Lab #1 — Building a Calculator

CS 152 Section 5 — Fall 2020
Michael McThrow
San José State University

Your task for Lab #1 is to build a calculator in Java that has two modes, controllable by command-line options `postfix` and `infix`. In both modes the calculator will read lines of input from standard input. For each input line, the calculator will evaluate the expression that was input and print the result on a separate line. The program ends when it encounters the EOF character, similar to the behavior of most Unix utilities.

To simplify matters regarding dealing with different types of numbers, in this assignment, all numbers are to be represented internally as either Java `float` primitives or `Float` objects depending on your implementation choices.

Your calculator will implement the following operations:

- +, -, *, /
- ^ (exponentiation: e.g., 2^3 = 8).

Your calculator will implement expressions containing either a sole number or operations applied to numbers.

Mode #1 — Postfix Notation

The first mode that you will implement is one where the calculator accepts an expression that is expressed in *postfix* notation. For those of you familiar with HP's line of scientific and graphing calculators, *postfix notation* is also known as RPN or Reverse Polish Notation, in honor of the Polish logician Jan Łukasiewicz who invented Polish notation, also known as prefix notation.

Here are examples of expressions written in postfix notation with their conversions to infix notation and their evaluations:

`2 3 +`	`2 + 3`	`5`
`2 3 + 5 *`	`(2 + 3) * 5`	`25`
`2 3 5 * +`	`2 + (3 * 5)`	`17`
`2 3 2 ^ * -10 -`	`2 * (3 ^ 2) - -10`	`28`

How an RPN calculator works internally is as follows: it maintains an internal stack that is used to store operands and intermediate results. Let's use the expression "4 3 + 5 *" as an example. The first thing that we do is lexical analysis on the input string by first splitting the string by its whitespace characters and then performing the proper type conversions on the numbers, resulting in a list that looks like this:

```
[4.0, 3.0, "+", 5.0, "*"]
```

Next, we iterate through the list. For each number, we push it onto the stack. Once we reach an operator on the list, we pop the stack twice, perform that operation on the popped numbers, and then push the result onto the stack. In this example, the elements 3.0 and 4.0 are popped from the stack. We then perform the "+" operation on the second and first elements popped from the stack (order matters for "-", "/", and "^"), and then push the result (12.0) onto the stack. Then, as we continue iterating through the list, we encounter 5.0, and thus we push it on the stack, resulting in a stack with the elements 12.0 (bottom) and 5.0 (top). Finally, the last token in the list is "*", and so we pop the stack twice, multiplying 5.0 and 12.0 to get 60.0, and then we push it back on the stack.

When we have exhausted the list of tokens, we pop the stack and print the popped value as the result of the expression.

One of the nice properties of postfix notion is the lack of a need to specify operator precedence. It is this property that makes it possible to implement an RPN calculator without the need for specify a formal grammar for expressions. In fact, there are full-fledged programming languages such as Forth and PostScript that use postfix notation and rely on a stack.

Mode #2 — Infix Notation

Unlike a postfix calculator where parsing is a very straightforward task, parsing is not as straightforward in infix notation, since you have to concern yourself with expressions of arbitrary length and operator precedence. Your calculator will have to properly implement the PEMDAS (parentheses, exponents, multiplication, division, addition, subtraction) order of operations that are from elementary algebra.

Thankfully with the help of parser generators such as ANTLR, you won't have to deal with the labor of implementing parsing algorithms.

Here is an excellent tutorial for using ANTLR: https://tomassetti.me/antlr-mega-tutorial/. Please also refer to the main ANTLR website at https://www.antlr.org.

Your goals are the following:

1. Write a BNF or EBNF grammar that is able to represent expressions in infix notation that is also able to implement the PEMDAS order of operations.
2. Express that grammar as an ANTLR grammar.
3. Use ANTLR to generate an abstract syntax tree.
4. Traverse the abstract syntax tree to evaluate the expression. There are two ways of doing this: (1) either evaluating the abstract syntax tree directly, or (2) using the AST to generate a postfix notation representation of the expression, and then evaluating it as in Mode #1.

Some Examples:

```
$ java Calculator postfix
Calculator> 2 4 * 2 ^ 10 -
```

```
54
Calculator> 5
5
Calculator> 8 24 + 9 -
23

$ java Calculator infix
Calculator> (2 * 4)^2 – 10
54
Calculator> 5
5
Calculator> 8 + 24 – 9
23
```

Deliverable:
A collection of Java and ANTLR source code files in a *.zip archive, where the main method is located in a class called `Calculator` and in a file called `Calculator.java`.

Grading Rubric:
Mode #1 (Postfix Notation): 30%
Mode #2 (Infix Notation Grammar and Parsing): 50%
Mode #2 (Infix Notation Evaluation): 20%

Note: Your code must compile in order for it to receive any credit; any submission that does not compile will receive a grade of 0%.