Project #1 — Building Calculators

CS 152 Section 6 — Fall 2021
Michael McThrow
San José State University

PAIRS ALLOWED: You may choose to work in pairs on this project. Keep in mind that once you begin working as partners, you may not change partners during the duration of the project, though you may either choose a different partner or work alone in the next project that allows pairs.

Your task for Project #1 is to build two calculator programs in Java:

1. An RPN (postfix notation) calculator that is in a file called `RPNCalculator.java`.
2. An infix notation calculator that is in a file called `InfixCalculator.java`.

In both calculators, they will read a single line of input from standard input (`System.in` in Java). The calculator will evaluate the expression that was input and print the result on a separate line.

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

Your calculator will implement the following operations:

- +, -, *, /
- ^ (exponentiation: e.g., 2^3 = 8).
- sqrt (square root; corresponds to `Math.sqrt()` in Java)
- ln (natural logarithm; corresponds to `Math.log()` in Java)
- abs (absolute value; corresponds to `Math.abs()` in Java)

Note that +, -, *, /, and ^ are binary operators (i.e., they operate on two elements), while sqrt, ln, and abs are unary operators (i.e., they operate on one element).

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

Postfix Notation for RPNCalculator

RPNCalculator accepts expressions 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. (You will have a lot more experience with prefix notation when we start programming in Scheme during Projects #2 and #3.)

If you have access to a Unix system, I recommend playing around with the `dc` command to get a feel of what RPN calculators are like. It's one of my favorite Unix commands.

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 a binary operator on the list (+, -, *, -), we pop the stack twice, perform that operation on the popped numbers, and then push the result onto the stack. (For unary operators like `sqrt`, `ln`, and `abs`, you will only pop the stack once and then perform the operation on that one number.) In this example, the elements 3.0 and 4.0 are popped from the stack; let's set variables `first = 3.0` and `second = 4.0`. We then perform `second + first` (it's important to keep this ordering for all operators since order matters for "-", "/", and "^"), and then push the result (7.0) 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 7.0 (bottom) and 5.0 (top). Finally, the last token in the list is "*", and so we pop the stack twice, multiplying 7.0 (`second`) and 5.0 (`first`) to get 35.0, 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 (notice how there are no parentheses). 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.

IMPORTANT: You <u>may not</u> rely on a parser generator tool like ANTLR to write the RPN expression parser in `RPNCalculator.java`, or you will get zero points in this portion of the assignment.

Infix Notation for InfixCalculator

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. In addition, it must implement support for three functions (defined the same way as in the RPN calculator):

- `sqrt()`
- `ln()`
- `abs()`

Each of these functions take only one argument. Under PEMDAS rules, evaluate everything inside of the parentheses of the function before evaluating the function itself. For example, the expression `4 * (3 + 8) + sqrt(2 * 3 + 4)` reduces to `4 * 11 + sqrt(10)`, then `44 + sqrt(10)`, then `44 + 3.162`, then finally `47.162`.

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. Create an ANTLR grammar named `Arith.g4` that is able to represent expressions in infix notation that is also able to implement the PEMDAS order of operations as well as the functions listed above.
2. Use ANTLR to generate a parse (derivation) tree.
3. Traverse the parse tree to evaluate the expression. There are two ways of doing this: (1) either evaluating it directly, or (2) using it to generate a postfix notation representation of the expression, and then evaluating it in the RPN calculator.

Some Examples:

```
$ echo "java 2 4 * 2 ^ 10 -" | RPNCalculator
54.0
$ echo 5 | java RPNCalculator
5.0
$ echo "8 24 + 9 -" | java RPNCalculator
23.0

$ echo "(2 * 4)^2 – 10" | java InfixCalculator
54.0
$ echo 5 | java InfixCalculator
5.0
$ echo "8 + 24 – 9" | java InfixCalculator
23.0
```

```
$ echo "-sqrt(2)" | java InfixCalculator
-1.41421356
```

Deliverable:
A *.zip archive containing the following files:

- A collection of Java source code files (including `RPNCalculator.java` and `InfixCalculator.java`, along with any Java files you needed to write)
- The ANTLR grammar file `Arith.g4`.
- A `README` file that contains your name and your partner's name.

Please do not submit any files generated by ANTLR or any artifacts from your development environment (including directory hierarchies, which some Java IDEs create); only submit the files that you created without making any directories. I will be compiling and running everyone's submission the exact same way; the naming conventions of the programs and the grammar matter. I will deduct points if the naming conventions for `RPNCalculator.java`, `InfixCalculator.java`, and `Arith.g4` deviate from the specification.

Only one partner needs to make a submission; both partners will receive the same grade for the assignment.

Grading Rubric:
Mode #1 (Postfix Notation): 40%
Mode #2 (Infix Notation): 60%

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