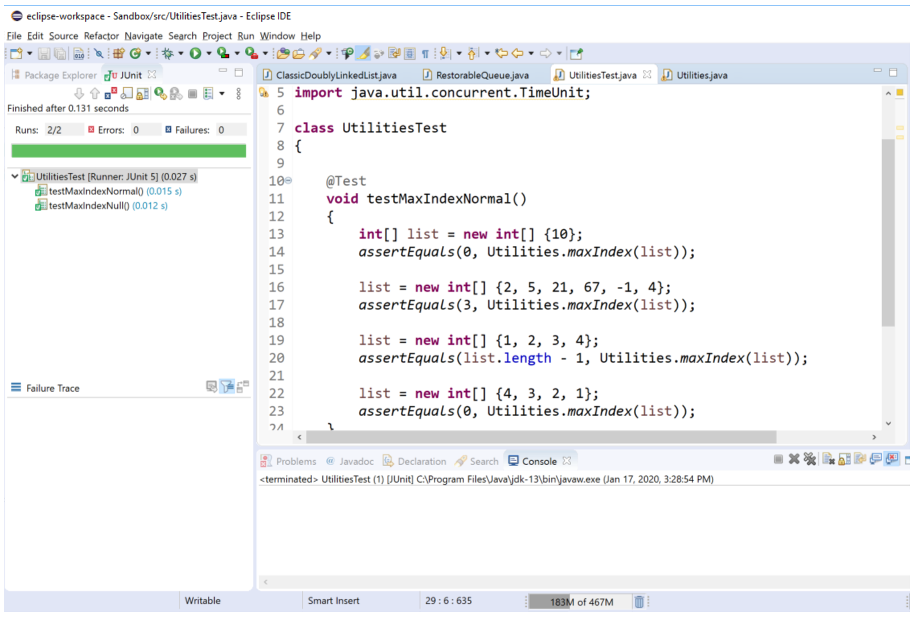
Inline Comments

Comment your code with a *reasonable amount of comments* throughout the program. Each method should have a comment that includes information about input, output, overall operation of the function, as well as any limitations that might raise exceptions; Javadoc comments are ideal. Each *block* of code (3-4 or more lines in sequence) in a function should be commented.

It is ***prohibited*** to use *long* comments to the right of lines of source code; attempt 80 character-wide text in source code files.

Submitting: Proof of Program Execution

Execute your code and take a screenshot of the associated junit window and output console (with no output). Place these screenshots into a word processing document (Word, OpenOffice, GoogleDocs, etc.). If multiple screenshots are necessary, label each clearly. Please make sure to crop and enlarge the screenshots so that the picture and / or text is clear (and doesn't strain my old eyes). For example,


eclipse-workspace - Sandbox/src/UtilitiesTest.java - Eclipse IDE

Finished after 0.131 seconds

Runs: 2/2 Errors: 0 Failures: 0

```
UtilitiesTest [Runner: JUnit 5] (0.027 s)
    testMaxIndexNormal() (0.015 s)
    testMaxIndexNull() (0.012 s)
```

```java
import java.util.concurrent.TimeUnit;

class UtilitiesTest
{

    @Test
    void testMaxIndexNormal()
    {
        int[] list = new int[] {10};
        assertEquals(0, Utilities.maxIndex(list));

        list = new int[] {2, 5, 21, 67, -1, 4};
        assertEquals(3, Utilities.maxIndex(list));

        list = new int[] {1, 2, 3, 4};
        assertEquals(list.length - 1, Utilities.maxIndex(list));

        list = new int[] {4, 3, 2, 1};
        assertEquals(0, Utilities.maxIndex(list));
    }
```

```
<terminated> UtilitiesTest (1) [JUnit] C:\Program Files\Java\jdk-13\bin\javaw.exe (Jan 17, 2020, 3:28:54 PM)
```

Create a PDF of this document and call it `evidence.pdf`.

Source Code Files

Please submit only two source code files: `ASTbuilder` and `ASTnode`.

Final Submission File

In a folder named `lab`, place (1) the source code files and (2) `evidence.pdf`. Zip folder `lab` and label that zip file as `lab.zip`. This zip file is to be submitted via Moodle.

Please be reminded that following instructions explicitly and submitting well-formatted documents (with consistent fonts and typeset equations) is an important part of professionalism.

Only one person in your pair needs to submit the final product.