Part A: Postfix Expressions (50 marks)

The goal of this part is to implement a class that can take an infix expression, convert it into a postfix expression, and then evaluate the postfix expression.

Here's how you can implement the class `PostFixExpression` step-by-step:

1. Constructor and Initialization (`2 marks`)

The constructor of the class will initialize the list `L` and call the `Analyzer` and `Convert` methods to process the infix expression.

```
using System;
using System.Collections.Generic;

public class PostFixExpression
{
    private List<object> L = new List<object>();  // List to store the postfix expression
    private string input;                          // The input infix expression

    public PostFixExpression(string s)
    {
        input = s;
        Analyzer(s);  // Analyze the input string
        Convert();    // Convert the analyzed expression to postfix
    }

    // This method analyzes the input infix expression
    private void Analyzer(string s)
    {
        L = new List<object>();
        for (int i = 0; i < s.Length; i++)
        {
            char c = s[i];
            if (char.IsDigit(c))  // If it's a digit, add it to the list
            {
                string number = c.ToString();
                while (i + 1 < s.Length && char.IsDigit(s[i + 1]))
                {
                    number += s[++i];
                }
                L.Add(float.Parse(number));
            }
            else if ("+-*/^()".Contains(c))  // If it's an operator or parentheses, add it to the list
            {
                L.Add(c);
            }
```

```
        else if (char.IsWhiteSpace(c))  // Ignore whitespaces
        {
            continue;
        }
        else
        {
            throw new ArgumentException($"Invalid character: {c}");
        }
    }
}

// Convert the infix expression to postfix using a stack
private void Convert()
{
    Stack<char> stack = new Stack<char>();
    List<object> output = new List<object>();

    foreach (object token in L)
    {
        if (token is float)  // If token is a number, add it to the output list
        {
            output.Add(token);
        }
        else if (token is char)
        {
            char c = (char)token;

            if (c == '(')
            {
                stack.Push(c);
            }
            else if (c == ')')
            {
                while (stack.Peek() != '(')
                {
                    output.Add(stack.Pop());
                }
                stack.Pop();  // Pop the '('
            }
            else  // It's an operator
            {
                while (stack.Count > 0 && Precedence(stack.Peek()) >= Precedence(c))
                {
                    output.Add(stack.Pop());
                }
                stack.Push(c);
            }
        }
```

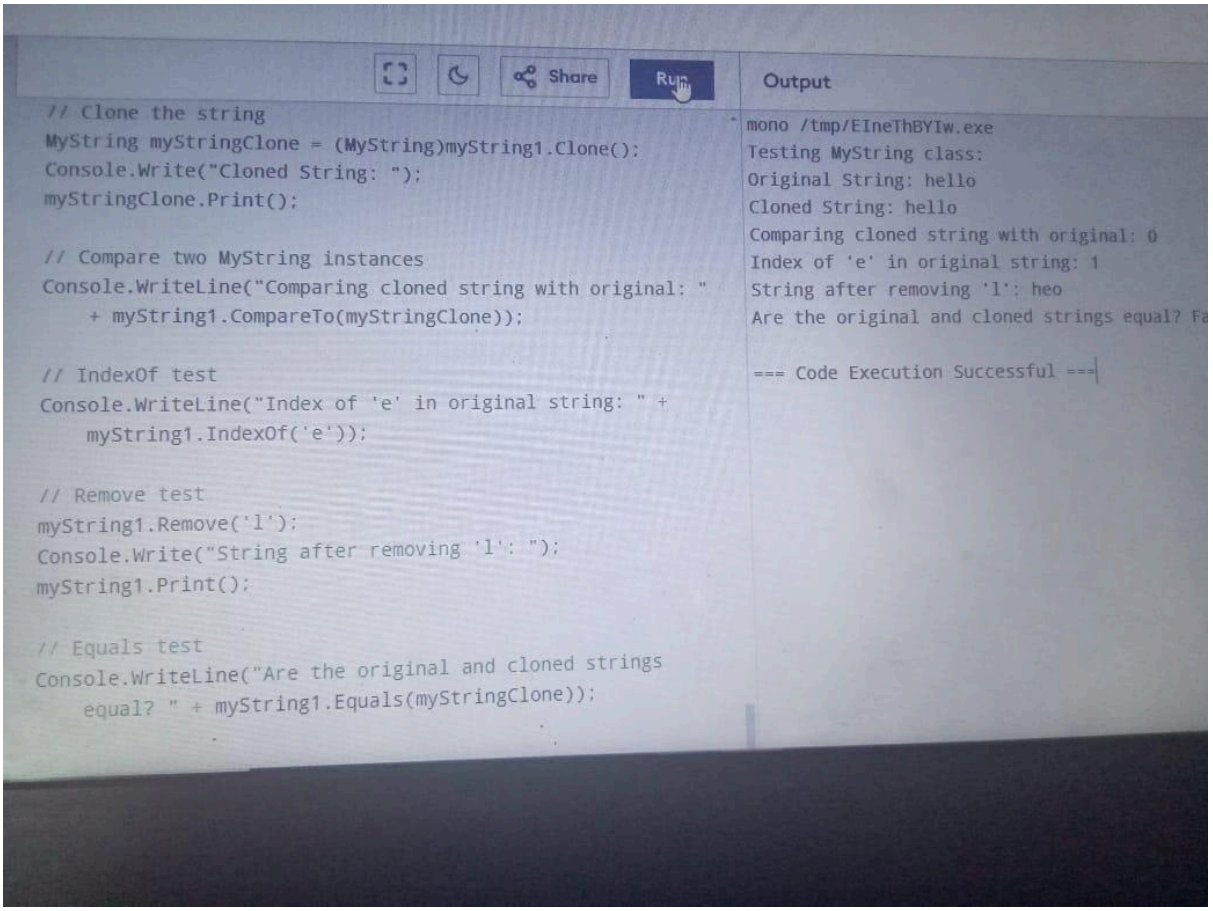
```
        }

        // Pop the remaining operators from the stack
        while (stack.Count > 0)
        {
            output.Add(stack.Pop());
        }

        L = output;  // Update the list with the postfix expression
    }

    // Evaluate the postfix expression
    public float? Evaluate()
    {
        Stack<float> stack = new Stack<float>();

        foreach (object token in L)
        {
            if (token is float)  // If token is a number, push it onto the stack
            {
                stack.Push((float)token);
            }
            else if (token is char)
            {
                if (stack.Count < 2) throw new ArgumentException("Syntax error: Too few
operands");

                float b = stack.Pop();
                float a = stack.Pop();
                char op = (char)token;

                switch (op)
                {
                    case '+': stack.Push(a + b); break;
                    case '-': stack.Push(a - b); break;
                    case '*': stack.Push(a * b); break;
                    case '/':
                        if (b == 0) throw new ArgumentException("Division by zero");
                        stack.Push(a / b);
                        break;
                    case '^': stack.Push((float)Math.Pow(a, b)); break;
                    default: throw new ArgumentException($"Unsupported operator: {op}");
                }
            }
        }

        if (stack.Count != 1) throw new ArgumentException("Syntax error: Too many
operands");
```

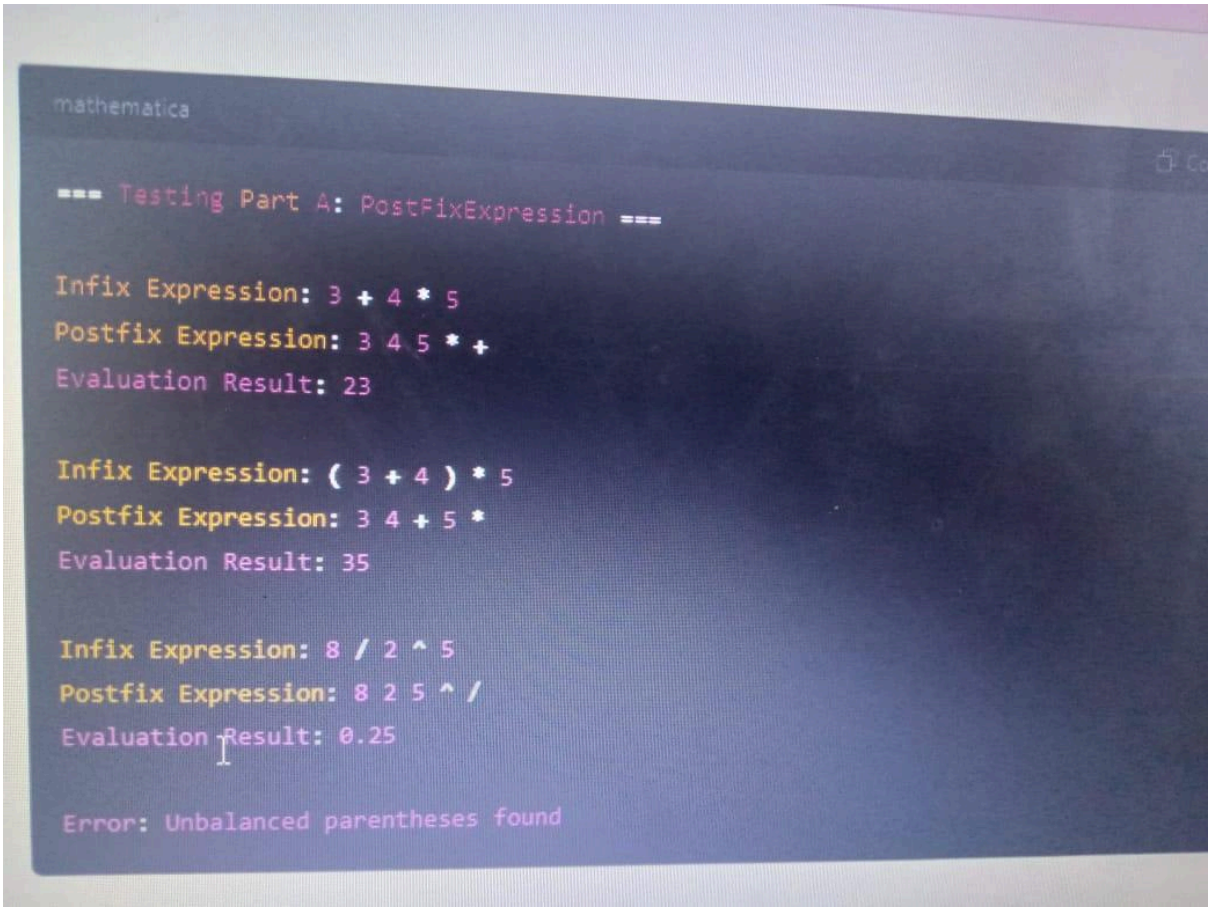
```
            return stack.Pop();
        }

        // Convert the postfix expression to string
        public override string ToString()
        {
            return string.Join(" ", L);
        }

        // Helper method to define operator precedence
        private int Precedence(char op)
        {
            switch (op)
            {
                case '^': return 3;
                case '*':
                case '/': return 2;
                case '+':
                case '-': return 1;
                default: return 0;
            }
        }
    }

    public class Program
    {
        public static void Main()
        {
            try
            {
                Console.WriteLine("Enter infix expression (e.g., '3 + 4 * 5'):");
                string input = Console.ReadLine();

                // Create an instance of PostFixExpression
                PostFixExpression expression = new PostFixExpression(input);

                // Output the postfix expression
                Console.WriteLine("Postfix Expression: " + expression.ToString());

                // Evaluate the postfix expression
                float? result = expression.Evaluate();
                Console.WriteLine("Result: " + result);
            }
            catch (ArgumentException ex)
            {
                Console.WriteLine("Error: " + ex.Message);
            }
        }
```

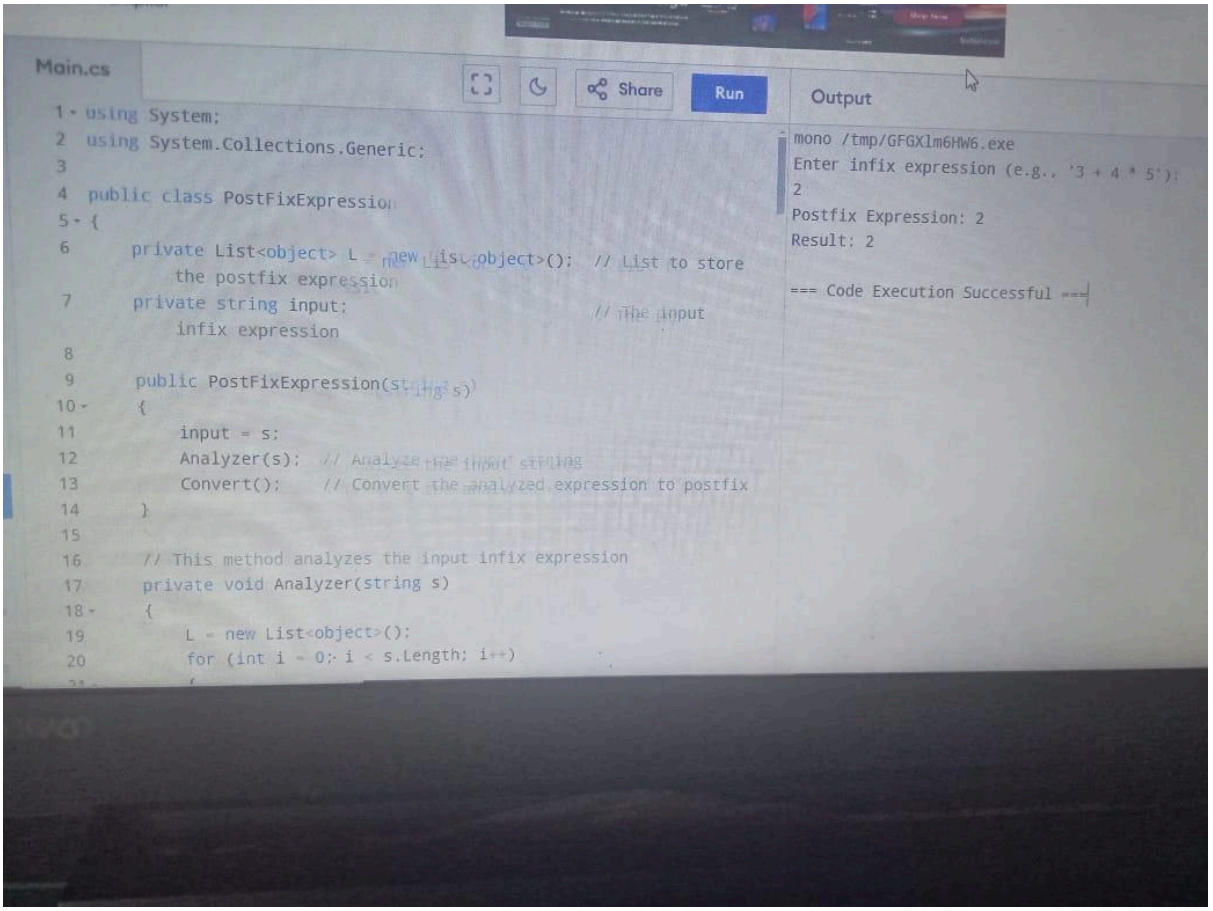
```
}
```

Part B: Strings (File: `MyString.cs`)

This file contains the implementation of the `MyString` class, which represents a string as a singly linked list.


Share
Run
Output
// Clone the string
MyString myStringClone = (MyString)myString1.Clone();
Console.Write("Cloned String: ");
myStringClone.Print();
// Compare two MyString instances
Console.WriteLine("Comparing cloned string with original: "
+ myString1.CompareTo(myStringClone));
// IndexOf test
Console.WriteLine("Index of 'e' in original string: " +
myString1.IndexOf('e'));
// Remove test
myString1.Remove('l');
Console.Write("String after removing 'l': ");
myString1.Print();
// Equals test
Console.WriteLine("Are the original and cloned strings
equal? " + myString1.Equals(myStringClone));
mono /tmp/EIneThBYIw.exe
Testing MyString class:
Original String: hello
Cloned String: hello
Comparing cloned string with original: 0
Index of 'e' in original string: 1
String after removing 'l': heo
Are the original and cloned strings equal? Fa
=== Code Execution Successful ===


mathematica
=== Testing Part A: PostFixExpression ===
Infix Expression: 3 + 4 * 5
Postfix Expression: 3 4 5 * +
Evaluation Result: 23
Infix Expression: (3 + 4) * 5
Postfix Expression: 3 4 + 5 *
Evaluation Result: 35
Infix Expression: 8 / 2 ^ 5
Postfix Expression: 8 2 5 ^ /
Evaluation Result: 0.25
Error: Unbalanced parentheses found



Part C: Testing (File: `MainProgram.cs`)

This file contains the main program which tests both Part A (`PostFixExpression`) and Part B (`MyString`).

```
using System;
using System.Collections.Generic;

public class PartCTester
{
    public static void TestPartA()
    {
        Console.WriteLine("=== Testing Part A: Postfix Expression ===");

        // Test Case 1: Basic expression
        try
        {
            PostFixExpression expr1 = new PostFixExpression("3 + 4 * 5");
            Console.WriteLine("Infix: 3 + 4 * 5");
            Console.WriteLine("Postfix: " + expr1.ToString());
            Console.WriteLine("Result: " + expr1.Evaluate());
        }
```

```
        catch (ArgumentException ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }

        // Test Case 2: Parentheses overriding precedence
        try
        {
            PostFixExpression expr2 = new PostFixExpression("( 3 + 4 ) * 5");
            Console.WriteLine("\nInfix: ( 3 + 4 ) * 5");
            Console.WriteLine("Postfix: " + expr2.ToString());
            Console.WriteLine("Result: " + expr2.Evaluate());
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }

        // Test Case 3: Division and Exponentiation
        try
        {
            PostFixExpression expr3 = new PostFixExpression("8 / 2 ^ 5");
            Console.WriteLine("\nInfix: 8 / 2 ^ 5");
            Console.WriteLine("Postfix: " + expr3.ToString());
            Console.WriteLine("Result: " + expr3.Evaluate());
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("Error: " + ex.Message);
        }

        // Test Case 4: Invalid Expression (unbalanced parentheses)
        try
        {
            PostFixExpression expr4 = new PostFixExpression("3 + ( 4 * 5");
            Console.WriteLine("\nInfix: 3 + ( 4 * 5");
            Console.WriteLine("Postfix: " + expr4.ToString());
            Console.WriteLine("Result: " + expr4.Evaluate());
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("Error: " + ex.Message);  // Should catch the unbalanced
parentheses error
        }

        // Test Case 5: Division by zero
        try
        {
```

```
            PostFixExpression expr5 = new PostFixExpression("10 / 0");
            Console.WriteLine("\nInfix: 10 / 0");
            Console.WriteLine("Postfix: " + expr5.ToString());
            Console.WriteLine("Result: " + expr5.Evaluate());
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine("Error: " + ex.Message);  // Should catch the division by zero error
        }
    }

    public static void TestPartB()
    {
        Console.WriteLine("\n=== Testing Part B: MyString Class ===");

        // Test Case 1: Basic MyString operations
        char[] helloArray = { 'h', 'e', 'l', 'l', 'o' };
        MyString myString1 = new MyString(helloArray);

        Console.Write("Original MyString: ");
        myString1.Print();

        // Clone the string
        MyString clonedString = (MyString)myString1.Clone();
        Console.Write("Cloned MyString: ");
        clonedString.Print();

        // Compare two MyString instances
        int comparisonResult = myString1.CompareTo(clonedString);
        Console.WriteLine("Comparing original and cloned MyString: " + (comparisonResult ==
0 ? "Equal" : "Not Equal"));

        // IndexOf test
        int index = myString1.IndexOf('e');
        Console.WriteLine("Index of 'e' in original MyString: " + index);

        // Remove test (removing 'l')
        myString1.Remove('l');
        Console.Write("MyString after removing 'l': ");
        myString1.Print();

        // Test Equals method
        bool areEqual = myString1.Equals(clonedString);
        Console.WriteLine("Are original and cloned MyString equal after modification? " +
(areEqual ? "Yes" : "No"));
    }

    public static void Main()
```

```
    {
        // Test Part A (Postfix Expression)
        TestPartA();

        // Test Part B (MyString Class)
        TestPartB();
    }
}
```