Andrew Hayes
21321503
a.hayes18@nuigalway.ie

CT2109 Assignment 2

2023-02-05

Solving Expressions in Postfix Notation Using Stacks

1 Problem Statement

The problem here is a reasonably basic one. We want to create a program that allows a user to enter an arithmetic expression in conventional algebraic form (infix notation), calculates the answer to the expression using a Stack, and displays the results to the user. The precedence rules outlined in the assignment specification are the well-known BIMDAS (sometimes PEMDAS), meaning that Brackets have the highest precedence, followed by Indices, followed by Multiplication & Division, followed by Addition & Subtraction.

To use a Stack to perform the calculations, the expression must first be converted from infix notation to postfix notation. This allows a much simpler way of evaluating the expressions. If infix was used on the stack, with one symbol per "slot" or index in the stack, the symbol at the top of the stack would be an operand. Our program would need to pop the stack several times before it could determine what it was actually supposed to do with said operand. If postfix notation was used, our program knows exactly what to do because the first symbol on the stack will be an operator. When an operator is encountered, the program will know that the operator requires two operands and pop them from the stack, avoiding any amount of guesswork or confusion that infix notation would've caused.

2 Analysis & Design Notes

Firstly, we want to scan in the expression from the user using a `Scanner` object and a `String`. There are some rules about valid & invalid expressions, and if the expression is invalid, we want to prompt the user to re-enter their expression. The criteria for validity are as follows: the expression must be between 3 & 20 characters, and it must only contain single digits 1-9 and the symbols `^`, `*`, `/`, `+`, `-`, `(`, & `)`. In a do-while loop, we'll prompt the user to enter an expression and scan it in. The expression will first be checked to ensure that it is of the appropriate length. If not, the loop will repeat. If the expression is of the appropriate length, it will then be checked using a regular expression to see if it contains any characters that are not the allowed characters. If so, the loop will repeat. If the expression does not contain any illegal characters, it will finally be checked using a regular expression to see if it contains numbers that have two or more digits (essentially just checking if a digit is ever followed by another digit). If so, the loop will repeat. Otherwise, the loop will end, and the program will proceed.

If the expression is valid, it will then be converted to postfix notation using the algorithm outlined in the assignment specification. The user's input String will be passed to a method that will convert it to a character array and loop over it, implementing the algorithm as follows:

1. If the character is a "`(`", it will be pushed to the stack.
2. Else, if the character is a "`)`", a loop will be entered wherein the stack is popped and the results appended to the output String until a "`(`" is encountered and the stack is not empty (to prevent errors). If a "`(`" is encountered, it will be popped from the stack and discarded.
3. Else, if the character is a digit (operand), it will be appended to the output String.
4. Else, if the stack is empty, or contains a "`(`", or the precedence of the scanned operator is greater than the precedence of the operator on the stack, the scanned operator will be pushed to the stack. (We know at this point that it is an operator, as it's not a digit). It's important that the `stack.isEmpty()` condition comes first, as if true, it will prevent the rest of the condition being evaluated (in Java, logical OR is such that if the first part of the condition is true, Java won't bother to evaluate the rest, as the whole condition must be true if part of it is true). This is important because if the subsequent `stack.top()` operations were called on an empty stack, an error would occur.
5. Else, all the operators in the stack which are greater than or equal to the scanned operator will be popped from the stack using a while loop (again, ensuring that the stack isn't empty first), and appended to the output String. If a "`(`" or "`)`" is encountered while poppung, it will be popped from the stack and the loop will break. After that, the scanned operator will be pushed to the stack.

6. Finally, any remaing content in the stack will be popped and appended to the output String, which will subsequently be returned.

A utility to determine the precedence of an operator will be required for the above method, so one will be implemented as a method with the signature `public static int precedence(char c)`. This method will return an integer value; The higher the value of the returned integer, the higher the precedence of the operator that was passed to the method. This will be implemented with a simple `switch (c)` statement, with `default` returning `-1` to indicate no precedence (invalid operator).

Once the postfix expression has been returned from the converter method in String form, it will then be passed to a method which will convert it to a character array and will evaluate it using the algorithm outlined in the assignment specification, which it will implement as follows, by iterating over each character in the postfix expression:

1. If the character is a digit (operand), it will be pushed to the stack.
2. Else, as it must be an operator, two operands will be popped from the stack to be evaluated using that operator. We will want these operands to be `doubles`, as there may be division involved, and it's more simple if only one numeric type is used. Operand 2 will be the one above operand 1 on the stack, as the expression is in postfix, but because Java works in infix, we will have to treat it as an infix expression. There is some difficulty involved, as the ArrayStack contains type `Object`. These `Objects` will be of two types: `Character` & `Double` (autoboxed from `char` & `double`). Since we are assigning these elements from the stack to variables of type `double`, we will need to cast them to type `double` first.

   If the element is an `instanceof Character`, it must first be converted to a `char` (unboxed) and then the value of the *character* "0" subtracted from it. This will give the numeric value of the character. Else, the element must be of type `Double` which can easily be cast to `double` to unbox it.

   Finally, a `switch (c)` statement will be used on the operator to determine how to evaluate it. There will be a case for each operator, and in it, the result of operand 1 combined with operand 2 using that operator will be pushed to the stack.
3. When each character in the character array has been looped over, the element at the top of the stack is the answer. This will be popped, cast to type `double`, and returned to be printed out & displayed to the user.

3 Code

```
1  import java.util.*;
2  import java.util.regex.*;
3
4  public class StackCalculator {
5      public static void main(String[] args) {
6          Scanner sc = new Scanner(System.in);    // creating a new scanner to read in expressions
       from the user
7          String expr;                            // creating a String to hold the expression read in
       .
8          boolean invalidInput;                   // boolean to tell whether the user's input was
       invalid
9
10         // will only loop if invalidInput is set to true
11         do {
12             // default false, meaning we assume valid input
13             invalidInput = false;
14
15             // prompting the user to enter expression & scanning it in
16             System.out.println("Enter an infix numerical expression between 3 & 20 characters:");
17             expr = sc.nextLine();
18
19             // regex that will be used to match expressions that contain illegal characters
20             Pattern illegalchars = Pattern.compile("(?=[^\\^\\*\\/\\+\\-\\(\\)])(?=[^0-9])"); //
       this is confusing-looking because in java, one has to escape the backslashes for one's regex
       escape sequences
21             Matcher illegalcharsMatcher = illegalchars.matcher(expr);
22
23             // regex that will be used to match numbers that are double-digit or more
24             Pattern doubledigit = Pattern.compile("[0-9][0-9]");    // just checking if a digit is
       ever followed by another digit
25             Matcher doubledigitMatcher = doubledigit.matcher(expr);
26
27             // checking that the input length is correct
28             if (expr.length() > 20 || expr.length() < 3) {
29                 System.out.println("Invalid input. Please ensure that the length of the input is
       between 3 and 20 characters");
```

```
                invalidInput = true;
            }
            // checking for invalid characters using a regular expression which matches strings that contain characters that are neither operands or digits
            else if (illegalcharsMatcher.find()) {
                System.out.println("Invalid input. Please use only the operators '^, *, /, +, -, (, )' and the operand digits 0-9");
                invalidInput = true;
            }
            // checking for numbers that are not single-digit
            else if (doubledigitMatcher.find()) {
                System.out.println("Invalid input. Please only use single-digit numbers.");
                invalidInput = true;
            }
        } while (invalidInput);

        // converting the expression to postfix
        String postexpr = in2post(expr);

        // evaluating the postfix expression & printing the result
        System.out.println(expr + " = " + evalpost(postexpr));
    }

    // method to evaluate postfix expressions
    public static double evalpost(String str) {
        ArrayStack stack = new ArrayStack();    // arraystack to be used during calculations
        char[] chars = str.toCharArray();       // turning the str expression into a character array to make iterating over it easy

        // iterating over the postfix expression
        for (char c : chars) {
            // if the element is an operand, pushing it to the stack
            if (Character.isDigit(c)) {
                stack.push(c);
            }
            // if the character is not a digit, then it must be an operator
            // popping two operands from the stack for the operator & evaluating them, then pushing the result to the stack
            else {
                // converting the operands to doubles for simplicity's sake if division is encountered
                // using an if statement to detect if the top is a Character or a Double.
                // if it's a Character, casting to char and subtracting the value of the character '0' to get the character's numeric value
                // else, casting it to double
                double operand2 = stack.top() instanceof Character ? (double) ((char) stack.pop() - '0') : (double) stack.pop();    // what would normally be operand 2 in infix will be the first on the stack
                double operand1 = stack.top() instanceof Character ? (double) ((char) stack.pop() - '0') : (double) stack.pop();

                // switch statement on the operator to see which operator it is
                // evaluating the expression and pushing the result to the stack
                switch (c) {
                    // exponentiation
                    case '^':
                        stack.push(Math.pow(operand1, operand2));
                        break;

                    // multipication
                    case '*':
                        stack.push(operand1 * operand2);
                        break;

                    // division
                    case '/':
                        stack.push(operand1 / operand2);
                        break;

                    // addition
                    case '+':
                        stack.push(operand1 + operand2);
                        break;

                    // subtraction
                    case '-':
                        stack.push(operand1 - operand2);
                        break;

                    // printing an error and exiting with code 1 if an unknown operator is somehow encountered
                    default:
                        System.out.println("The postfix expression contained an unrecognised operator! Exiting...");
                        System.exit(1);
                }
            }
        }

        // returning the final answer - the number on the stack
        return (double) stack.pop();
    }
```

```
111
112     // method to convert infix to postfix
113     public static String in2post(String str) {
114         ArrayStack stack = new ArrayStack();
115         char[] chars = str.toCharArray();   // converting str to a character array to make it
        easier to iterate over
116         String output = "";                 // output string to be returned
117
118         // looping through each character in the array
119         for (char c : chars) {
120             // if the scanned character is a '(', pushing it to the stack
121             if (c == '(') {
122                 stack.push(c);
123             }
124             // if the scanned character is a ')', popping the stack & appending to the output until
         a '(' is encountered
125             else if (c == ')') {
126                 while (!stack.isEmpty()) {
127                     // if a ( is encountered, popping it & breaking
128                     if (stack.top().equals('(')) {
129                         stack.pop();
130                         break;
131                     }
132                     // otherwise, popping the stack & appending to the output
133                     else {
134                         output += stack.pop();
135                     }
136                 }
137             }
138             // appending the character to the output string if it is an operand (digit)
139             else if (Character.isDigit(c)) {
140                 output += c;
141             }
142             // if the stack is empty or contains '(' or the precedence of the scanned operator is
        greater than the precedence of the operator in the stack
143             // important that stack.isEmpty() comes first - the rest of the if condition will not
        be evaluated if this is true as we are using OR
144             // this prevents any NullPointerExceptions from being thrown if we try to access the
        top of an empty stack
145             else if (stack.isEmpty() || stack.top().equals('(') || precedence(c) > precedence((char
        ) stack.top())) {
146                 // pushing the scanned operator to the stack
147                 stack.push(c);
148             }
149             else {
150                 // popping all the operators from the stack which are >= to in precedence to that
        of the scanned operator & appending them to the output string
151                 while (!stack.isEmpty() && precedence((char) stack.top()) >= precedence(c)) {
152                     // if parenthesis is encountered, popping it, stopping, and pushing the scanned
         operator
153                     if (stack.top().equals('(') || stack.top().equals(')')) {
154                         stack.pop();
155                         break;
156                     }
157                     // otherwise, popping the stack and appending to output
158                     else {
159                         output += stack.pop();
160                     }
161                 }
162
163                 // after that, pushing the scanned operator to the stack
164                 stack.push(c);
165             }
166         }
167
168         // popping and appending to output any remaining content from the stack
169         while (!stack.isEmpty()) {
170             output += stack.pop();
171         }
172
173         // returning the generated postfix expression
174         return output;
175     }
176
177     // method to get the precedence of each operator - the higher the returnval, the higher the
        precedence. -1 indicates no precedence (invalid char)
178     public static int precedence(char c) {
179         switch (c) {
180             // exponentiation
181             case '^':
182                 return 2;
183
184             // multiplication
185             case '*':
186                 return 1;
187
188             // division
189             case '/':
190                 return 1;
191
192             // addition
193             case '+':
```

```
                return 0;

            // subtraction
            case '-':
                return 0;

            // default - invalid operator
            default:
                return -1;
        }
    }
}
```

StackCalculator.java

4 Testing

The first series of tests that we'll want to perform are testing that the program rejects invalid inputs. The testing that it accepts valid inputs will come later. We expect that the program will prompt us to re-enter our expression in the following circumstances:

- If the expression is not between 3 & 20 characters in length.
- If the expression entered contains a character other than the digits 0-9 and the symbols `^`, `*`, `/`, `+`, `-`, `(`, & `)`.
- If the expression contains any double-digit numbers.

The screenshots below show that the expected output for the scenarios outlined above match the real output:

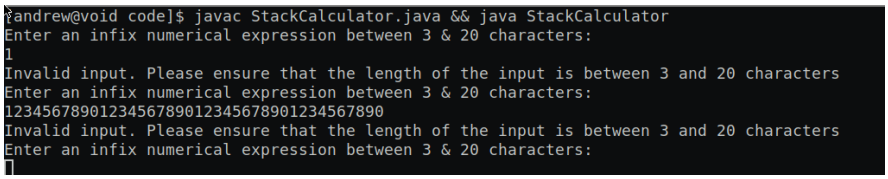

Figure 1: Testing Expressions of Illegal Length

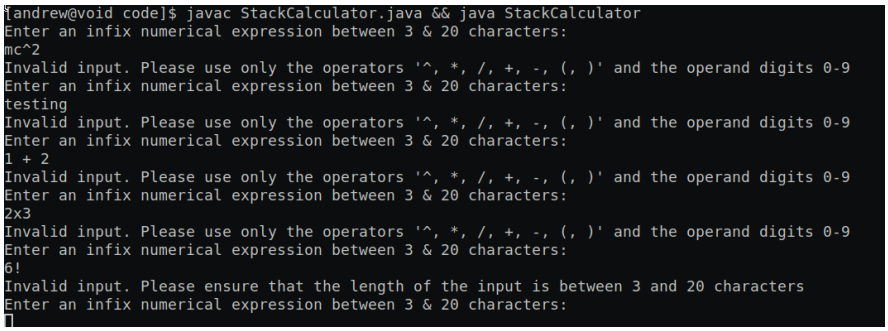

Figure 2: Testing Expressions which Contain Illegal Characters

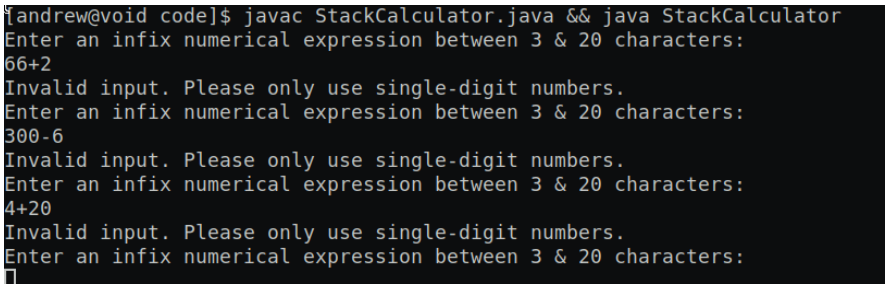

Figure 3: Testing Expressions which Contain Double-Digit Numbers

Of course, the next thing that we must ensure is that the program accepts valid inputs. However, for the sake of convenience & concision, these tests can be bundled with tests ensuring that the calculations work properly.

Assuming that the program passes the tests which check if the program calculates expressions correctly, then we know that the program must also be accepting valid inputs.

The next series of tests that we'll want to perform are checking that the program calculates the correct answer to the expressions that are entered. We'll want to test each operator both individually and in concert with other operators. We expect that the program obeys the standard BIMDAS rules, and that the results match the results you would get from any other calculator.

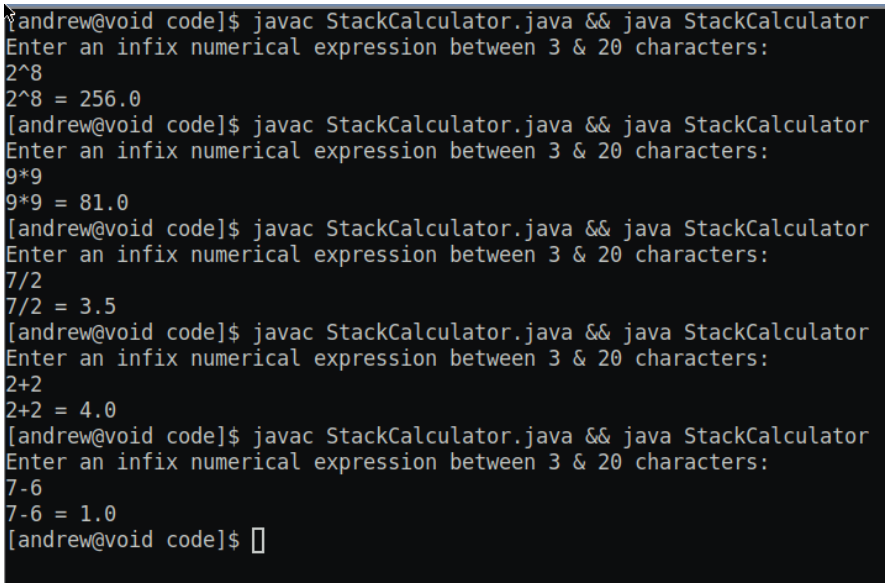

Figure 4: Testing Each Operator Individually

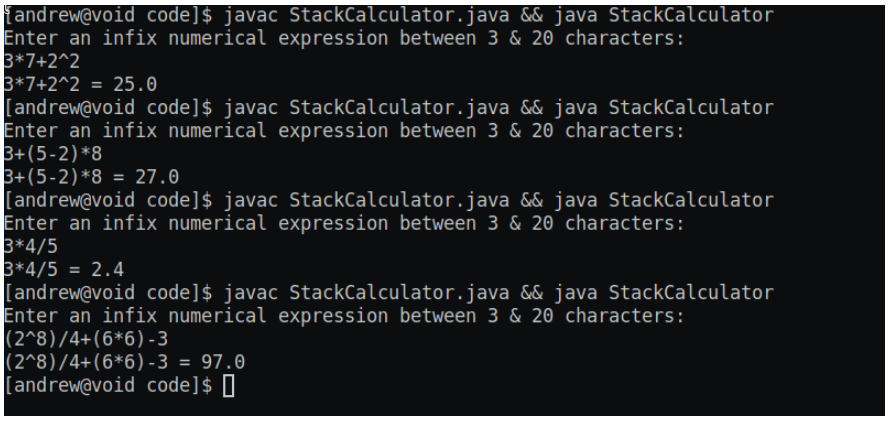

Figure 5: Testing Combinations of Operators

These screenshots demonstrate that the program behaved as expected in each potential situation. The program rejects invalid input, accepts valid input, and calculates the correct answer for each valid input, regardless of which operators or which numbers are combined together.