

CT2109: Assignment 2

Problem Statement

We are required to create a program that reads in a numerical infix expression from a user input and converts it to a postfix expression. Then the program should solve the postfix expression and return the correct result to the user.

The numerical infix expression that the program should accept would be in the form `<operand> <operator> <operand>`. For example: `1 + 3 = 8` and `5 - (3 - 1)`.

The postfix expression that the program will compute will be in the form `<operand> <operand> <operator>`. For example: `1 2 +` and `5 8 3 * +`.

The program is required to implement the *ArrayStack* implementation that was provided.

Analysis and Design Notes

The program will have to allow the user to input a numerical infix expression. Then the program will validate this infix expression based on several rules, mainly: expression length and valid characters. The expression must be a minimum of three characters and have a maximum of 20 characters. Also the numerical expression can only consist of single digits 0-9 and `+`, `-`, `*`, `/`, `^`, `(`, `)`. The length of the expression can first be found using the *length()* function. To validate the characters in the expression, the program can use regular expression alongside the string *matches()* function. If the user provides an incorrect expression, then the program will prompt the user to re-enter an expression.

Therefore, we can consider the following pseudocode for validating the characters in the user submitted infix expression:

```
if (length < 20 && length > 3 && str.matches("[0-9+\\-*/\\^(\\)]*$")) return true;
```

The program should use a loop that runs the length of the expression to determine whether the numbers are double digits or not.

If the numerical infix expression is correct, the program needs to convert the infix expression to postfix. For this, we look at some examples of how infix expressions get converted into postfix:

<code>A + B * C + D</code>	<code>A B C * + D +</code>
<code>(A + B) * (C + D)</code>	<code>A B + C D + *</code>
<code>A * B + C * D</code>	<code>A B * C D * +</code>
<code>A + B + C + D</code>	<code>A B + C + D +</code>

From this, we can deduce that the program needs an algorithmic approach to convert infix to postfix expression. For this operation, a function will accept a validated infix expression, scan it left to right and create a character array with each character from the infix expression in the array. Next, the

program will use a loop and convert the infix expression to postfix. The algorithm should function as follows:

- If the current character is an operand, append it to an output String.
- If the current character is a left parenthesis, push it onto the Stack.
- If the current character is a right parenthesis, pop the Stack until the corresponding left parenthesis is removed. At this point, the algorithm will also append each operator to the end of the output String.
- If the current character is an operator, push it on the Stack. To ensure that the operators follow the correct precedence, the algorithm will remove any operators already on the Stack that have higher or equal precedence and append them to the output String.

The program needs a way to tell the precedence of the various operators. Therefore, list the precedence of each operator and create pseudocode that allows the program to access this information:

(0
)	0
+	1
-	1
*	2
/	2
^	3

```
getPrecedence (char input) {  
    if (input == '(' || input == ')') {  
        return 0  
    } ...  
}
```

Once the program has created the required postfix expression, we need to compute it. The basis for the computation of the postfix expression can be extended from the algorithm that is used for converting infix into postfix. Using this algorithm, the program can scan the postfix expression. An *ArrayStack* will hold the expression's operands. For every element in the expression, we determine if the element is a number or an operator. If the element is an operand, we can push it onto the stack, else if the element is an operator, we pop two operands for the operator from the stack and evaluate it, remembering to include the precedence for each operator. At the end, the number on the stack should be the answer.

Testing

A. Correct infix expressions

Inputted Infix Expression	Converted Postfix Expression	Expected Postfix Expression	Result	Expected Result
$4+8*3$	$483*+$	$4\ 8\ 3\ *\ +$	28.0	28
$(4+8)*3$	$48+3*$	$4\ 8\ +\ 3\ *$	36.0	36
$6/2*(1+2)$	$62/12+*$	$6\ 2\ /\ 1\ 2\ +\ *$	9.0	9
$5*(6^2-2)$	562^2-*	$5\ 6\ 2\ ^\ 2\ -\ *$	170.0	170
$6+9+(4*2+4^2)$	$69+42*42^++$	$6\ 9\ +\ 4\ 2\ *\ 4\ 2\ ^\ +\ +$	39.0	39
$5-3*(2^3-5+7*3)$	$5323^5-73*+*-$	$5\ 3\ 2\ 3\ ^\ 5\ -\ 7\ 3\ *\ +\ *-\$	-67.0	-67
$((3+2)^2+3)-9+3^2$	$32+2^3+9-32^+$	$3\ 2\ +\ 2\ ^\ 3\ +\ 9\ -\ 3\ 2\ ^\ +$	28.0	28
$(8/3)^2+((3+7)*5^2)$	$83/2^37+52^*+*$	$8\ 3\ /\ 2\ ^\ 3\ 7\ +\ 5\ 2\ ^\ *+\$	257.111	257.113
$7+(3-5*(2+8))/6$	$73528+*-6/+$	$7\ 3\ 5\ 2\ 8\ +\ *-\ 6\ /\ +$	-0.833	-0.833
$(2^2+5-7)*7$	22^5+7-7*	$2\ 2\ ^\ 5\ +\ 7\ -\ 7\ *$	14.0	14
$6+(8*4)/2$	$684*2/+$	$6\ 8\ 4\ *\ 2\ /\ +$	22.0	22

B. Incorrect Infix Expressions


Inputting these expressions into the program, results in the error pane displaying.

Inputted Infix Expression	Cause of Error
$-3+4$	Negative number
$((3))$	Unbalanced parentheses
$(3+-3)$	Negative number
$5*(6^2--2)$	Negative number
$5-3*(2^3-5+7*3$	Unbalanced parentheses
$(8/3)^-2+((3+7)*5^2)$	Negative number
$(3)(3)$	Invalid parentheses placement
$(8/3)^2+((3+4)(3+7)*5^2)$	Invalid parentheses placement
$--3+4$	Negative number
$32+4$	Double digit number
$(8/3)^-2+((32+7)*5^2)$	Double digit number


C. Program Output Screenshots

Infix Expression: $5-3*(2^3-5+7*3)$

Input

 Please enter a numerical infix expression between 3 and 20 characters:


Message

 The result of the expression is:
Infix: $5-3*(2^3-5+7*3)$
Postfix: 5323^5-73^{*+}
Result: -67.0


```
2.0 ^ 3.0 = 8.0
8.0 - 5.0 = 3.0
7.0 * 3.0 = 21.0
3.0 + 21.0 = 24.0
3.0 * 24.0 = 72.0
5.0 - 72.0 = -67.0
```

Infix Expression: $5*(6^2-2)$

Input

 Please enter a numerical infix expression between 3 and 20 characters:

Message


 The result of the expression is:
Infix: $5*(6^2-2)$
Postfix: 562^2-^*
Result: 170.0

```
6.0 ^ 2.0 = 36.0
36.0 - 2.0 = 34.0
5.0 * 34.0 = 170.0
```

Infix Expression: $(8/3)^{-2} + ((3+7) \cdot 5^2)$ (negative number)

Input

×


 Please enter a numerical infix expression between 3 and 20 characters:

OK

Cancel

Message

×

 Only the following characters are valid: +, -, *, /, ^, (,) and numbers 0-9

OK

Source Code

```
import java.util.regex.Pattern;
import javax.swing.JFrame;
import javax.swing.JOptionPane;

public class ExpressionSolver {
    public static void main(String[] args) {
        String expression = getExpression();
        String postfix = infixToPostfix(expression);
        double result = evaluatePostfix(postfix);

        JOptionPane.showMessageDialog(null,
            "The result of the expression is:\n" +
            "Infix: " + expression + "\n" +
            "Postfix: " + postfix + "\n" +
            "Result: " + result
        );
    }

    public static String getExpression() {
        String str = JOptionPane.showInputDialog(null, "Please enter a numerical infix expression between 3 and 20 characters:");
        boolean valid = validateInfixExpression(str);

        // If validation fails, show an error pane, and run function again
        if (!valid) {
            JFrame error = new JFrame();
            JOptionPane.showMessageDialog(error, "Only the following characters are valid: +, -, *, /, ^, (, ) and numbers 0-9");
            return getExpression();
        }

        return str;
    }

    public static boolean validateInfixExpression(String expression) {
        // Use ReGex to validate for correct characters.
        boolean chars = Pattern.compile("[\\d+*\\/\\^\\(\\)\\-]*$").matcher(expression).matches();

        // Validate length.
        boolean length = expression.length() >= 3 && expression.length() <= 20;

        // Validate for balanced brackets by counting and comparing the number of left and right parentheses.
        int left = 0, right = 0;
        for (int i = 0; i < expression.length(); i++) {
            char c = expression.charAt(i);
            if (c == '(') left++;
            else if (c == ')') right++;
        }

        // Validate for single digit numbers.
        char initial = expression.charAt(0);

        // Check if first character is a minus sign.
        if (initial == '-') return false;
    }
}
```

```

for (int i = 1; i < expression.length(); i++) {
    char c = expression.charAt(i);

    // If the previous and current chars are digits, then a double-digit number is present.
    if (Character.isDigit(initial) && Character.isDigit(c)) return false;

    // Fail validation based on rules that compare the previous and current character:
    // 1. Digit followed by left parenthesis
    // 2. Left parenthesis followed by right parenthesis
    // 3. Two minus signs together
    // 4. Operator followed by a minus sign (use precedence() function to check if
operator)
    // This offers basic validation for minus numbers and parentheses
    if (Character.isDigit(initial) && c == '(' ||
        initial == ')' && c == '(' ||
        initial == '-' && c == '-' ||
        precedence(initial) != 0 && c == '-') return false;

    initial = c;
}

if (chars && length && left == right) return true;

return false;
}

public static String infixToPostfix(String infix) {
    StringBuffer postfix = new StringBuffer(infix.length());
    Stack stack = new ArrayStack();

    for (int i = 0; i < infix.length(); ++i) {
        char c = infix.charAt(i);

        // If current character is operand then append to postfix expression.
        if (Character.isLetterOrDigit(c)) {
            postfix.append(String.valueOf(c));
        }

        // If current character is left parenthesis then push onto stack.
        else if (c == '(') {
            stack.push((Object) c);
        }

        // If current character is right parenthesis then while the stack is not empty
        // and the top of the stack is not a left parenthesis, then append the value of the
popped stack.
        else if (c == ')') {
            while (!stack.isEmpty() && (char) stack.top() != '(') {
                postfix.append(String.valueOf(stack.pop()));
            }
            stack.pop();
        }

        // If current character is an operator then while the stack is not empty
        // and the precedence of the operator is less than or equal to precedence of the
operator on the

```

stack. // top of the stack append the value of the popped stack. Then push the result to the

```
else {
    while (!stack.isEmpty() && precedence(c) <= precedence((char) stack.top())) {
        postfix.append(String.valueOf(stack.pop()));
    }
    stack.push((Object) c);
}
}
```

```
// Create the postfix string by popping the stack, while it is not empty.
while (!stack.isEmpty()) {
    postfix.append(String.valueOf(stack.pop()));
}
```

```
return postfix.toString();
}
```

```
public static int precedence(char operator){
    switch (operator){
        case '^':
            return 3;
        case '/':
        case '*':
            return 2;
        case '+':
        case '-':
            return 1;
        default:
            return 0;
    }
}
```

```
public static double evaluatePostfix(String postfix) {
    double x, y, result = 0;
    Stack stack = new ArrayStack();

    for (int i = 0; i < postfix.length(); ++i) {
        char c = postfix.charAt(i);

        if (Character.isLetterOrDigit(c)) {
            // If the current character is a number, then push it on the stack.
            stack.push((Object) c);
        } else {
            // Get values of operands from the stack.
            y = Double.parseDouble(String.valueOf(stack.pop()));
            if (stack.isEmpty()) break;
            x = Double.parseDouble(String.valueOf(stack.pop()));

            // Calculate based on current operator.
            switch (c) {
                case '*':
                    result = x * y;
                    System.out.printf("%.1f * %.1f = %.1f\n", x, y, result);
                    stack.push(result);
                    break;
                case '+':
```



```

        result = x + y;
        System.out.printf("%.1f + %.1f = %.1f\n", x, y, result);
        stack.push(result);
        break;
    case '-':
        result = x - y;
        System.out.printf("%.1f - %.1f = %.1f\n", x, y, result);
        stack.push(result);
        break;
    case '/':
        result = x / y;
        System.out.printf("%.1f / %.1f = %.1f\n", x, y, result);
        stack.push(result);
        break;
    case '^':
        result = Math.pow(x, y);
        System.out.printf("%.1f ^ %.1f = %.1f\n", x, y, result);
        stack.push(result);
        break;
    default:
        break;
    }
}

return result;
}
}

```