BTEC: Mechatronics

2023

Computer System Principles and Practice

Development of a computer program to solve an engineering related problem onscreen

Written by:

Mohammed Ghalib Al-Hosni | 1615 | 7A

Report brief

This report focuses on solving a problem seen throughout software engineering related to mathematical expression efficiency. It will go through the process of developing an infix to postfix conversion algorithm using the C# programming language. The objective is to create a solution that efficiently converts infix expressions to postfix expressions while considering operator precedence, associativity, and parentheses. The report will also delve into what exactly is the difference between the two notations and why might postfix notation be more efficient when working with software.

The report encompasses both the technical implementation and theoretical understanding of the problem, as it will discuss be requirements to develop the conversion algorithm using appropriate data structures and handle different input expressions to generate accurate postfix output. Additionally, from the theoretical and planning aspect of software engineering, flowcharts will be used to visually represent the conversion process, aiding in algorithm development and comprehension.

Contents table

Application brief

Throughout the section, the general idea behind the intended application will be provided, along with an overview of its overall benefit and the key aspects that it covers. Moreover, the section will explore the features and functionality of the application, including the key aspects that it will be covering and what the intention is for the end product relative to its overall benefit for the user and its potential to improve the accuracy and efficiency of working with mathematical expressions.

Infix to postfix notation:

Definitions of infix and postfix notations:

Infix: *Infix notation is the commonly used method of writing mathematical expressions, where operators are placed between the operands. For example, in the expression "3 + 5," the operator "+" is placed between the operands "3" and "5." Infix notation follows the standard mathematical conventions, including the use of parentheses to indicate the precedence of operations.*

Postfix: *Postfix notation, also known as Reverse Polish Notation (RPN), is an alternative way of representing mathematical expressions. In postfix notation, operators are placed after the operands. Using the same example, the expression "3 5 +" in postfix notation represents "3 + 5." Postfix notation eliminates the need for parentheses to indicate the precedence of operations as it relies on the order of operands and operators.*

Why postfix notation:

Why is infix used: *Infix notation is predominantly used in mathematical expressions due to its familiarity and ease of human readability. The placement of operators between operands aligns with traditional mathematical conventions and is more intuitive for humans to interpret and write. Infix notation allows us to express mathematical expressions in a way that closely resembles how we speak and think about mathematical operations. The use of parentheses in infix notation also aids in explicitly indicating the precedence of operations and helps clarify complex expressions. While infix notation may be more natural for human comprehension, there are instances where postfix notation is preferrable, primarily in technical computational applications.*

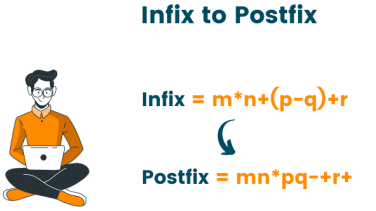


Figure 1 – Infix and postfix example

Why postfix notation might be used: *As stated by* (Geeks For Geeks, 2022)*, Postfix notation offers several benefits specifically for computational applications. First, it eliminates the need for parentheses to indicate the order of operations. By placing operators after their operands, postfix notation ensures unambiguous evaluation, making it easier to write and evaluate mathematical expressions programmatically. This simplifies the parsing and processing of expressions, reducing the complexity of expression evaluation algorithms.*

Moreover, postfix notation enables straightforward implementation of expression evaluation using a stack-based approach. As each operand is encountered, it is pushed onto the stack. When an operator is encountered, the necessary number of operands are popped from the stack, the operation is performed, and the result is pushed back onto the stack. This stack-based evaluation simplifies expression parsing and reduces the need for recursive evaluation or complex operator precedence rules.

Essentially, through eliminating the need for parentheses and precedence, postfix notation offers and unambiguous representation of mathematical expressions that is well-suited for computational applications.

Application of postfix notation:

Reverse Polish Notation (RPN) calculators: *Postfix notation, particularly Reverse Polish Notation, is commonly used in calculators and mathematical tools. RPN calculators operate on the principle of evaluating expressions in postfix form, providing a streamlined and efficient approach to perform complex calculations.*

Virtual machines and interpreters: *Postfix notation can be leveraged in the design of virtual machines and interpreters for programming languages. Intermediate representations, such as bytecode or abstract syntax trees, may use postfix-like structures for expression evaluation, optimizing code execution and reducing parsing complexity.*

Compiler optimizations: *Postfix notation or postfix-like representations can be employed during compiler optimizations. Transformations, such as common subexpression elimination, constant folding, and loop invariant code motion, can benefit from postfix-like structures to simplify expression evaluation and identify optimization opportunities.*

Computer algebra systems: *Postfix notation is often utilized in computer algebra systems, where symbolic manipulation and evaluation of mathematical expressions play a significant role. Postfix-like representations facilitate algebraic simplifications, differentiation, integration, and other advanced mathematical operations.*

Computer-aided design (CAD) software: *Some CAD software applications incorporate postfix notation for precise geometric calculations. Postfix notation allows for the straightforward representation of complex geometric formulas, enabling efficient evaluation and manipulation of 2D and 3D models.*

Postfix-based programming languages: *There are programming languages specifically designed around postfix notation, such as Forth. These languages use postfix expressions as the primary syntax for writing programs. Programmers write code in postfix notation, and the interpreter or compiler evaluates the expressions based on stack-based algorithms.*

Project scope:

Conversion:

*Through the use of the **Shunting Yard algorithm**, conversion of infix to postfix is made possible. In order to implement said algorithm, two data structures are utilized: a queue and a stack. Operators and operands are read from left to right and pushed to the stack or the queue based on their precedence within "BIDMAS". Operands are pushed into the queue in order of the postfix index. Higher precedence operators are pushed onto the stack, while lower precedence operands are pushed to the queue. Once the entire expression has been processed, the remaining operators in the stack are popped and pushed to the queue. The algorithm's logic will be more evident as it is explained within code examples.*

Evaluation:

*To evaluate a postfix expression after it has been converted, a **stack-based approach** is utilized. The operands are pushed onto a stack as they are read from left to right. When an operator is encountered, the top two operands are popped from the stack and the operator is applied to them. The result of the operation is pushed back onto the*

stack. This process is repeated until the entire expression has been evaluated, and the final result is the only item left on the stack. This approach allows for efficient and accurate evaluation of mathematical expressions in postfix notation.

Algorithm example:

Shunting Yard: *For an example such as* **(5*4+3*6)-1**, *the algorithms works as follows: An empty stack and an empty queue are created. The first token is 5, which is an operand, so we add it to the queue. The next token is *****, which is an operator, so we push it onto the stack. The next token is 4, which is another operand, so we add it to the queue. The next token is +, which is an operator with lower precedence than *****, so we remove the ***** from the stack and push it to the queue and push the + onto the stack. The next token is 3, which is another operand, so we add it to the queue. The next token is *****, which is an operator with higher precedence than +, so we simply push it on top of the + on the stack. The next token is 6, which is another operand, so we add it to the queue. From here, the first part of the expression, (54+36) has been processed. Since the following token is a closing parenthesis, then the stack is emptied and is pushed onto the queue. The ***** is removed from the stack and is enqueued, followed by the +.. Finally, the next token is an operand, 1, so it is added to the queue, and since there is only one more token left, it is enqueued rather than pushed to the stack. Resulting in a final expression of* → **5 4 * 3 6 * + 1 –**

Post-fix evaluation: *An empty stack is initialized, with the application iterating over each token in the postfix notation. If the token is an operand and the stack is empty, it is pushed onto the stack. If the token is an operator, the operand atop the stack and the one right below are taken out and apply the operator. Using the previously shown example, it will work as follows:*

*The first token encountered is 5, so it is pushed onto the stack. The next token is 4, so we it is pushed onto the stack as well. The next token is *****, for that, the operand at t the top of the stack (4) and the one below (5) are taken out and apply the (*) operator. This will result in (20), which is then pushed back onto the stack. The same process is then repeated until the final output of (38) is reached.*

Note: *More would need to be added on top of the original algorithm including handling of functions and logarithms. However, in terms of the working process, the shown example covers the core concept of the Shunting Yard Algorithm and how post-fix evaluation is processed.*

Process flow and ideal output

In the following section, the diagrammatic representation of the algorithm's logic would be showcased as well as the testing plan and the expected results from the application. The diagrammatic representation will provide a visual depiction of the step-by-step flow of the algorithm, illustrating how the different components interact and the order in which they are executed. Said visual representation helps in understanding the algorithm's structure and aids in identifying potential areas for optimization or improvement. The testing plan outlines the approach for testing the algorithm's functionality and performance. It defines the test cases, expected inputs, and desired outputs, ensuring that the algorithm produces the correct results under various scenarios.

Testing plan:

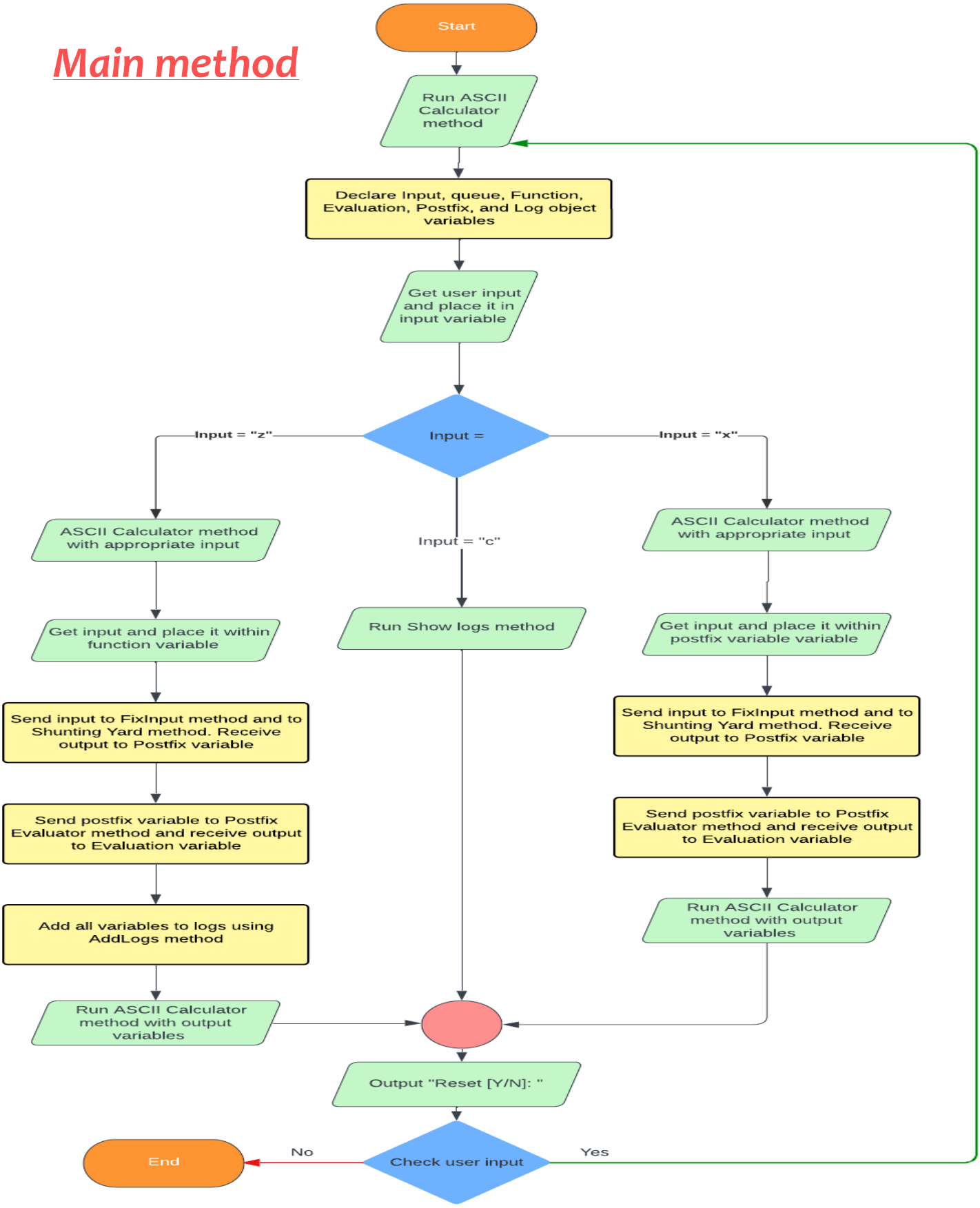
Test name	Purpose of test	Input	Expected result
Basic conversion	Test conversion capabilities with a basic expression	An expression that does not uses functions or nested parentheses	To output the postfix expression
Basic evaluation	Test evaluation capabilities with a basic expression	The expression generated from the previous test	To output a definite value
Direct postfix input	Test inputting postfix directly for evaluation	A postfix expression	To output a definite value
Handling missing asterisk	Ensure that user inputs missing asterisks still work	An expression with multiplying brackets or variables	To add missing asterisks and continue as required
Using functions	Conversion and evaluation of functions and logs	An expression that uses log, ln, sin, cos, or tan	To perform the required functions
Complex expression	Testing all functionality with a complex expression	An expression that uses nested parentheses and functions and is long	To output the postfix expression and evaluation
Logs	Testing how well the application logs u	N / A	Show logs of all user inputs

Flowchart:

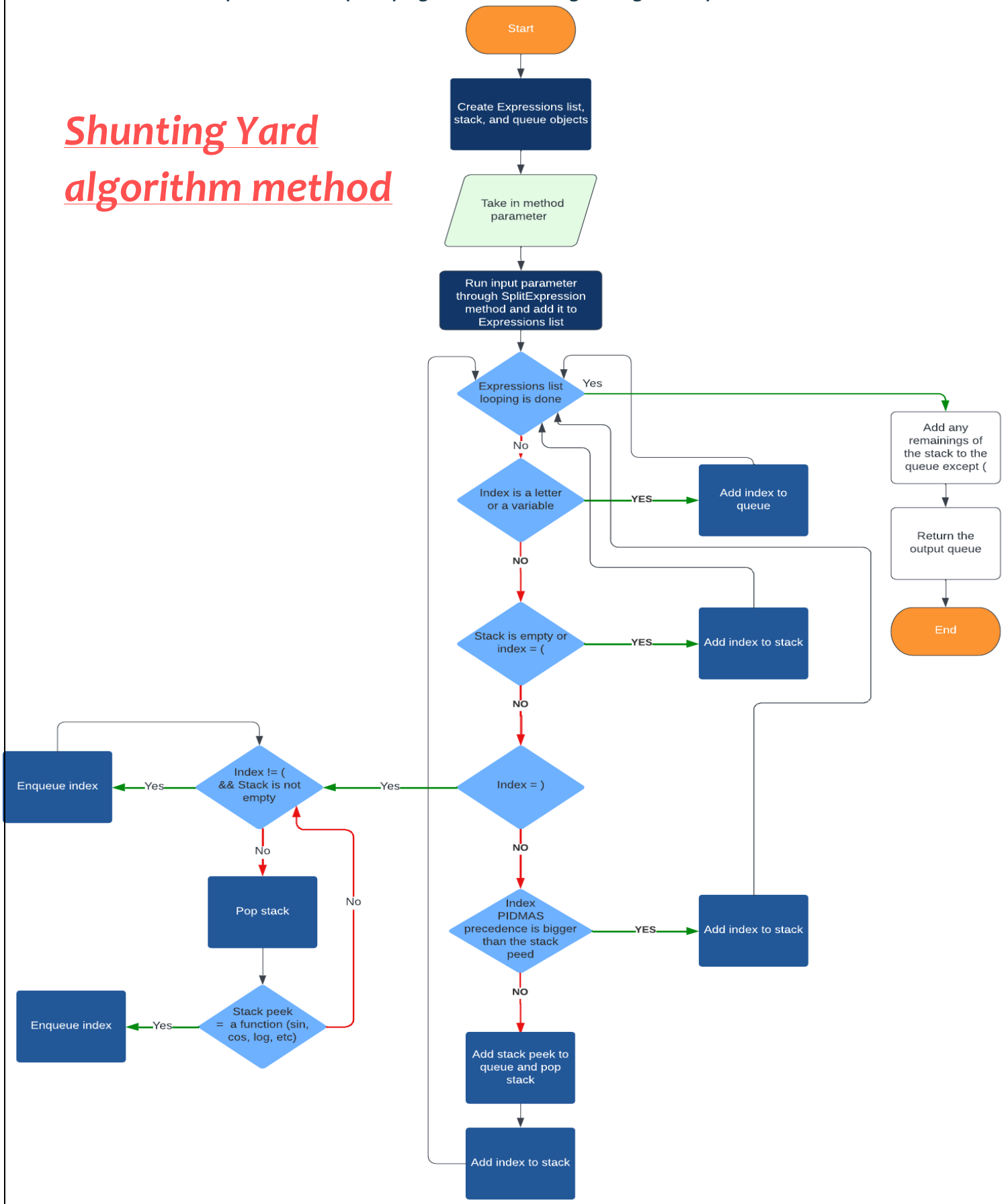
Flowcharts are vital tools for visualizing and documenting processes, making them essential in various fields such as software development, business analysis, and project management. Flowcharts provide a clear and concise representation of the sequential flow of activities, decision points, and outcomes within a system or process. They help in understanding complex systems, identifying bottlenecks, analyzing dependencies, and improving overall efficiency. By presenting a step-by-step visual guide, flowcharts enable effective communication and collaboration among team members, ensuring everyone is on the same page and facilitating problem-solving and decision-making.

It should be noted that in the following section, due to the size and complexity of the application, the flowcharts have been separated as to make it so each chart represents a single function within the program. Since flow charts are meant to be easily digestible and the developer or reader is able to understand the logic almost at a glimpse, having one large flowchart would be very difficult to comprehend easily and for that reason this method of separating the flowchart into functions was utilized. Not to mention that it will also be difficult to fit within a single page.

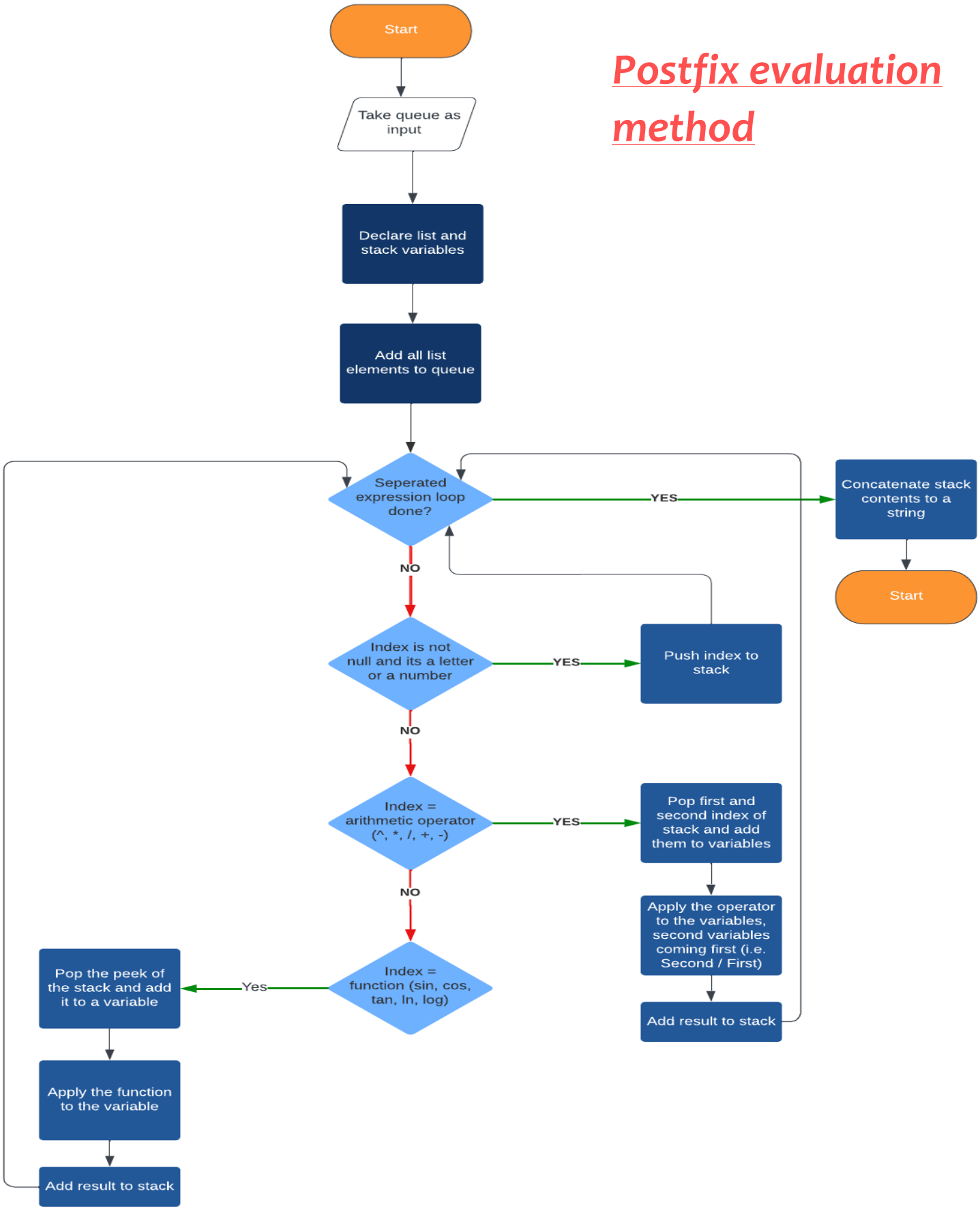
Main method


Start
Run ASCII Calculator method
Declare Input, queue, Function, Evaluation, Postfix, and Log object variables
Get user input and place it in input variable
Input =
Input = "z"
Input = "c"
Input = "x"
ASCII Calculator method with appropriate input
Get input and place it within function variable
Send input to FixInput method and to Shunting Yard method. Receive output to Postfix variable
Send postfix variable to Postfix Evaluator method and receive output to Evaluation variable
Add all variables to logs using AddLogs method
Run ASCII Calculator method with output variables
Run Show logs method
ASCII Calculator method with appropriate input
Get input and place it within postfix variable variable
Send input to FixInput method and to Shunting Yard method. Receive output to Postfix variable
Send postfix variable to Postfix Evaluator method and receive output to Evaluation variable
Run ASCII Calculator method with output variables
Output "Reset [Y/N]: "
Check user input
No
Yes
End

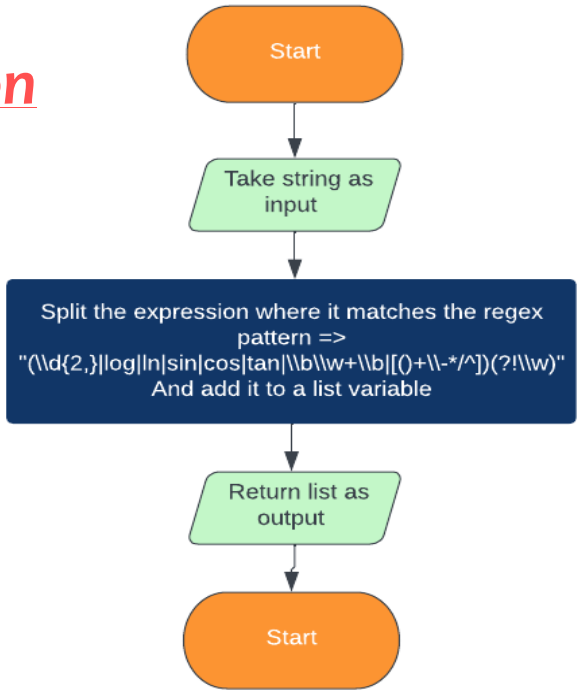
Shunting Yard algorithm method



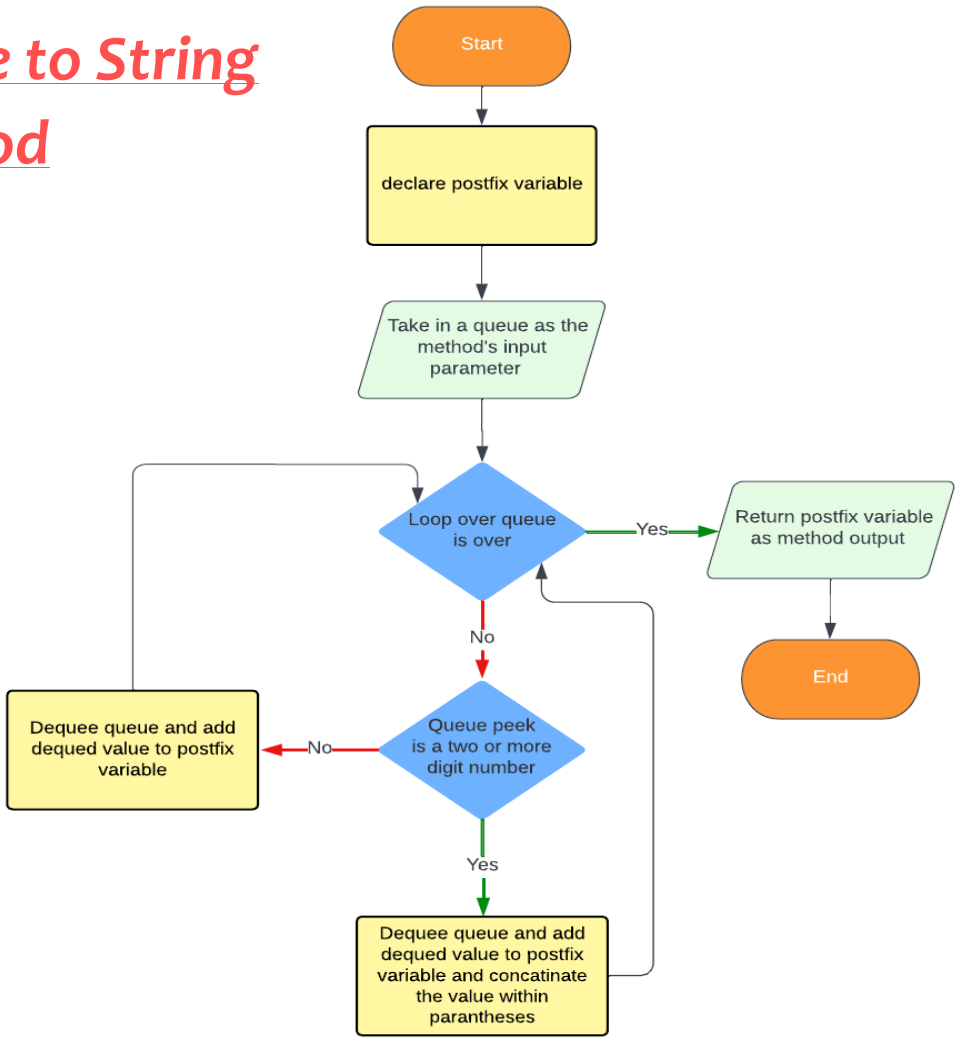
Postfix evaluation method



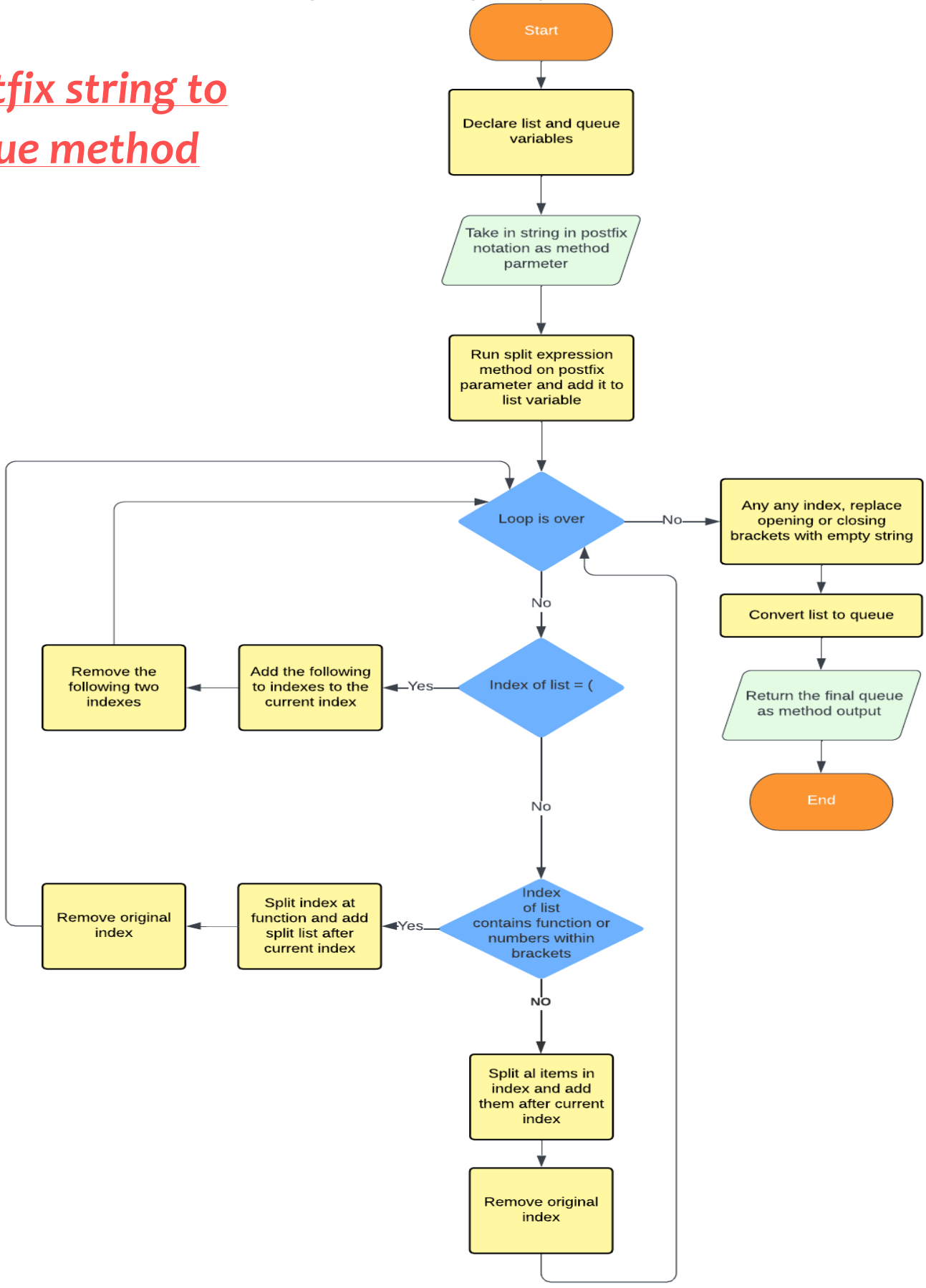
Split Expression method



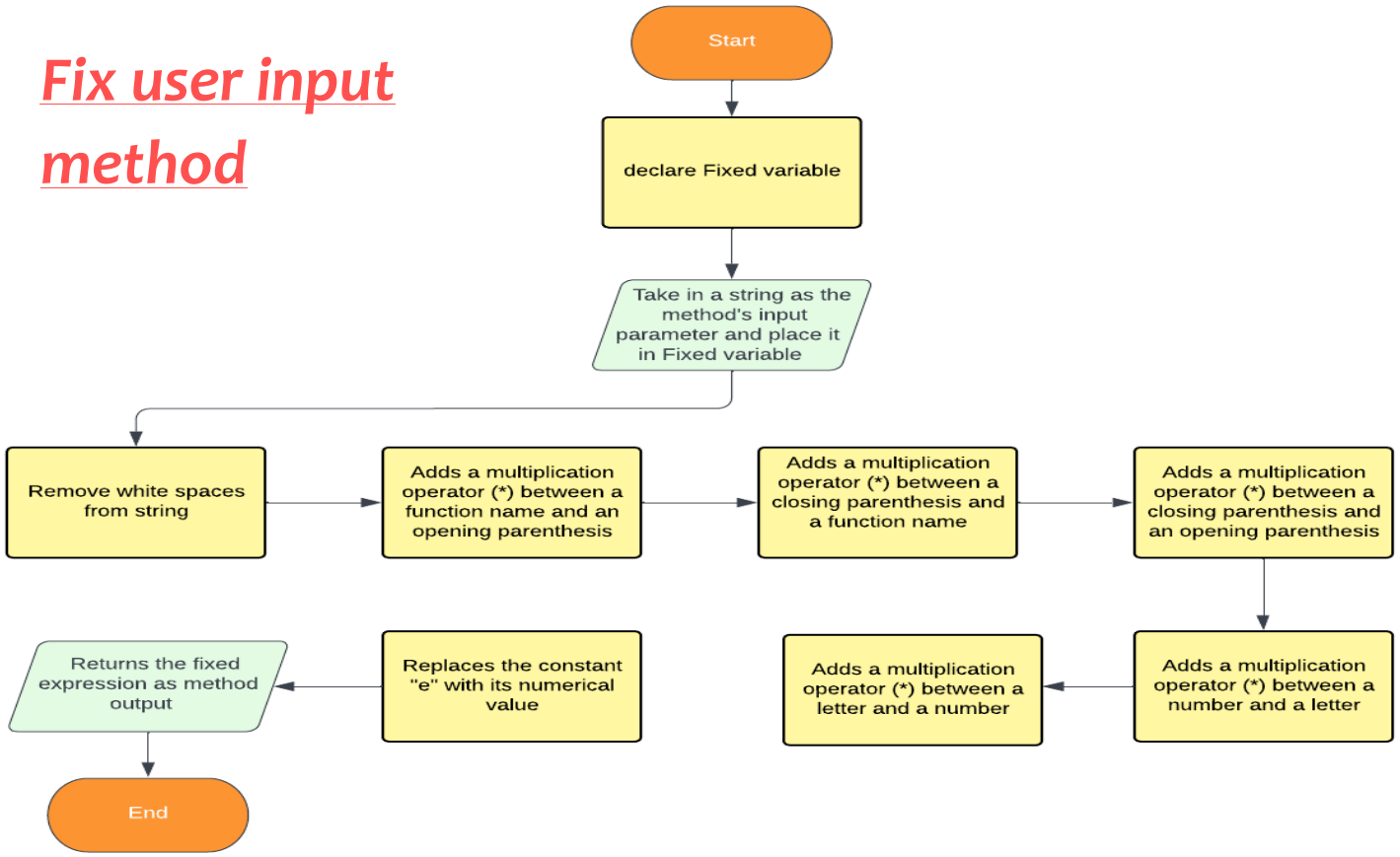
Queue to String method



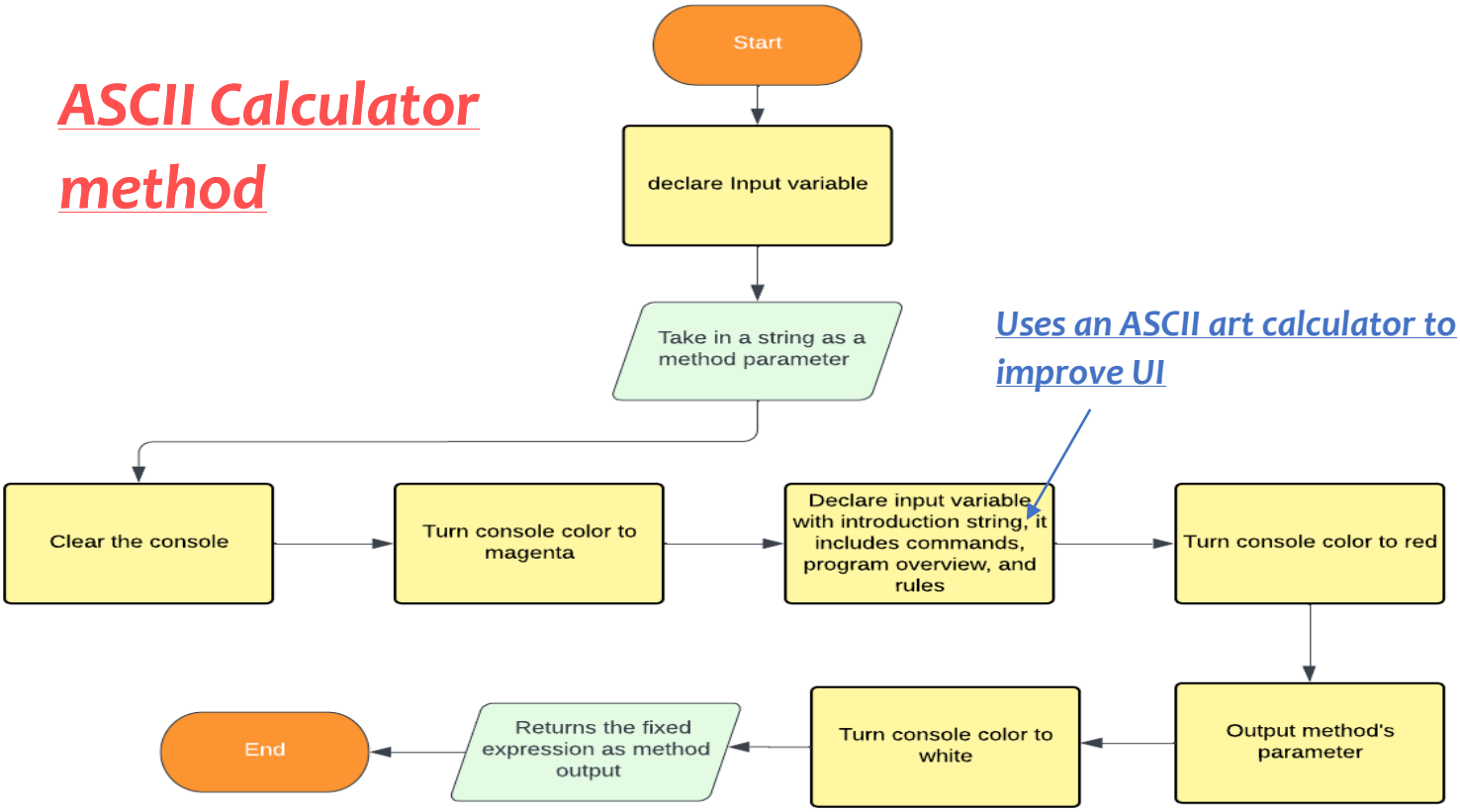
Postfix string to queue method

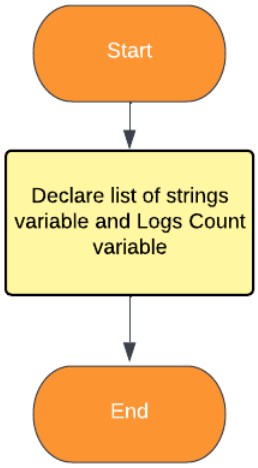


Fix user input method

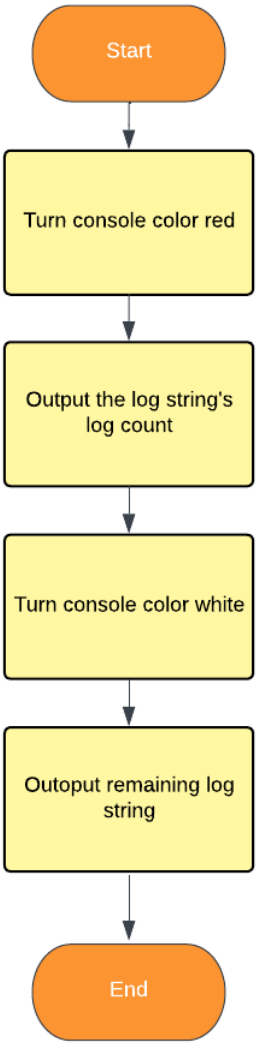


ASCII Calculator method





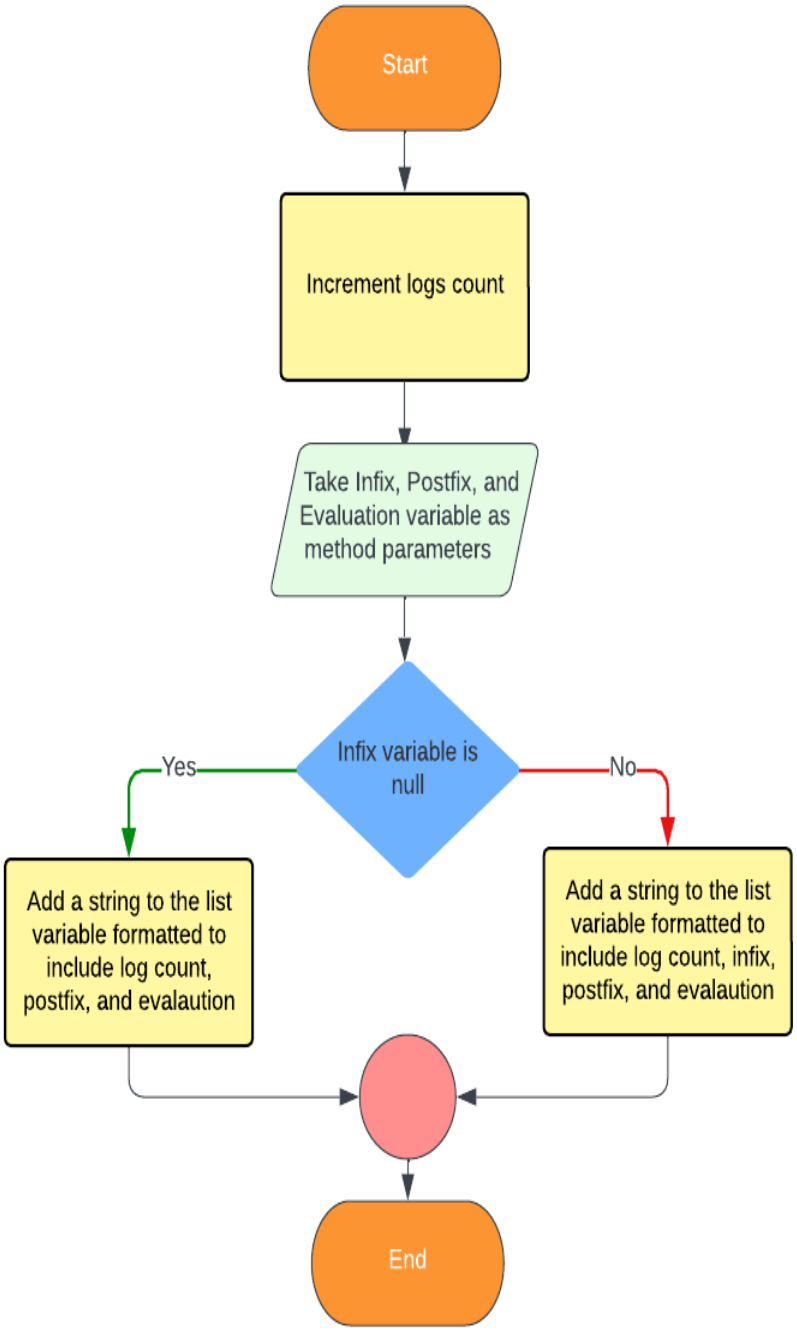
Class properties



Show logs method

New class (Logs)

Add logs method



Codebase

The following section will include the final working base alongside a URL and QR code that link to a GitHub Gist repository that deploys the code. It should also be noted that said code not only contains the logic itself but as well as adequate documentation through the form of comments added where the code is not entirely evident and requires certain elaboration.

Link to code:

https://gist.github.com/Saucter/5b831735811ea55f0e51b00b48bad2c3

Codebase:

```
using System.Collections;
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using System.Globalization;
using System.Text;
using System.Linq.Expressions;

namespace Project
{
    public class PostfixToInfixConverter
    {
        protected readonly static Logs logs = new Logs();


        // Entry point of the program
        static void Main(string[] args)
        {
            while (true)
            {
                ASCII_Calculator("Your input: ");  // Display ASCII
calculator header and prompt for user input
```

```
                string Input = Console.ReadLine().ToLower();  // Read user
input and convert it to lowercase
                string Function = "";

                //Ensure that if a wrong input is made at any of the
options, the error won't crash the application
                try
                {
                    switch (Input)
                    {
                        case "z":
                            ASCII_Calculator("Expression: ");  // Prompt the
user for an expression
                            Function = FixUserInput(Console.ReadLine());  //
Sanitize the input

                            Queue<string> queue = ShuntingYard(Function);
// Convert infix to postfix using the Shunting Yard algorithm
                            string Postfix = ToString(queue);  // Convert
the resulting queue to a string
                            string Evaluation =
PostfixEvaluation(ShuntingYard(Function));  // Evaluate the postfix
expression

                            logs.AddLog(Function, Postfix, Evaluation);  //
Add the input and results to the log

                            RedTitle("The postfix expression: ", Postfix +
"\n");  // Display the postfix expression
                            RedTitle("The postfix evaluation: ", Evaluation
+ "\n");  // Display the postfix evaluation

                            RedTitle("Back [Y/N]: ", "");  // Prompt the
user to continue or exit

                            if (Console.ReadLine().ToLower() == "n")
Environment.Exit(0);  // If the user chooses to exit, terminate the program
                            else continue;  // Continue to the next
iteration
                            break;

                        case "x":
                            ASCII_Calculator("Put multi-digit numbers within
brackets\n");  // Display instructions for multi-digit numbers
```

```
                            RedTitle("Postfix notation: ", "");  // Prompt
the user for a postfix expression
                            string PostfixExpression =
Console.ReadLine().Trim();

                            string EvaluatedExpression =
PostfixEvaluation(ToQueue(PostfixExpression));  // Evaluate the postfix
expression

                            logs.AddLog("", PostfixExpression,
EvaluatedExpression);  // Add the input and results to the log

                            RedTitle("The postfix expression: ",
EvaluatedExpression + "\n");  // Display the postfix expression

                            RedTitle("Back [Y/N]: ", "");  // Prompt the
user to continue or exit

                            if (Console.ReadLine().ToLower() == "n")
Environment.Exit(0);  // If the user chooses to exit, terminate the program
                            else continue;  // Continue to the next
iteration
                            break;

                        case "c":
                            ASCII_Calculator("");  // Display the ASCII
calculator header and logs

                            logs.ShowLogs();  // Show usage logs

                            RedTitle("Back [Y/N]: ", "");  // Prompt the
user to continue or exit

                            if (Console.ReadLine().ToLower() == "n")
Environment.Exit(0);  // If the user chooses to exit, terminate the program
                            else continue;  // Continue to the next
iteration
                            break;

                        default:
                            ASCII_Calculator("");  // Display the ASCII
calculator header
```

```
                            Console.WriteLine("That's an invalid option");
// Show error message for invalid option

                            RedTitle("Back [Y/N]: ", "");  // Prompt the
user to continue or exit

                            if (Console.ReadLine().ToLower() == "n")
Environment.Exit(0);  // If the user chooses to exit, terminate the program
                            else continue;  // Continue to the next
iteration
                            break;
                        }
                    }
                    catch
                    {
                        ASCII_Calculator("");  // Display the ASCII calculator
header

                        Console.WriteLine("The expression is invalid");  // Show
error message for invalid input

                        RedTitle("Back [Y/N]: ", "");  // Prompt the user to
continue or exit

                        if (Console.ReadLine().ToLower() == "n")
Environment.Exit(0);  // If the user chooses to exit, terminate the program
                        else continue;  // Continue to the next iteration
                        break;

                    }
                }
            }


            //Converts Infix expressions to Postfix and returns the output queue
            static Queue<string> ShuntingYard(string Expression)
            {
                List<string> SeperatedExpression = SplitExpression(Expression);
                Stack<string> stack = new Stack<string>();
                Queue<string> queue = new Queue<string>();

                foreach (var value in SeperatedExpression)
                {
                    if (char.IsNumber(value.ToCharArray()[0]) ||
(char.IsLetter(value.ToCharArray()[0]) && value.Length == 1))
```

```
                {
                    queue.Enqueue(value);  // Enqueue numbers and single-
letter variables
                }
                else
                {
                    if (stack.Count() == 0 || value == "(")
                    {
                        stack.Push(value);  // Push opening parentheses or
operators onto the stack
                    }
                    else if (value == ")")
                    {
                        for (int i = 0; i < stack.Count(); i++)
                        {
                            if (stack.Peek().ToString() != "(")
                            {
                                queue.Enqueue(stack.Pop());  // Pop
operators until opening parenthesis is found
                            }
                            else
                            {
                                stack.Pop();  // Pop the opening parenthesis

                                if (stack.Peek().ToString() == "sin" ||
stack.Peek().ToString() == "cos" || stack.Peek().ToString() == "tan" ||
stack.Peek().ToString() == "ln" || stack.Peek().ToString() == "log")
                                {
                                    queue.Enqueue(stack.Pop());  // Pop
functions and enqueue them
                                }

                                break;
                            }
                        }
                    }
                    else if (BIDMAS(stack.Peek().ToString()) >
BIDMAS(value.ToString()) || stack.Peek().ToString().Trim() == "(")
                    {
                        stack.Push(value);  // Push operators with higher
precedence onto the stack
                    }
                    else
                    {
```

```
                        queue.Enqueue(stack.Pop());  // Pop operators with
lower precedence and enqueue them
                        stack.Push(value);  // Push the current operator
onto the stack
                    }
                }
            }

            if (stack.Count() != 0)
            {
                for (int i = 0; i < stack.Count(); i++)
                {
                    queue.Enqueue((stack.Peek() != "(") ? stack.Pop() :
null);  // Pop any remaining operators and enqueue them
                }
            }

            return queue;  // Return the queue containing the output
        }


        //Evaluates the value of a given postfix expression
        static string PostfixEvaluation(Queue<string> queue)
        {
            List<string> SeperatedExpression = new List<string>();
            Stack<double> stack = new Stack<double>();

            int StartingCount = queue.Count();

            // Dequeue the items from the queue and store them in a list
            for (int i = 0; i < StartingCount; i++)
            {
                SeperatedExpression.Add(queue.Dequeue());
            }

            // Evaluate the postfix expression
            for (int i = 0; i < SeperatedExpression.Count(); i++)
            {
                string CurrentValue = SeperatedExpression[i];

                if ((!string.IsNullOrEmpty(CurrentValue)) ?
char.IsNumber(CurrentValue.ToCharArray()[0]) ||
(char.IsLetter(CurrentValue.ToCharArray()[0]) && CurrentValue.Length == 1) :
false)
                {
```

```
                    stack.Push(int.Parse(SeperatedExpression[i]));  // Push
numbers onto the stack
                    continue;
                }

                switch (CurrentValue)
                {
                    case "*":
                        stack.Push(stack.Pop() * stack.Pop());  // Perform
multiplication
                        break;
                    case "/":
                        double Divisor = stack.Pop();
                        double Dividend = stack.Pop();
                        stack.Push(Dividend / Divisor);  // Perform division
                        break;
                    case "^":
                        double Exponent = stack.Pop();
                        double Base = stack.Pop();
                        stack.Push((float)Math.Pow(Base, Exponent));  //
Perform exponentiation
                        break;
                    case "+":
                        stack.Push(stack.Pop() + stack.Pop());  // Perform
addition
                        break;
                    case "-":
                        double Subtrahend = stack.Pop();
                        double Minuend = stack.Pop();
                        stack.Push(Minuend - Subtrahend);  // Perform
subtraction
                        break;
                    case "sin":
                        stack.Push(Math.Sin(stack.Pop()));  // Perform sine
calculation
                        break;
                    case "cos":
                        stack.Push(Math.Cos(stack.Pop()));  // Perform
cosine calculation
                        break;
                    case "tan":
                        stack.Push(Math.Tan(stack.Pop()));  // Perform
tangent calculation
                        break;
                    case "log":
```

```
                        stack.Push(Math.Log(stack.Pop()));  // Perform
logarithm calculation
                        break;
                    case "ln":
                        stack.Push(Math.Log(stack.Pop(), Math.E));  //
Perform natural logarithm calculation
                        break;
                }
            }

            string Evaluation = "";
            int StackTotal = stack.Count();

            // Build the evaluation string by popping the items from the
stack
            for (int i = 0; i < StackTotal; i++)
            {
                Evaluation += stack.Pop();
            }

            return Evaluation;  // Return the evaluated expression
        }


        // Checks the hierarchy of a given operator and returns an integer
        static int BIDMAS(string Operator)
        {

            List<string> Precedence = new List<string>() { "sin", "cos",
"tan", "log", "ln", "(", "^", "/", "*", "+", "-" }; // Define the precedence
of operators in descending order

            return (Precedence.Contains(Operator)) ?
Precedence.IndexOf(Operator) : -1; // Return the index of the operator in
the precedence list, or -1 if not found
        }


        // Splits the expression where there are functions or multi-digit
numbers
        static List<string> SplitExpression(string Expression)
        {
            // Use regular expression to split the expression into
individual tokens
```

```
            List<string> SplitExpression = Regex.Split(Expression,
"(\\d{2,}|log|ln|sin|cos|tan|\\b\\w+\\b|[()+\\-*/^])(?!\\w)")
                .Where(x => !string.IsNullOrWhiteSpace(x)).ToList();

            return SplitExpression.Where(x =>
!string.IsNullOrWhiteSpace(x)).ToList();
        }


        // Used to convert a postfix expression into a usable queue
        static Queue<string> ToQueue(string Postfix)
        {
            List<string> Split = SplitExpression(Postfix);
            Queue<string> queue = new Queue<string>();

            for (int i = 0; i < Split.Count(); i++)
            {
                if (Split[i] == "(")
                {
                    // Combines the opening parantheses, number within, and
closing parantheses into one index
                    Split[i] += Split[i + 1] + Split[i + 2];
                    Split.RemoveRange(i + 1, 2);
                }
                else if (Regex.IsMatch(Split[i],
@"(log|ln|sin|cos|tan)|((?>!\()\d{2,}(?!\)))"))
                {
                    // Split the token further if it contains functions or
multi-digit numbers
                    Split.InsertRange(i + 1, Regex.Split(Split[i],
@"(log|ln|sin|cos|tan)|((?>!\()\d{2,}(?!\)))")
                        .Where(x => !string.IsNullOrWhiteSpace(x)));
                    Split.RemoveAt(i);
                }
                else
                {
                    // If neither of the previous conditions apply, then the
index is a multi-digit number that is NOT in brackets, for that it gets
split into individual numbers
                    Split.InsertRange(i + 1, Split[i].ToCharArray().Select(x
=> x.ToString()));
                    Split.RemoveAt(i);
                }
            }
```

```
            // Remove parentheses and enqueue the tokens
            Split = Split.Select(s => s.Replace("(", "").Replace(")",
"")).ToList();
            Split.ForEach(x => queue.Enqueue(x));
            return queue;
        }


        // Converts a queue of tokens back to a string representation of
postfix expression
        static string ToString(Queue<string> queue)
        {
            string Postfix = "";
            int QueueTotal = queue.Count();

            for (int i = 0; i < QueueTotal; i++)
            {
                try
                {
                    // Check if the token is a multi-digit number and
enclose it in parentheses
                    Postfix += (queue.Peek().All(x => char.IsNumber(x)) &&
queue.Peek().Length > 1)
                        ? "(" + queue.Dequeue().ToString() + ")"
                        : queue.Dequeue();
                }
                catch { }
            }

            return Postfix;
        }


        // Fixes the user input by applying necessary transformations
        static string FixUserInput(string Expression)
        {
            string FixedInput = Expression.Replace(" ", "").ToLower();

            // Replaces all spaces and converts the input to lowercase
            FixedInput = Expression.Replace(" ", "").ToLower();

            // Adds a multiplication operator (*) between a function name
and an opening parenthesis
            FixedInput = Regex.Replace(FixedInput,
@"(?<!\b(log|ln|sin|cos|tan))([A-Za-z]+)(\()", "$2$3");
```

```
            // Adds a multiplication operator (*) between a closing
parenthesis and a function name
            FixedInput = Regex.Replace(FixedInput, @"(\))([A-Za-z]+)(?!
*\()", "$1*$2");

            // Adds a multiplication operator (*) between a closing
parenthesis and an opening parenthesis
            FixedInput = Regex.Replace(FixedInput, @"(\))(\()", "$1*$2");

            // Adds a multiplication operator (*) between a number and a
letter
            FixedInput = Regex.Replace(FixedInput, "(\\d+)([a-zA-Z])",
"$1*$2");

            // Adds a multiplication operator (*) between a letter and a
number
            FixedInput = Regex.Replace(FixedInput, "([a-zA-Z])(\\d+)",
"$2*$1");

            // Replaces the constant "e" with its numerical value
            FixedInput = FixedInput.Replace("e", Math.E.ToString());

            // Wraps the fixed input with parentheses to ensure proper
evaluation
            return $"({FixedInput})";
        }


        //Starting screen including the main info about the program
        static void ASCII_Calculator(string Extra)
        {
            string Info = @"
_____________________
|  ___________________  |
| | Infix - Postfix | |                              Overview of the program
Commands                                    Scope
| |___________________| |
_____________________________________
_____________________________________
_____________________________________
|  ___ ___ ___   ___  |
| | 7 | 8 | 9 | | + | |                        This program allows you to
convert any                    (Z) -> Infix to postfix
1. Utilize computer-based mathematical
```

```
| |___|___|___| |___| |                       infix-based mathematical
expression to
operators (brackets, ^, /, *, +, -)
| | 4 | 5 | 6 | | - | |                       postfix notation. A popular
expression                    (X) -> Evaluate RPN expression
| |___|___|___| |___| |                       format to increase computational
speed
2. Trig functions and logarithm should
| | 1 | 2 | 3 | | x | |                       and efficiency.
(C) -> Show usage logs                                 be in short format
(sin, not sine)
| |___|___|___| |___| |
| | . | 0 | = | | / | |                       The program also includes
postfix eva-
3. Only log of base 10 is available
| |___|___|___| |___| |                       -luation for the converted
expressions
|_____________________|                       with or without numeric
variables
4. Variable are lowercase, e is unusable";


            //Clears the screen in order to not add clutter to the screen
and then showcases the program's information
            Console.Clear();
            Console.ForegroundColor = ConsoleColor.Magenta;
            Console.WriteLine(Info + "\n\n\n");

            //Extra information to be added. Can be changed when calling the
method
            Console.ForegroundColor = ConsoleColor.Red;
            Console.Write(Extra);
            Console.ForegroundColor = ConsoleColor.White;
        }


        // Displays a title in red color followed by content in white color.
Reduces code repetition
        static void RedTitle(string Title, string Content)
        {
            Console.ForegroundColor = ConsoleColor.Red;  // Set console text
color to red
            Console.Write(Title);  // Display the title in red color
            Console.ForegroundColor = ConsoleColor.White;  // Set console
text color back to white
```

```
            Console.Write(Content);  // Display the content in white color
        }
    }


    //A log class responsible for adding log functionality. Added as a clase
to be used globally through the application
    public class Logs
    {
        protected List<string> AppLogs = new List<string>();  // List to
store the application logs
        int LogsCount = 0;  // Counter for the number of logs


        // Method to add a log entry to the list. Showcasesn the user input
and showcauses the notation and evaluation
        public void AddLog(string Infix, string Postfix, string Evaluation)
        {
            LogsCount++;  // Increment the logs count
            StringBuilder sm = new StringBuilder();
            sm.AppendFormat("{0}) =>  {1} RPN: {2} | Evaluation: {3}",
LogsCount, (!string.IsNullOrEmpty(Infix) ? (" Infix: " + Infix + " | ") :
""), Postfix, Evaluation);
            AppLogs.Add(sm.ToString());  // Add the formatted log entry to
the list
        }


        // Method to display all the logs
        public void ShowLogs()
        {
            foreach (var log in AppLogs)
            {
                Console.ForegroundColor = ConsoleColor.Red;  // Set console
text color to red
                Console.Write(log.Substring(0, log.IndexOf(">")));  //
Display the log count in red color
                Console.ForegroundColor = ConsoleColor.White;  // Set
console text color back to white
                Console.WriteLine(log.Substring(log.IndexOf(">") + 2));  //
Display the remaining log content in white color
            }
        }
    }
}
```

Test results

The subsequent section will go through any issues that were encountered during the testing phase of the project, how they were approached, and what the end solution for them was.

Testing table:

Test name	Input	Expected result	Actual result
Basic conversion	An expression that does not uses functions or nested parentheses	To output the postfix expression	**Worked flawlessly without issues**
Basic evaluation	The expression generated from the previous test	To output a definite value	**Worked flawlessly without issues**
Direct postfix input	A postfix expression	To output a definite value	**Split expression method was taking two numbers that are next to each other as a single index**
Handling missing asterisk	An expression with multiplying brackets or variables	To add missing asterisks and continue as required	**It did not understand mathematical notation for variable and parentheses multiplication**
Using functions	An expression that uses log, ln, sin, cos, or tan	To perform the required functions	**Worked flawlessly without issues**
Complex expression	An expression that uses nested parentheses and functions and is long	To output the postfix expression and evaluation	**Worked without issues after the two aforementioned ones were solved**
Logs	N / A	Show logs of all user inputs	**Worked flawlessly without issues**

Solutions:

Direct postfix input:

Problem: *Essentially, the problem was that since the output queue of converting the infix expression to postfix was not available so there was not conversion in the first place but rather the postfix expression was directly taken as a string from the user, there was no way for the program to differentiate between two separate values such as and a two-digit value in a postfix expression.*

Solution: *A method was created called "ToQueue" which essentially takes the postfix expression and converts it into a queue. However, that alone would not work, as there is still no way to detect whether or not the expression number is two digit or two separates one-digit values. For that the user is told before inputting the postfix expression to wrap two or more-digit values within brackets. Using that, the program splits the expression using the Split Expression method and then when it detects an open parenthesis, it concatenates it with the following two indexes (i.e., the number and closing parentheses). At the end of the method, the parentheses is simply removed from the list to ensure that the brackets do not go into the output queue. This list is then converted into a queue and the operation continues as normal.*

Handling missing asterisks:

Problem: *The issue at hand was related to the fact that the user could potentially input a logically invalid expression that is not invalid in terms of mathematical notation. In that they could enter an expression where they utilize brackets as multiplication or place the number next to the variable without a multiplication sign. An example of which might look like*

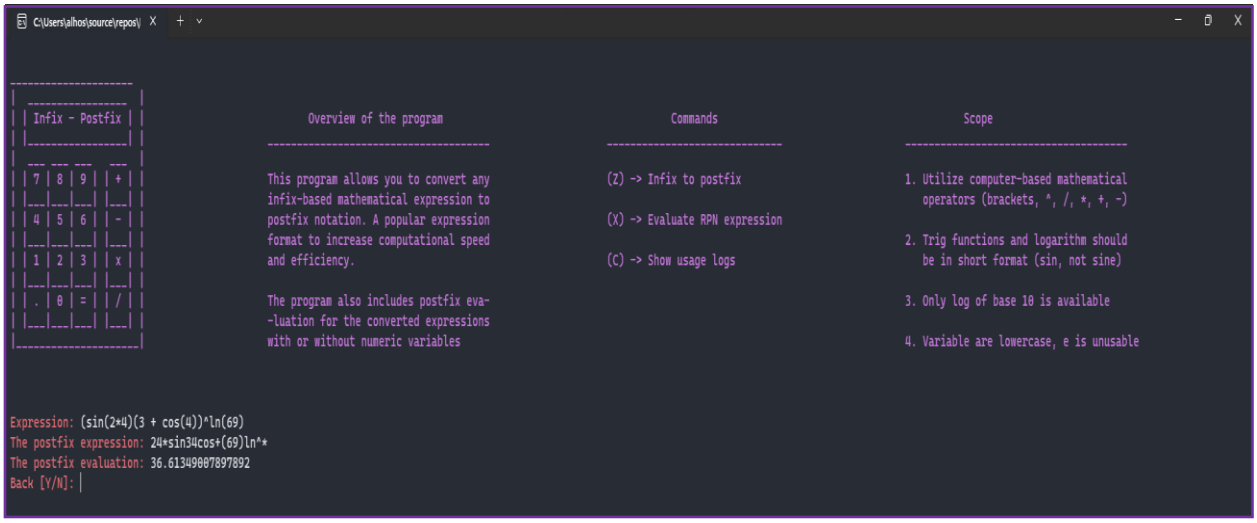
- *2x instead of 2 * x*
- *(8/3)(55+8) instead of (8/3) * (55+8)*
- *9(8) instead of 9 * (8)*
- *(8)9 instead of 8 * 9*

Solution: *To solve this, a method called "FixUserInput" simply takes the expression before it gets sent to the conversion and evaluation method and simply checks for any of the aforementioned occurrences using various regex patterns and adds an asterisk by replacing the occurrence with the same one albeit including the asterisk. The same method is also responsible for removing any white spaces that might interfere with the logic and for replacing the letter "e" with Euler's value.*

Running example

This section will simply showcase a running example of the application when being utilized for the evaluation and conversion of a relatively complex infix and postfix notation.

Infix conversion and evaluation (option "z"):



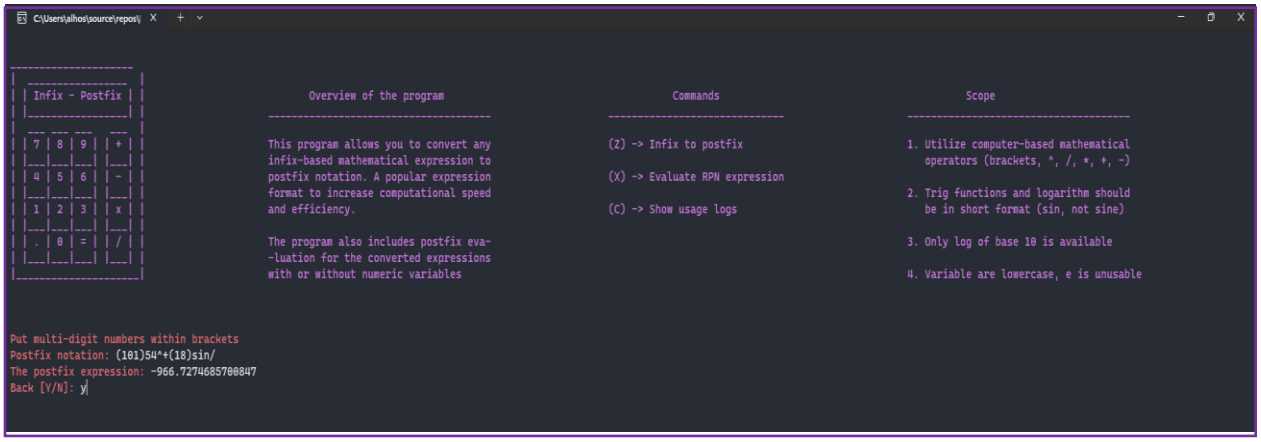
Expression used: (sin(2*4)(3 + cos(4))^ln(69)

Conversion: 24*sin34cos+(69)ln^*

Evaluation: 36.61

Evaluation and conversion accuracy: **Perfect**

Postfix evaluation (option "x"):



Expression used: (101)54^*(18)sin/

Evaluation: -966.72

Evaluation accuracy: **Perfect**

Showing logs (option "c"):

References

Geeks For Geeks. (2022, July 1). *Why do we need Prefix and Postfix notations?* Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/why-do-we-need-prefix-and-postfix-notations/

"In the information age, the barriers [to entry into programming] just aren't there. The barriers are self imposed. If you want to set off and go develop some grand new thing, you don't need millions of dollars of capitalization. You need enough pizza and Diet Coke to stick in your refrigerator, a cheap PC to work on, and the dedication to go through with it. We slept on floors. We waded across rivers."

-John D. Carmack

Pioneer of 3D game development