SOFTWARE PROJECT - CALCULATOR

I. Title Page

Assignment Title: Expression Evaluator and Calculator GUI

Name: Jianfei Zhao

Student ID: 918126149

Class & Semester: CSC 413 - Section 01 in Summer 2018

Link to the repository: https://github.com/jzhao11/calculator

II. Introduction

a. Project Overview

This project is mainly about evaluating infix mathematical expressions and executing operators with operands to compute corresponding results. Through the application of GUI to the calculator, we could also become acquainted with the graphical user interface.

b. Technical Overview

From the technical perspective, much endeavor is made on the design of classes and the implementation of evaluating algorithm for infix expressions. More importantly, it offers us a good opportunity to practice object-oriented programming (OOP). The class hierarchy and inheritance is uncomplicated, such that it is relatively simple to design classes with high cohesion and decouple their relations. The completion of this project helps us achieve further understanding about OOP principles, as well as review the utilization of some data structure (stack and hash map).

c. Summary of Work Completed

With the procedure of how to parse infix mathematical expressions, the initial method is rearranged based on logic analysis, and confirmed as the algorithm for evaluation and computation. The data structure for evaluation (eval) function is stack, and repetitive code for processing operators and operands is extracted as an individual function to enhance program compactness.

On the ground of eval() method in class Evaluator, operand and operator classes are created and modified to provide method interfaces needed for eval(). Specifically, the abstract class Operator is treated as the superclass, and several subclasses derive from it, including left / right parenthesis and usual computing operators. Among subclasses, Unary- and BinaryOperator are also abstract, which can be extended by concrete operators (“!”, “+”, “-”, “*”, “/”, “^”), and other unary / binary operators as well. Detailed discussion for class design is available in section **VII**.

When working on the development mentioned above, the whole project is debugged and rebuilt back and forth, with the running outcomes compared with expected results. After passing multiple test cases (both simple and complex expressions), the action listener is also implemented on the calculator GUI to response the clicking events.

III. Development Environment

a. Version of Java Used

The version of Java used for development is 1.8.0_101.

b. Integrated Development Environment (IDE) Used

The IDE for Java is NetBeans 8.2 on Windows System.

IV. How to Build or Import the Project in the IDE

i). Clone the repository from the link in title page (page 1), by using the command: git clone https://github.com/jzhao11/calculator.

ii). Open the NetBeans IDE. Import the Project by clicking "File->New Project".

iii). Choose "Java Project with Existing Sources" in the "Projects" choice box and click "Next". Specify a name (e.g. csc413_hw1_jzhao11) and location for the new project and click "Next".

iv). Click "Add Folder" next to the "Source Package Folder" choice box. In the pop-up window, find the local "csc413-p1-jzhao11" repo, and go to "csc413-p1-jzhao11->calculator", and choose both "Evaluator" and "operators". Click "Open" and the file path will appear in the upper box. Then click "Next".

v). *.java files will appear in the "Included Files" box. All files are necessary, with nothing excluded. Click "Finish" to finish importing the project.

vi). To build the project, right click the imported project in the top-left "Project" box, and choose "clean and build". This can also be done by clicking the "Clean and Build Project" icon (Shift+F11 as default shortcut) on the top navigation bar.

V. How to Run the Project

i). In the top-left "Project" box, expand the categories under the imported project, and go to "Source Packages->Evaluator".

ii). To check the correctness of the test cases for this evaluator, right click "EvaluatorTester.java" and choose "Run File". This can also be done by clicking the "Run Project" (Ctrl+F11 as default shortcut) on the top navigation bar.

iii). To test the GUI of this evaluator, right click "EvaluatorUI.java" and choose "Run File". User may input any valid infix expression and click "=" to see the result, or "CE" to clear the last operand entry, or "C" to clear the text field.

VI. Assumptions in Design and Implementation

In this project, the evaluator only considers some basic binary operators and a special unary operator (factorial). Therefore, the following operators are permitted: addition ("+"), subtraction ("-"), multiplication ("*"), division ("/"), power ("^"), left parenthesis ("("), right parenthesis (")"), and factorial ("!"). Besides, the calculator deals with complete and valid infix expressions, among which any binary operator is between two operands.

The priority of usual operators is the same as normal mathematical calculations, where factorial (4) > power (3) > multiplication, division (2) > addition, subtraction (1). Although left and right parentheses are not operators for computation, their priorities are also defined. Herein, the priority of parenthesis operator is set as 0, which ensures that any usual computing operators (priority >= 1) between "(" and ")" can be pushed onto the stack.

Additionally, the evaluator only accepts non-negative integers as valid operands.

This means that all decimal numbers together with negative integers are excluded from the expressions.

VII. Implementation Discussion

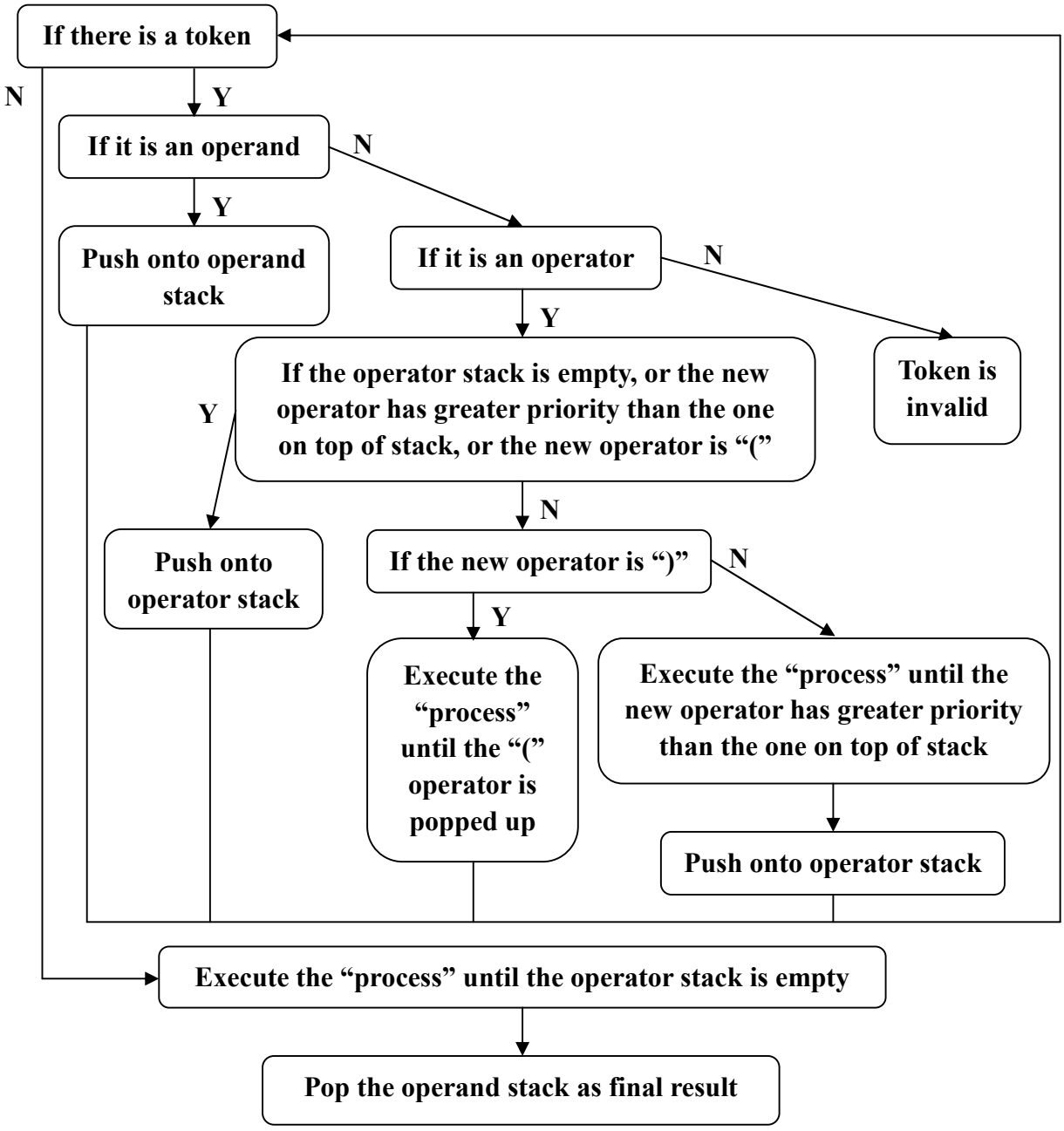

Figure 1. Flowchart of the evaluating algorithm for infix expression

The implementation starts with the evaluating (eval) function for infix expressions in class Evaluator, and then I modify related classes (such as Operand and Operator) according to what I need in the algorithm. Based on the description about the evaluation procedure listed in the .pdf file, I draw a flowchart at first and analyze the operations in different cases, during which I rewrite the logic as a new version.

The rewritten flowchart is shown as Figure 1, by using stack as the data structure. The “process”, also known as the abbreviation of “processing an operator”, refers to a series of specific operations. It can be summarized as follows: **i)** pop an operator from the stack; **ii)** pop an operand; **iii)** compute with this operand if a unary operator, otherwise (it is a binary operator) pop another operand and execute the operator with two operands; **iv)** push the result onto the operand stack. Such “process” occurs several times in the flowchart, and therefore is extracted as an individual function in class Evaluator, reducing the duplicate code lines while making the program more readable.

Compared to the original description in .pdf, the modified infix evaluating algorithm combines some cases (the operator stack is empty || the new operator has greater priority than the top element on stack || the new operator is “(”), since they all require the same following operation - pushing the new operator onto stack. In this way, the logic becomes clear, which also improves the compactness of code.

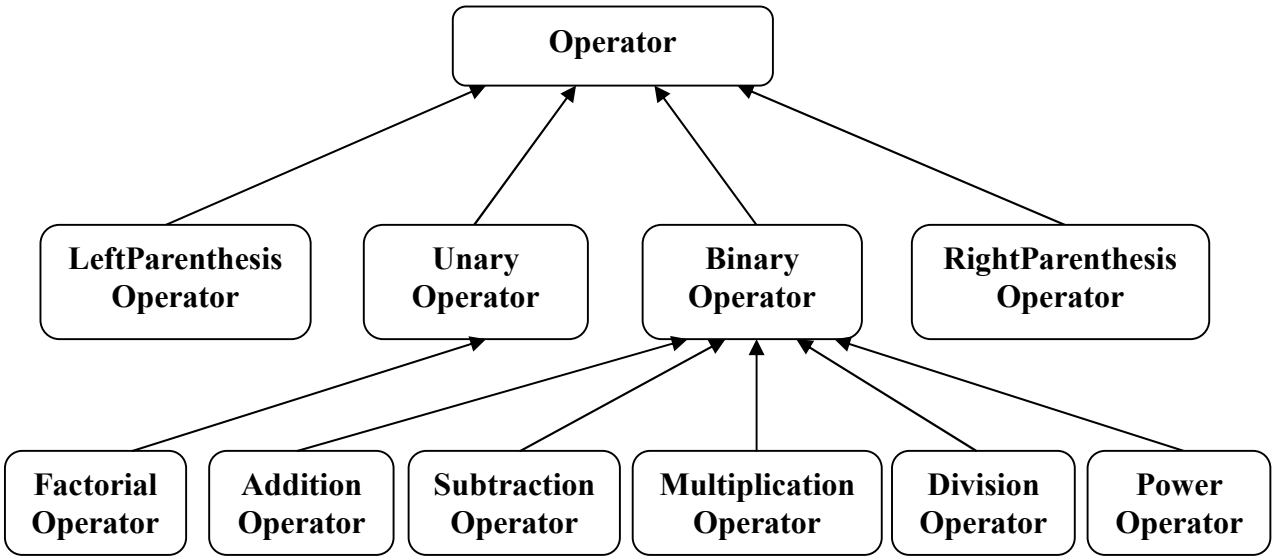

Figure 2. Class hierarchy and inheritance of operators

As drawn in Figure 2, the graph depicts the hierarchy of classes within this project, where each arrow refers to the inheritance from a subclass to a superclass. When implementing the evaluating (eval) function, all bottom-level operators together with left / right parenthesis in Figure 2 are concrete classes. The method execute() is migrated from Operator to its abstract subclasses, UnaryOperator and BinaryOperator. The reason for such design can be explained from two aspects. For one thing, LeftParenthesis and RightParenthesis do not need to worry about implementing the abstract method execute(), since they are not operators for executing computations. For another, it is easy for code extension when other operators are introduced in the future. To be more precise, the execution for unary and binary operators do not have to share the same version of execute() method since they require different number of operands (1 for unary but 2 for binary). Instead, the original execute() method in

Operator is split in two versions and moved to Unary- and BinaryOperator, respectively. Thus, if other unary operators (√, sin, cos, etc.) are included, they can directly inherit UnaryOperator and override the method execute(Operand op), while binary operators (%, &, |, etc.) can be added by extending the BinaryOperator and overriding execute(Operand op1, Operand op2).

A static hash map is initialized in class Operator to contain all the operators, with valid tokens serving as keys and concrete operator references as values. It is "static" (created only once) since the operator types are known beforehand and independent of the evaluating algorithm. The evaluation method requires operand values and operator priorities. Hence, I add a member variable in class Operand to store the integer operand value, and also define a method in Operator to retrieve the priority for each computing operator. Moreover, check() methods are defined in Operand and Operator to guarantee the validity of tokens.

VIII. Project Reflection

Having passed all the expressions in tester file and some self-defined complex cases, this project satisfies the requirements listed in the assignment document. However, there is still defect within this evaluator. For example, it is not able to handle negative or floating point numbers as operands. As a result, if a negative integer is generated as the final result in calculator GUI, the following calculation cannot be done correctly. More effort has to be put into the current

version to make it a fully developed calculator.

Despite of the drawbacks it may have, this project succeeds to evaluate infix expressions under the assumptions mentioned in section **VI**. Furthermore, it is extensible for future modification such as involving other operators. I regard this development as a comprehensive practice for OOP design, where effort has been spent in class inheritance and removal of code redundancy. Taking the project as a bone frame, I also aim to perfect my calculator after the semester by making it more robust in resolving highly complicated expressions.

IX. Project Conclusion and Results

There are two main methods in “EvaluatorTester.java” and “EvaluatorUI.java”, with the former one checking the correctness of evaluation method, and the latter one checking the GUI.

After successfully building the project, the results of “EvaluatorTester.java” will appear in the bottom-right output console of Netbeans. For each line, the left side of equality denotes the infix test cases, and the right side implies that all expressions are correctly evaluated, shown in the form of “program-calculated value : expected value”. For the GUI tests in “EvaluatorUI.java”, the calculator can also achieve right result (shown in text field), as long as the input is a valid infix expression.