# PROJECT 4, Com S 228, Spring 2020

*Due at 11:59pm, Monday, April 13*

## 1 Problem Description

In this project, you are asked to implement the following three utilities for integer expressions using stacks:

- Postfix evaluation
- Infix to postfix conversion
- Infix evaluation

The two types (infix and postfix) of expressions are implemented by the classes `InfixExpression` and `PostfixExpression`, which extend the abstract class `Expression`. You are only allowed to use the stack implementation provided in the files Purestack.java and ArrayBasedStack.java.

You may add new instance variables and methods to the classes `Expression`, `InfixExpression` and `PostfixExpression`, but you may not rename or remove any existing ones, or change any of them from public to private or vice versa.

## 2 Operands and Operators

To simplify the parsing effort, all the operands in this project are either single lower case English letters (`a` - `z`) or non-negative integers. A single letter is treated as a variable whose value needs to be provided in the input. A hash map will be constructed to store all variables appearing in the expression and their values. Evaluation of this expression may produce negative (intermediate) operands, which are the values of subexpressions.

Parentheses `(` and `)` are allowed to appear in an infix expression but not in a postfix expression. They may be viewed as "special operators."

There are seven standard operators, six of which are `binary`, meaning that they act on two operands:

`+, -, *, /, %, ^`

In the above, `+` is for addition, `-` for subtraction, `*` for multiplication, `/` for integer division, `%` for modulo (the remainder operation), and `^` for exponentiation.

### 2.1 Unary Minus

There is one operator that is *unary*, meaning that it acts on only one operand: the negative sign or `unary minus`, which has the same notation `-` as subtraction in an infix expression. A unary minus appears in the following three situations only:

1. at the beginning of an expression, as in `- 3 * 5 - 7` (where the first `-` is a unary minus operator and the second `-` is a binary minus operator);

2. after another operator (either binary or unary), as in `8 - - - 10 * - 2` (where the first `-` is binary, and the remaining three are unary);
3. after a left parenthesis (i.e., first in a subexpression), as in `x / (- y + 2)`.

All other occurrences of `-` in an infix expression must be of the binary minus operator. The unary minus operator is right associative and has a higher precedence than all the binary operators. For example, the expression

```
2 * - - 5 ^ 3
```

becomes, after parenthesization,

```
2 * ((- (- 5)) ^ 3)
```

In a postfix expression, the unary minus operator is represented by a tilde: `~`. If we still used `-`, there would be no way to tell it apart from the binary minus operator in postfix notation.

### 2.2 Rank and Precedence

An operand has a rank of 1, a unary operator 0, a binary operator -1, and a parenthesis 0. To be a valid infix expression, a necessary condition is that the cumulative rank in a left-to-right scan of the expression must vary between 0 and 1, and has a final value of 1.

In an infix expression, every operator has an *input precedence* and a *stack precedence*. The precedences and ranks of operators and ranks are summarized in the table below.

| operator | input precedence | stack precedence | rank |
|---|---|---|---|
| `-` (unary) | 6 | 5 | 0 |
| `+`, `-` (binary) | 1 | 1 | -1 |
| `*`, `/`, `%` | 2 | 2 | -1 |
| `^` | 4 | 3 | -1 |
| `(` | 7 | -1 | 0 |
| `)` | 0 | 0 | 0 |

## 3 Expression Formats

In an input, operands, operators, and parentheses are separated by one or more blanks or tabs. For example, the following is a valid infix expression.

```
( x - 15 ) * 4 / 2 ^ 2
```

This expression has value `13` if `x == 28`. All integers in the input are nonnegative. An example of a valid postfix expression:

```
8 ~ 7 * y 3 + /
```

An input infix string should contain no characters other than those for the seven operators, the two parentheses, the ten digits, and the 26 English letters (in lower case). An input postfix string may also include one more character `~` for the unary minus operator but it does not contain parentheses. Your code needs to check for possible violations.

## 4 Hashing of Variables

All variables from an input expression should have their values provided by the user. Your code should store these variables and their values in a `hash map` declared as

```
private HashMap<Character, Integer> varTable;
```

This hash map will operate like a fast dictionary, allowing you to quickly retrieve the value of each variable. It should be constructed as a Java `HashMap` object in the `main()` method and passed to the postfix or infix expression. The hash map is empty if no variables occur in the input expression.

The demo code below illustrates the use of Java `HashMap`. A hash map named `hashMap` is constructed to store the values of three single-letter variables `x`, `y`, and `z`. Then, the value of `x` is retrieved.

```
class HashMapDemo {
    public static void main(String args[]) {
        HashMap<Character, Integer> hashMap = new HashMap<Character, Integer>();
        hashMap.put('x', 4);
        hashMap.put('y', 2);
        hashMap.put('z', 5);
        char c = 'x';
        if (hashMap.containsKey(c)) {
            int x = (int) hashMap.get(c);
            System.out.println(x);
        }
    }
}
```

Check out `http://docs.oracle.com/javase/6/docs/api/java/util/HashMap.html` for a summary of `HashMap` methods, though you might only need `put()` and `get()` for the project. We will study this class in more detail soon.

## 5 Correctness of Expressions and Exceptions

Your code should detect whether an input infix or postfix expression is valid. If the cumulative rank of an infix expression goes below 0 or above 1 during a scan or has a final value that is not 1, it can be immediately rejected as invalid.

Because parentheses have rank 0, passing the rank test does not guarantee that the infix expression is valid. An incorrect infix expression may also be converted into postfix, which is later detected during the evaluation.

For example, the invalid infix expression `(2 *) 3` can be converted into the postfix `2 * 3` by the algorithm described in class,[1] but evaluation of the postfix expression will then detect a missing operand of the multiplication operator `*`.

[1] See the Lecture 25 notes on Canvas.

The correctness of a postfix expression is verified during its evaluation. For instance, if a parenthesis appears in an input for a postfix expression, then that's an error. An exception is thrown in the following situations:

- An expression is found to be incorrect.
- An illegal arithmetic operation (such as division by 0 or the zeroth power of 0) is being carried out.
- A variable in the infix or postfix expression to be evaluated was not provided a value from the input.

Possible errors (and their corresponding exceptions) are itemized in the Javadoc comments in infixExpression.java and postfixExpression.java. So read those comments carefully.

# 6 Input/Output Format

You may assume (without checking it in your code) that an input file already meets the following requirements:

1. Every expression occupies a separate line.
2. Adjacent expressions may be separated by one or more blank lines.
3. Adjacent operators and/or operands must be separated by one or more blanks.
4. A line containing an infix expression starts with the letter `I`, followed by at least one blank, followed by the actual expression.
5. A line containing a postfix expression starts with the letter `P`, followed by at least one blank, followed by the actual expression.
6. If the expression contains any variables, then their names and values are listed on the lines immediately following the expression line. Each such line contains zero or more blanks, then the variable name, then `=`, then one or more blanks, then the variable's value.

Below is a sample valid input file consisting of an infix expression and a postfix expression.

```
I ( 2 + x ) -   (  33 * y )
  x      = 1
y    =    2

P  a   a ^  b    c /  +
a   =  2
b     =  8
 c =       4
```

The output must leave:

1. exact one blank separating any two operands, two non-parenthesis operators, or an operand and a non-parenthesis operator.
2. no blank between a parenthesis and an operand, or between two parentheses.

# 7 Execution Scenario

The class `InfixPostfix` repeatedly accepts infix and postfix expressions via standard input or file input. On standard input, enter the letter `I` followed by one or more blanks before an infix expression, and the letter `P` followed by one or more blanks before a postfix expression. If the expression contains some variables, the interface should then prompt the user to enter their values one by one (see Trial 2 for an example). This may be implemented in a helper method called by `main()` in `InfixPostfix.java`.

On an input infix expression, your code should output:

1. the same infix expression in the required format (see end of Section 6)
2. the equivalent postfix expression in the required format
3. (a) for standard input: the variables (if there are any) with prompts for their values
   (b) for file input: the variables (if there are any) and their values
4. the calculated value of the expression.

If some variables are not provided values, the infix and postfix expressions must be output before an `UnassignedVariableException` is thrown. On an input postfix expression, your code should do the same as above, except it skips line 1.

Below is a sample execution scenario. Note that the underlined portions represent user keystrokes.

```
Evaluation of Infix and Postfix Expressions
keys: 1 (standard input) 2 (file input) 3 (exit)
(Enter "I" before an infix expression, "P" before a postfix expression)

Trial 1: 1
Expression: I - ( 2 * i + - 3 ) * 5
Infix form: - (2 * i + - 3) * 5
Postfix form: 2 i * 3 ~ + ~ 5 *
where
i = 1
Expression value: 5


Trial 2: 1
Expression: P a 6 + b 3 ^ -
Postfix form: a 6 + b 3 ^ -
where
a = 50
b = 2
Expression value: 48


Trial 3: 2
Input from a file
Enter file name: expr.txt

Infix form: (x + y) * z
```

```
Postfix form: x y + z *
where
x = 21
y = 13
z = 5
Expression value: 170

Infix form: 8 / (1 + 3) - 6 ^ 2
Postfix form: 8 1 3 + / 6 2 ^ -
Expression value: -34

Postfix form: 2 2 +
Expression value: 4

Postfix form: 2 x ~ *
where
x = 2
Expression value: -4

Trial 4: 3
```

In trial 1, note that the uppercase `I` refers to the input being an infix expression while the lowercase `i` refers to a variable in the expression. In trial 3, the file `expr.txt` has the content below.

```
I ( x + y ) * z
x = 21
y = 13
z = 5
I 8 / ( 1 + 3 ) - 6 ^ 2
P 2 2 +
P 2 x ~ *
x = 2
```

Your code needs to print out the same text messages for user interactions.

## 8 Submission

Write your classes in the `edu.iastate.cs228.hw4` package. Turn in the zip file, not your class files. Please follow the submission guidelines posted on Canvas. You are not required to submit any JUnit test cases. Nevertheless, you are encouraged to write JUnit tests for your code. Since these tests will not be submitted, feel free to share them with other students. Include the Javadoc tag `@author` in every class source file you have made changes to. Your zip file should be named `Firstname_Lastname_HW4.zip`