Table of Contents

1 Introduction

> [!question] "What is the <Infix-to-Postfix-Calculator> (MC01)?"

This is a simple calculator that parses in a string with 255 length and outputs it to the console.

1.1 Background

This project is created by Group 15 for their MC01 Project in their CCDSALG Class S13. This is under De La Salle University - Manila.

1.2 Authors

These are university students of De La Salle University - Manila, Philippines. They are:

1. Bunyi, Christian Joseph C. @cjbnyi
2. Campo, Roan Cedric V. @ImaginaryLogs
3. Chan, Enzo Rafael S. @nomu-chan

2 Code Documentation and Description

Stacks and Queues

The stacks are implemented using doubly-linked lists, whereas the queues are implemented using the singly-linked lists.

For the stacks, they are declared by setting one doubly-linked list node pointer, being top, which pertains the top of the stack.

For the queues, they are declared by setting two singly-linked list node pointers, being the pHead and pTail, which pertains to the head of the queue and tail of the queue, respectively. In our implementation, the tail is defined as the last element inserted rather than the memory allocation where the next element will be inserted.

Algorithm Implementation

For our algorithms, we followed the following operator precedence:

1. Paretheses ( ), left to right
2. Not Operator !, right to left
3. Exponent ^, right to left
4. Multiplication, Division and Modulo * / %, left to right
5. Addition and Subtraction + -, left to right
6. Comparison Operators > >= < <=, left to right
7. Comparison Operators == !=, left to right
8. And Operator &&, left to right
9. Or Operator ||, left to right

2.1 Infix to postfix

Our first major algorithm is the infix to postfix algorithm, which takes in the ff. inputs:

1. The infix string to be converted to postfix.
2. A pointer to a queue which will store the postfix tokens
3. An operation table containing necessary information about each operator

High-level overview

1. Create a stack to keep track of operators.
2. Iterate through each token of the infix string.
    1. Check if the current token is an operand or an operator.
    2. If it's a number/operand, enqueue it to the provided queue.
    3. If it's an operator, check which operator it is.
        1. If it's an open parenthesis, push it to the operator stack.
        2. Else if it's a close parenthesis, pop operations from the operator stack and enqueue them to the postfix queue until an open parenthesis is encountered.
        3. Else, pop operations from the operator stack while the operator at the top of stack has a higher precedence than the current token.
3. Finally, pop operations from the operator stack and enqueue them to the postfix queue while there are any remaining operators in the operator stack.

By the end of the infix to postfix algorithm, the provided pointer to queue must already point to a queue that contains the tokens of the converted postfix expression in order. This queue will eventually be passed in to the evaluate postfix function, which is essentially an implementation of the Shunting yard algorithm.

2.2 Shunting yard algorithm

Our second major algorithm is the shunting yard algorithm, which takes in the ff. inputs:

1. A pointer to a queue of postfix tokens
2. A pointer to a string holding the evaluated answer
3. An operation table containing necessary information about each operator

High-level overview

1. Create a stack to maintain operands.
2. Iterate through each token in the queue of postfix tokens.
    1. Check if the current token is an operand or an operator.
    2. If the current token is an operand, push it to the stack maintaining operands.
    3. Else if the current token is an operator, pop the top two operands and evaluate them accordingly with the operator. Push the result to the stack of operands.
3. Pop the top of the stack of operands. This is the result of the postfix expression.

By the end of the shunting yard algorithm, the queue of postfix tokens will have been have evaluated as a postfix expression. The corresponding answer must then be stored in the string.

Do note that the two provided algorithm overviews are very high-level. In the program implementation, there are many things happening under the hood, such as handling potential errors, utilizing helper functions, etc. We only expounded on the major algorithms that fulfill the main goal of the program, but they are in fact composed of many more functions with their own algorithms.

Code Limitations and Errors

We have addressed some of the errors that can be made when inputting a string of operators and operands. In fact, we have also addressed any other incorrect string input that can be placed by the user.

For the errors when inputting a string of operators and operands, we have addressed the following errors:

1. Mismatched Parentheses, which occurs when there are either incorrectly placed parentheses or if there is an unequal number of right and left parentheses.

2. Malformed expressions, cause by incorrectly placed operators.

3. Division by 0.

4. 0 ^ 0 expression.

As for the general case of an incorrect string input, our code also detects incorrect characters, empty inputs, unrecognizable operators, and character overflow (if the input has more than 255 characters).

Lastly, though this is not a limitation of our code itself, but a limitation of C datatypes, the code cannot handle exceedingly large integer/operand inputs.