School of Accountancy, Management, Computing and Information Studies
Saint Louis University

Group Programming Project 1 for Data Structures (Midterm)

Prepared by

The faculty of the IT/CS department

October 2023

Vision – Mission Statement

Saint Louis University is an excellent missionary and transformative educational institution zealous in developing human resources who are creative, competent, socially involved and imbued with the Christian spirit.

Assumption:

The Java Development Kit (JDK) and IntelliJ IDEA are both installed in your computing device.

Working folder (project):

Project name for the midterm group activities will have the naming convention: <CourseNumber>_Class Code>_midGrp_<GroupLabel> (e.g. IT212_**9300_midGroupA**). Follow the same grouping as in the previous group projects. The group label(i.e. A, B, C, ...) for each group will be given by the instructor.

You may use the same Gitlab project that your group used in previous projects. You may just create a new package within the project.

For your submission:

ONLY one representative of the group should make an upload to the Submission Bin. After ensuring that a representative of the group "Turned-In" the group's work, the other members should mark the requirement as "Turned-In".

.

Midterm Programming Project 1

Activity: Conversion of infix to postfix expression and Evaluation of a postfix expression

Create an application that allows the user to do the following:

- Input an ***infix expression*** and output a ***table of values*** as well as its equivalent converted ***postfix expression*** similar to the one shown in class(See Review notes(I) below)
  - The input expression should not have any spaces in between the characters.
  - Use the caret (^) symbol to symbolize the exponentiation operator.
  - Output an error message if the expression cannot be converted to its postfix expression equivalent.
  - The output (postfix) expression's operands and operators must be separated by space.
- Input a ***postfix expression*** and output a table of values as well as the ***result of evaluating the expression*** similar to the one shown in class(See Review notes(II) below)
  - The input (postfix) expression's operands and operators must be separated by space.
  - Use the caret (^) symbol to symbolize the exponentiation operator.
  - Output an error message if the expression cannot be evaluated.

Notes:

- For the purpose of gaining mastery to a user-defined implementation of Data Structures, your group must use the provided linked implementation of the stack data structure. You should not use existing classes (that represent a stack) from the Java API for this activity.
- Refer to the discussion on the stack data structure for this activity.
- The user interface of the application may be a menu-based console application or you may opt to create a graphical interface.

Review Notes:

I. Converting an Expression from infix form to postfix form.

An expression is in infix form if every operator in the expression is placed between its operand (examples: a+5 and x*y+2). An expression is in postfix form if every operator in the expression are placed after its operands (examples: a5+ and xy*2+). When converting an infix expression involving more than one operator to postfix, the precedence of operators must be known. It must be noted that the application of operators is based on programming rules on precedence. For instance, the multiplication operator always precedes addition. Multiplication is applied before addition in case an expression has both the addition and the multiplication operators without the parenthesis.

Assume that a function (i.e. method) with two parameters, called precedence, for determining if operator1 precedes operator2 exists and that precedence works as follows:
precedence(operator1, operator2) returns TRUE if operator1 has precedence over operator2.
precedence(operator1, operator2) returns FALSE if operator1 does not have precedence over operator2. Examples:

precedence(***, +**)	= true
precedence(**+, /**)	= false

Consider the following algorithm for deriving the postfix expression corresponding to a given infix expression. If necessary, improve the algorithm.

INPUT: A String representing an Infix Expression

OUTPUT: A String representing a Postfix expression equivalent to the Infix Expression

PROCESS:

```
postfixExpression = empty String
operatorStack = empty stack
while (not end of infixExpression) {
    symbol = next token
    if (symbol is an operand)
         concatenate symbol to  postfixExpression
    else {    // symbol is an operator
         // If the symbol is an open parenthesis, push symbol onto the Stack
         // Otherwise do the following
         while (not operatorStack.empty() &&
                precedence(operatorStack.peek(),symbol) {
             topSymbol = operatorStack.pop();
             concatenate topSymbol to  postfixExpression;
         }    // end while
         if (operatorStack.empty() ||  symbol != ‘)’ )
             operatorStack.push(symbol);
         else // pop the open parenthesis and discard it
             topSymbol = operatorStack.pop();
     }  // end else
 }      // end while
 // get all remaining operators from the stack
 while (not operatorStack.empty) {
      topSymbol = operatorStack.pop();
      concatenate topSymbol to postfixExpression
 }      // end while
 return postfixExpression
```

Illustration:
Suppose ((A–(B+C))*D)$(E+F) is applied as input to the above algorithm. Complete the following table that shows a simulation of the above algorithm by showing the value of Symbol, postfixExpression and operatorStack after each symbol of the infix expression ((A–(B+C))*D)$(E+F) is read.

For the expression, + is for addition, - is for subtraction, * is for multiplication, / is for division and $ is for exponentiation. After the open parenthesis, the level of precedence of operators from highest to lowest is as follows:

1. Exponentiation (A hypothetical operator – not an actual operator in some programming languages like Java)

2. Multiplication/Division
3. Addition/Subtraction

Consecutive operators with the same precedence are applied from left to right except for exponentiation where the order is from right to left.

Symbol	postfixExpression	operatorStack
(		(
(		((
A	A	((
-	A	((-
(	A	((-(
B	AB	((-(
+	AB	((-(+
C	ABC	((-(+
)	ABC+	((-
)		
*		
D		
)		
$		
(		
E		
+		
F		
)		

Note: For this project, let your application handle as many operators as possible. Based on your readings about operators in most programming languages, decide on the operators that your application will handle. Have a way to inform the users of your application regarding the operators that are valid as far as your application is concerned.

II. Evaluation of a Postfix Expression

Given: Postfix Expression and a Stack where operands are stored.

Each time we read an operand from the input postfix expression, we push it into the stack. When we reach an operator, its operands are the top two elements in the stack. We can then pop the two elements, perform the indicated operation on them, and push the result into the stack so that it will be available for use as an operand of the next operator.

Algorithm

```
operandStack = an empty stack
// Scan input string reading one element at a time and assign it to symbol
while (not end of postfixExpression) {
    symbol = next token
    if (symbol is an operand)
        operandStack.push(symbol)
    else {     // symbol is an operator
         operand2 = operandStack.pop();
         operand1 = operandStack.pop();
         value = result of applying symbol to operand1 and operand2
         operandStack.push(value)
    }
}
return (operandStack.pop())
```

Illustration:

Postfix String **6 2 3 + - 3 8 2 / + * 2 $ 3 +**

Symbol	operand1	operand2	Value	operandStack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
$	7	2	49	49

3	7	2	49	49,3
+	49	3	52	52

Required deliverables:

1. Archived IntelliJ IDEA project
   a. Naming convention for this project: <CourseNumber>_Class Code>_midGroup_<GroupLabel> (e.g. IT212_**9300_midGroupA**)
   b. Place all your source codes under the package name: **midlab1**
   c. Submit the archived/zipped project → e.g. IT212_**9300_midGroupA.zip**
2. Video file → mp4 format, maximum of 15 minutes
   a. Naming convention to be used: code_midGroupLabel_Act1 (e.g. IT212_**9300_midGroupA_Act1.mp4**)

3. One printed copy of accomplished Peer evaluation form that applies to both the activity 1(i.e. Conversion and Evaluation of Postfix Expression) and activity 2(i.e. Huffman Coding) and the Lecture Project. Use the same peer evaluation form template that was used during the previous group project.

Other considerations:

● The medium of instruction to be used in the video presentation must be English.

● Aside from the usual parts of the presentation(i.e. Introduction of group members, Objective of the presentation, Brief background of the problem, etc.), the video presentation should cover some important screen snapshots of your program and/or the outcome of your program during run-time.

Due date: **To Be Announced (TBA) c/o Submission Bin in the Google Classroom**