Infix Expression Parser

Group Project 2 for CPT-287

Created by:

Isaac Hodge

Angel Hristov

Adrian Ronchetto

The second project our group has created is the Infix Expression Parser. The basic premise for this project was to take in infix expressions and evaluate them. The infix expression would be taken in as a string and be read in to one of two stacks: operands or operators. If the current character in the expression was a number, it would be pushed onto the operand stack. If it was an operator, it would be pushed onto the operator stack. From there, the expression could be evaluated piece by piece.

Our project required no new class to be created, only needing to utilize the stack data structure. The project required two stacks to be created, those being the aforementioned operator stack and operand stack. Whenever the character in the expression was an integer, it would be pushed onto the operand stack and wait to be evaluated with another number and an operator. If the character was an open parenthesis, it would be pushed onto the operand stack and wait for the closing parenthesis. When the closing parenthesis was added, it would evaluate all numbers and operators inside the two parentheses in order of precedence. When any other operator was added, the operator would be added to the stack and checked for precedence against the previous operator if there was one. If the previous operator had precedence, it would be evaluated immediately. Otherwise, the expression would continue to be read in until the end. Once the expression had been read in, it would be evaluated entirely and outputted to the console.

Each team member took part in the coding for the project. Adrian Ronchetto created the main code for the precedence of the operators and the code to parse in the infix expression to the stacks. Angel Hristov created the code used to decide what operations needed to be used and the code to evaluate the expressions. Isaac Hodge created the code to read in the input file, compiled the code and made sure it worked together, assisted in the reading in of the comparison and logical operators, assisted in the precedence code, and wrote the project report.

The system could be improved in a number of ways. First, the code could most likely be condensed in order to have better organization. One possible idea would be to assign precedence based on their group rather than checking every individual possibility. This would reduce the number of lines required and give the code better readability. Another change would be to remove the array that is currently in the evaluate method, as it is most likely unnecessary. One final change is that the number of parses could probably be reduced. For example, the operators are read in through a string, put into a character array, put into the operator stack as another string, and then the evaluation occurs. This could most likely be reduced again by removing the character array and replacing the respective code.

Test Case 1

Input:

1+2\*3

2+2^2\*3

1==2

1+3 > 2

(4>=4) && 0

(1+2)\*3

2%2+2^2-5\*(3^2)

Output:

7

14

0

1

0

9

-41

Operations Used: (Based on the operators in the input)

Test Case 2

Input:

24 -18\*2

57 || 0

27/2^3

((2+2)\*2)^2

6 > 5 && 4 > 5

3 <= 3

17/0

Expected Output:

-12

1

3

64

0

1

Exception in thread "main" java.lang.UnsupportedOperationException: Cannot divide by zero

Operations Used: (Based on the operators in the input)

UML Chart

