Assignment 3a: Order of Operations

Data Structures, CSCI-UA 102, Section 7
Instructor: Max Sklar

Interpret an artimetical expression and build a corresponding expression tree.

Goals

- Work with expression trees.
- Use stacks can be used to build expression trees while respecting the order of operations.
- Use recursion on tree nodes.

Overview

Your goal is to take an arithmetic expression with variables - like $5 + 2 * x$ and turn it into an expression tree. After that, perform the following operations:

- Print out the inorder traversal of the tree to confirm that it was created properly
- Print out the postorder traversal of the tree to get the postfix expression
- Do partial evaluation, and keep track of which unbound variables appear in the expression.

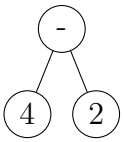
Background

An *arithmetic expression* can be thought of as code in the language of math. It is a combination of operators (usually binary like +) and operands to express a calculation. Some examples are `5 + 1`, `3 * (1 - 2)`, `5^8/4+1`, and `7`.

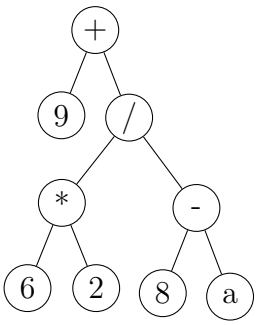
For the purposes of this assignment, we will restrict ourselves to 5 binary operations: `+`, `-`, `*`, `/`, and `^`. We will also consider arithmetic expressions with variables, such as `3*x+2`

An *expression tree* is a data structure used to represent an arithmetic expression. There are two types of nodes in an expression tree: operators and operands. *Operand nodes* contain either numbers or variables. They are the *leaves* in expression trees (they have no children).

Operator nodes have two children that are also expression trees (each child can be an operator or an operand). For example, the expression `(4 - 2)` will be represented by an expression tree with a root operator node. It'll store the fact that it's the subtraction operator, and it will have two children, each number nodes 4 and 2.



A more complicated expression will have more nodes associated with it. Consider the expression `9+6*2/(8-a)`. This expression is the sum of two other expressions, so the root will be an operator node, with operator "+", and two child expressions. The left child will be the leaf node with "9", but the right child is going to be an operator node with "/". It's children are also operator nodes: the left child is 6*2 and the right child is 8-a.



Order of Operations

In arithmetic, the order of operations—often remembered by the acronym PEMDAS—defines the sequence in which parts of an expression are evaluated. This order ensures that expressions with multiple operators are evaluated consistently.

- **Parentheses** (P) are evaluated first, allowing expressions within parentheses to be handled independently of surrounding operators.
- **Exponents** (E) (`^`), are evaluated next.
- **Multiplication** (M) and **Division** (D) are processed third, from left to right as they appear in the expression.
- **Addition** (A) and **Subtraction** (S) complete the evaluation, also from left to right.

With these rules, there can be no mistake about how to evaluate an expression, and by extension which tree represents it.

Another way to look at this is to assign each operator a *precedence*, or score to determine when it is evaluated. The purpose of the parentheses is to override these precedence values.

Operator	Symbol	Description	Precedence
Exponent	`^`	Raises a base to an exponent	High
Multiply	`*`	Multiplies two values	Medium
Divide	`/`	Divides two values	Medium
Add	`+`	Adds two values	Low
Subtract	`-`	Subtracts two values	Low

Table 1: Operator Precedence Table

Code Provided

The main function exists in ExpressionInterpreter.java, which takes an arithmetic expression as its only argument. Because aritmetic expressions often contain spaces, it will be put around quotes in order to be run from the commant line

```
java ExpressionInterpreter "1 + 6 * 9"
```

ExpressionInterpreter.java is given to you. The first thing that it does in the main function is to turn the input expression into tokens (using the given Tokenizer.java), which is a singly linked list of strings, each representing an operator, an operand, an open parenthesis, or a close parenthesis.

A `Node` class is given to you in Node.java to represent expression trees.

Expression Tree Methods

The following methods should be implemented in the `ExpressionInterpreter.java` class. You may use any method signatures you prefer, as long as it achieves the correct result.

- `Node buildExpressionTree()`: Takes a `SinglyLinkedList<String>` of tokens representing an arithmetic expression, constructs an expression tree from the tokens, and returns the root node of the tree. (See Steps to Construct the Expression Tree)
- `void printInOrder(Node root)`: Performs an inorder traversal of the expression tree and prints each node's value in the correct order. The output of each operator node should be enclosed in parentheses, and the operators themselves should be surrounded by spaces. Parentheses should not be added around individual operands. Although this may result in more parentheses than the original expression, it ensures that the expression can be evaluated without relying on operator precedence. (See Sample Output)
- `void printPostOrder(Node root)`: Performs a postorder traversal of the expression tree and prints each node's value in postorder sequence. (See Sample Output)
- `Node solveAsMuchAsPossible(Node root)`: This function solves the expression tree as much as possible. If there are no variables in the expression tree, it should end up with a single operand node which is the result of the arithmetic computation. If there

are variables, then that part of the expression tree should not be solved. Instead, the program should print out `Unbound variable: x`, where instead of x it is the name of the variable. It may print out multiple "Unbound variable" lines, and it may observe the same variable multiple times in an expression. (See Sample Output)

Algorithm for Building an Expression Tree from a List of Tokens

This algorithm builds an expression tree by using stacks to manage operator precedence. This approach ensures that operands and operators are processed in the correct order. You will be turning this algorithm into code. As you work on it, observe how it works and why it produces the correct result.

Steps to Construct the Expression Tree

The algorithm uses two stacks to keep track of operators and operands as it processes each token in the list. Here is a step-by-step breakdown:

1. **Initialize Stacks**:
   - Create two empty stacks. The first is a stack of `Node` objects called `expressionStack`. Each element in `expressionStack` represents a partial expression tree.
   - The second stack is a stack of `String` objects, referred to as the `operatorStack`. This stack stores operators (e.g., `+`, `/`) and parentheses as they appear in the token list.
2. **Push Initial Parenthesis on the Operator Stack**:
   - Push an open parenthesis `(` onto the `operatorStack`.[1]
3. **Define the Helper Method** `popOperatorStack()`:
   - The `popOperatorStack()` method pops an operator from `operatorStack` and uses it to build a new subtree in `expressionStack`. It proceeds as follows:
     (a) Pop the top operator from `operatorStack`.
     (b) Pop the top two nodes from `expressionStack`.
     (c) Create a new operator node with the popped operator as its element. Set the two nodes from the previous step as its right and left children, respectively.
     (d) Push this new node back onto `expressionStack`.
4. **Process Each Token Sequentially**:
   (a) If the token is an operand (either a number or a variable):

[1] The open parenthesis acts as a sentinel value, allowing the entire expression to be treated as if it were enclosed in parentheses.

- Create a leaf node for the operand and push it onto `expressionStack`.

(b) If the token is an open parenthesis `"("`:

- Push it directly onto `operatorStack`.

(c) If the token is an operator (one of `+`, `-`, `*`, `/`, or `^`):

- Attempt to push the operator onto `operatorStack`. This can only be done if the top element on `operatorStack` has a strictly lower precedence or is an open parenthesis.
- If the precedence condition is not met, call `popOperatorStack()` repeatedly until the condition is met.[2]

(d) If the token is a closed parenthesis `)`:

- Call `popOperatorStack()` repeatedly until the top of `operatorStack` is an open parenthesis.
- Pop and discard the open parenthesis.

5. **Finalize the Expression Tree**:

   - To complete the expression tree, process a final closed parenthesis to match the initial open parenthesis from Step 2.

6. **Validation and Final Steps**:

   - At this point, `operatorStack` should be empty, and `expressionStack` should contain exactly one node. If `expressionStack` does not have a size of one, this indicates either mismatched parentheses or a malformed expression.
   - Pop the remaining node from `expressionStack`. This node is the root of the completed expression tree, representing the entire expression.

Sample Output

```
> java ExpressionInterpreter "1 + 1"
Postfix: 1 1 +
Infix: (1 + 1)
Solved: 2.0

> java ExpressionInterpreter "1 * 2 / 5 + 1 - 3 * 4 ^ 0"
Postfix: 1 2 * 5 / 1 + 3 4 0 ^ * -
Infix: ((((1 * 2) / 5) + 1) - (3 * (4 ^ 0)))
Solved: -1.6

> java ExpressionInterpreter "m * x + b"
Postfix: m x * b +
```

[2]Eventually, the operator stack will be able to recieve the new operator.

```
Infix: ((m * x) + b)
Unbound Variable: m
Unbound Variable: x
Unbound Variable: b
Solved: ((m * x) + b)

> java ExpressionInterpreter "x^2 - x - 1"
Postfix: x 2 ^ x - 1 -
Infix: (((x ^ 2) - x) - 1)
Unbound Variable: x
Unbound Variable: x
Solved: (((x ^ 2) - x) - 1)

> java ExpressionInterpreter "(3 + 5) / (3*3 - 1)"
Postfix: 3 5 + 3 3 * 1 - /
Infix: ((3 + 5) / ((3 * 3) - 1))
Solved: 1.0
```

Hints

- Use `Math.pow(x, y)` to evaluate `x^y`.
- The main function as given in ExpressionInterpreter.java may not need to be changed at all, and you might be able to get it working by implementing the functions it calls. You may change it if you decide to solve the problem in an alternative way.
- Your code will not be tested on incorrect input, but you are encouraged to try inputs that are malformed to see what your code does.

Submission

The entire submission should be a zip file containing all relevant files and folders and should be uploaded to Brightspace by the due date. **The zip file should be of the form [netid].zip where you insert your netid**. The zip file includes:

- Your code, which should include one or more Java files and may or may not be organized into packages. This includes code provided for the assignment.
- Your `readme.txt` file.

README

Included in the `readme.txt` file should be the following:

RUNBOOK: Information on how to compile and run your code in the command line.

TIME SPENT: An estimate of the time spent working on this project.

NOTES: Include the difficulties and roadblocks encountered while working on this project, including notable bugs that needed to be overcome. Are there any quirks in your solution that we should be aware of? Was the outcome what you expected?

RESOURCES AND ACKNOWLEDGEMENTS: Disclose the sources you consulted for help on this project. Possible human sources include: NYU Staff (the instructor, TA, tutor), other students, and anyone who helped you with the project. Sources that you consulted, including code that you found online, in the book, and output from LLMs. Describe the nature of your LLM usage.

There will never be a grading penalty for a complete and accurate accounting of resources and acknowledgments.