COMSATS UNIVERSITY ISLAMABAD

ATTOCK CAMPUS

DEPARTMENT OF COMPUTER SCIENCE

LAB MID

Submitted by:-

Muhammad Usman

Reg No:-

SP22-BCS-036

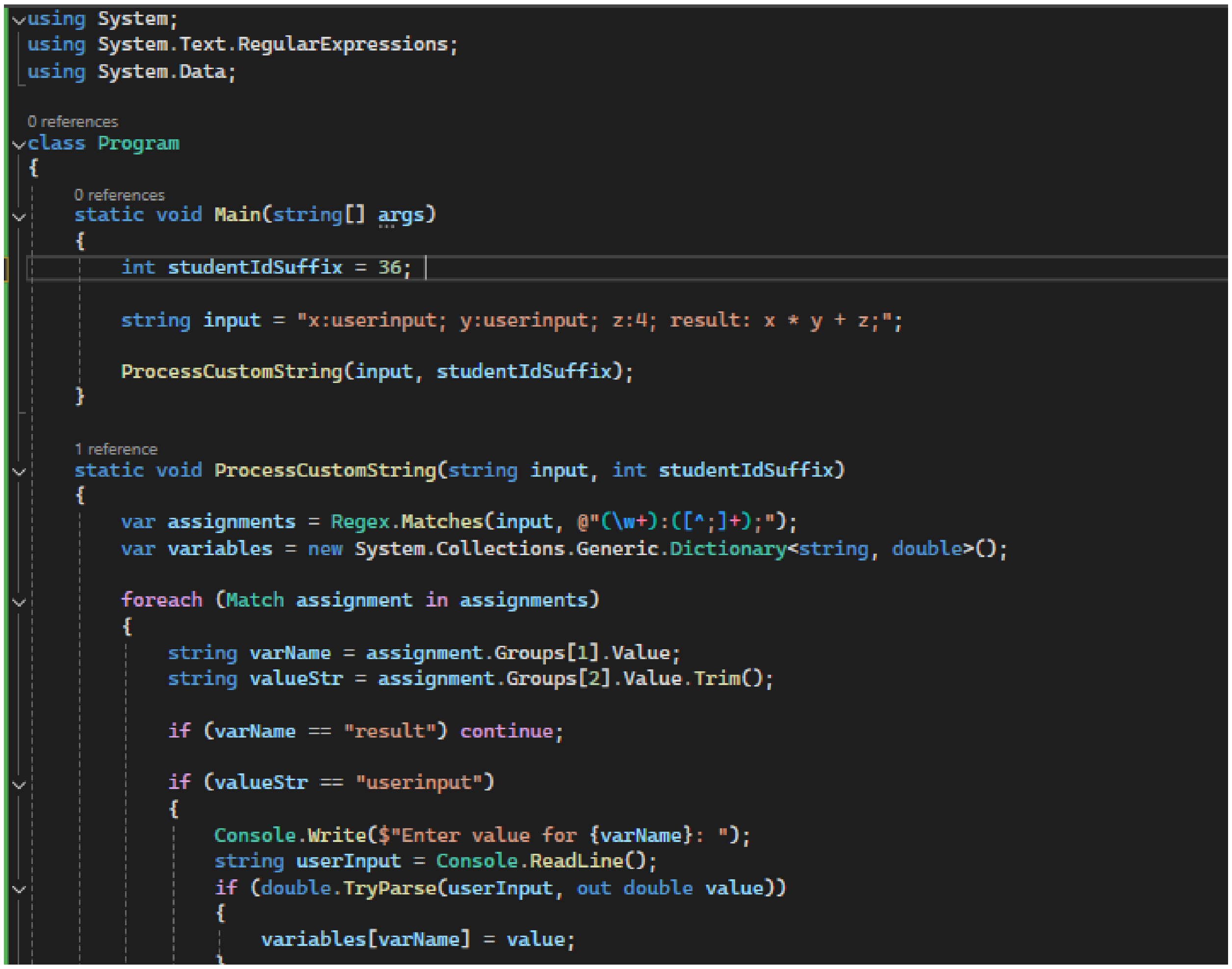
Submitted to:-

Syed Bilal Haider

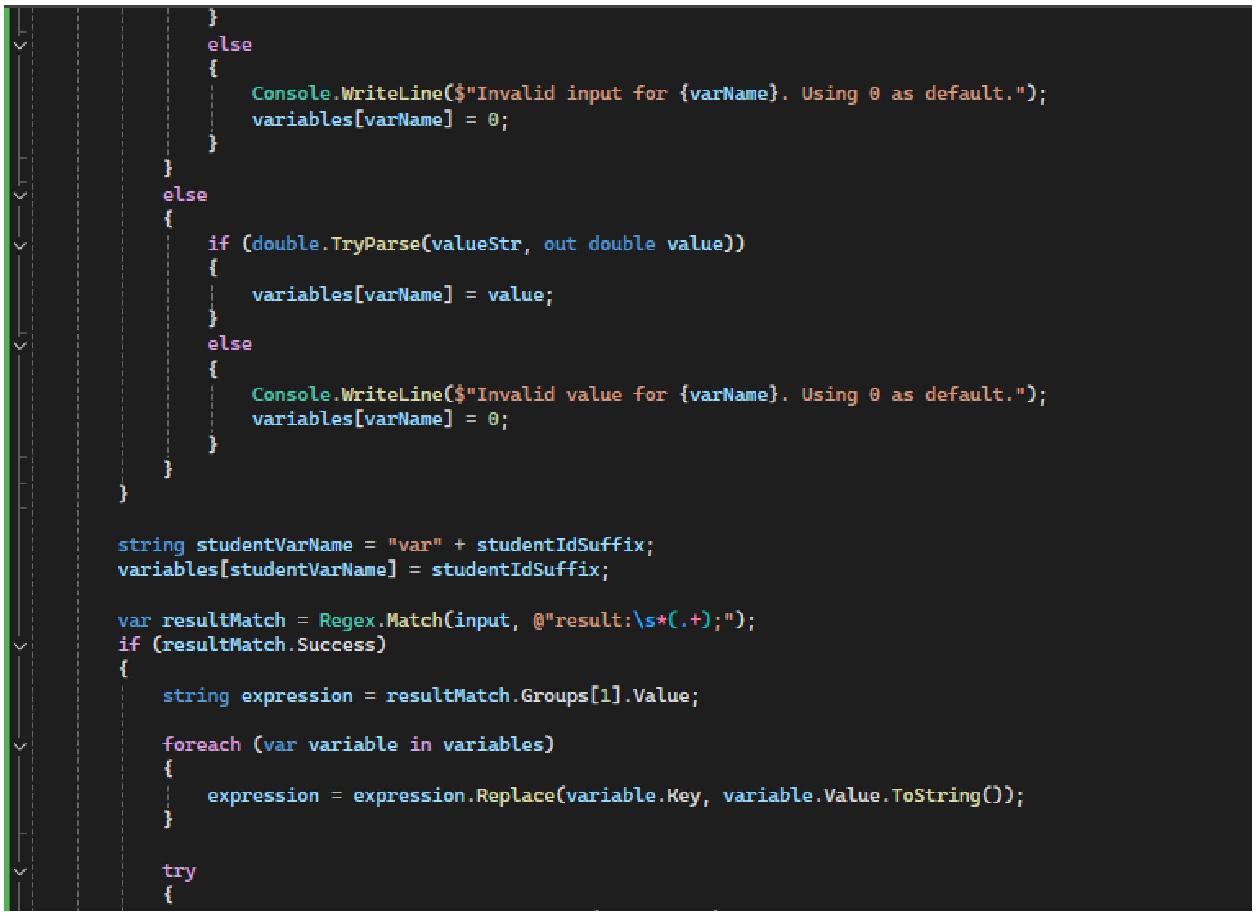
Subject:-

Compiler construction

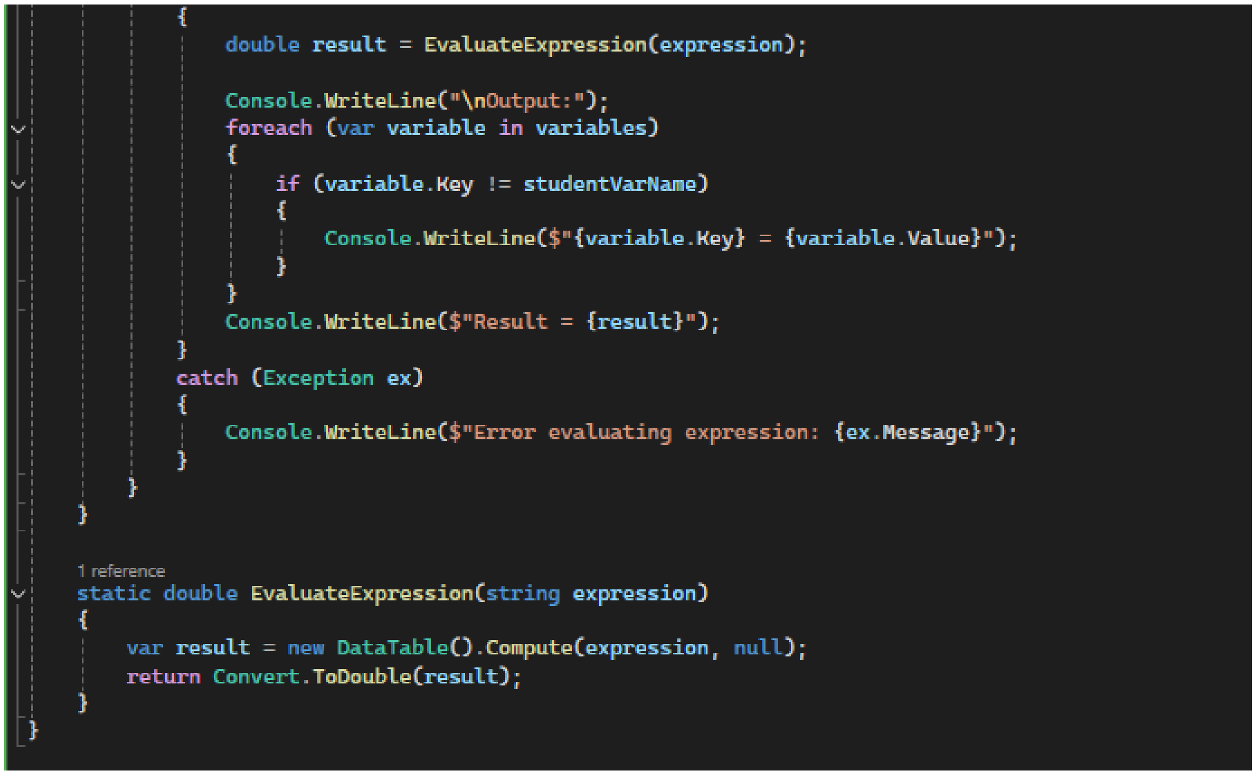
Q1:



```
using System;
using System.Text.RegularExpressions;
using System.Data;

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        int studentIdSuffix = 36;

        string input = "x:userinput; y:userinput; z:4; result: x * y + z;";

        ProcessCustomString(input, studentIdSuffix);
    }

    1 reference
    static void ProcessCustomString(string input, int studentIdSuffix)
    {
        var assignments = Regex.Matches(input, @"(\w+):([^;]+);");
        var variables = new System.Collections.Generic.Dictionary<string, double>();

        foreach (Match assignment in assignments)
        {
            string varName = assignment.Groups[1].Value;
            string valueStr = assignment.Groups[2].Value.Trim();

            if (varName == "result") continue;

            if (valueStr == "userinput")
            {
                Console.Write($"Enter value for {varName}: ");
                string userInput = Console.ReadLine();
                if (double.TryParse(userInput, out double value))
                {
                    variables[varName] = value;
```

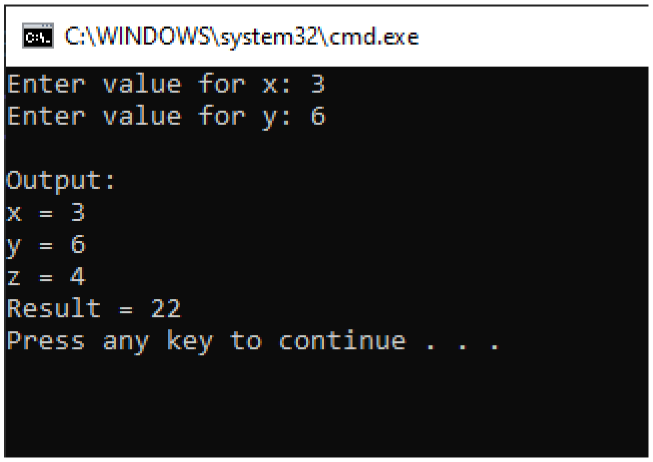


```
                }
                else
                {
                    Console.WriteLine($"Invalid input for {varName}. Using 0 as default.");
                    variables[varName] = 0;
                }
            }
            else
            {
                if (double.TryParse(valueStr, out double value))
                {
                    variables[varName] = value;
                }
                else
                {
                    Console.WriteLine($"Invalid value for {varName}. Using 0 as default.");
                    variables[varName] = 0;
                }
            }
        }

        string studentVarName = "var" + studentIdSuffix;
        variables[studentVarName] = studentIdSuffix;

        var resultMatch = Regex.Match(input, @"result:\s*(.+);");
        if (resultMatch.Success)
        {
            string expression = resultMatch.Groups[1].Value;

            foreach (var variable in variables)
            {
                expression = expression.Replace(variable.Key, variable.Value.ToString());
            }

            try
            {
```



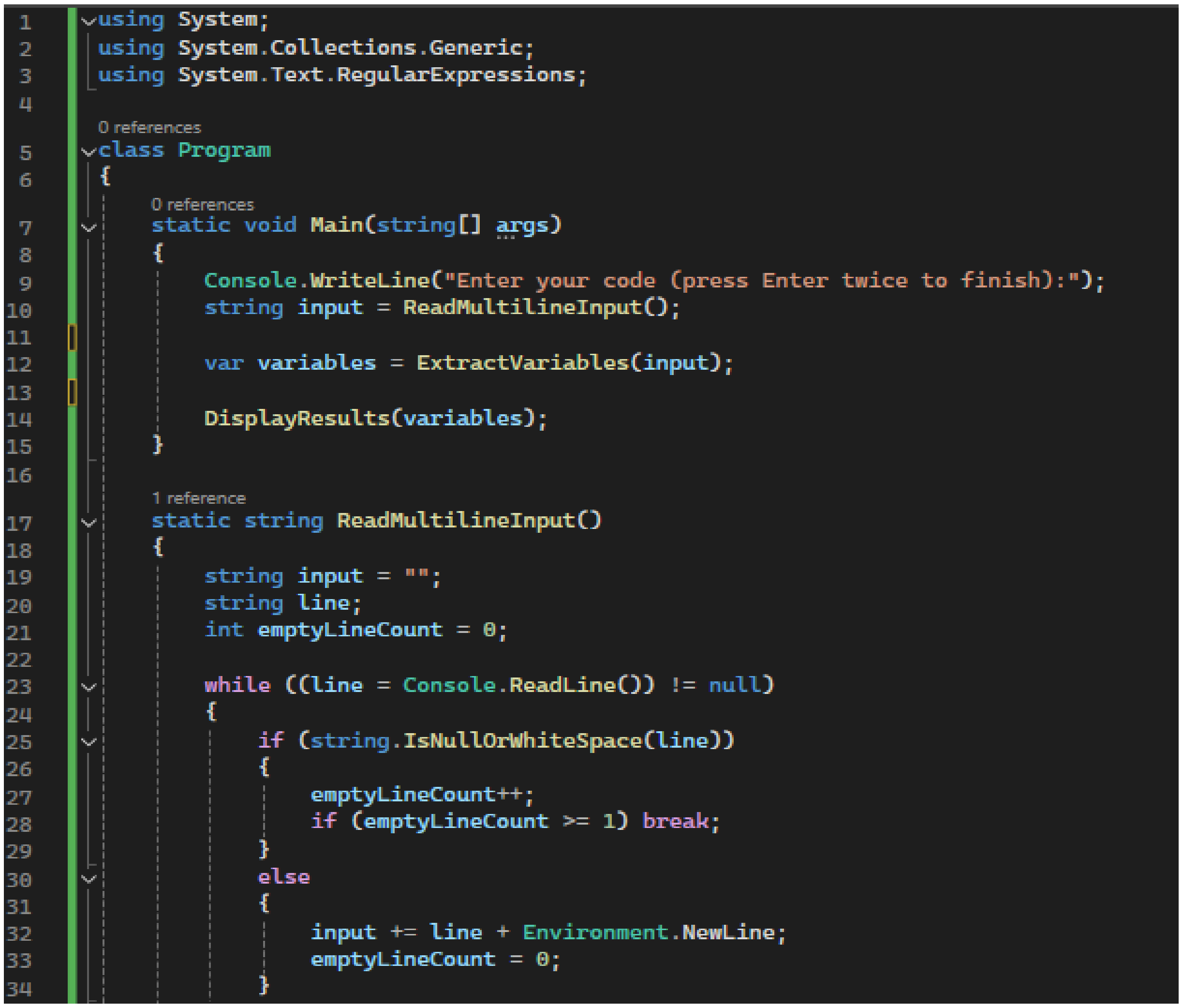
```
            {
                double result = EvaluateExpression(expression);

                Console.WriteLine("\nOutput:");
                foreach (var variable in variables)
                {
                    if (variable.Key != studentVarName)
                    {
                        Console.WriteLine($"{variable.Key} = {variable.Value}");
                    }
                }
                Console.WriteLine($"Result = {result}");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error evaluating expression: {ex.Message}");
            }
        }
    }

    1 reference
    static double EvaluateExpression(string expression)
    {
        var result = new DataTable().Compute(expression, null);
        return Convert.ToDouble(result);
    }
}
```

OUTPUT

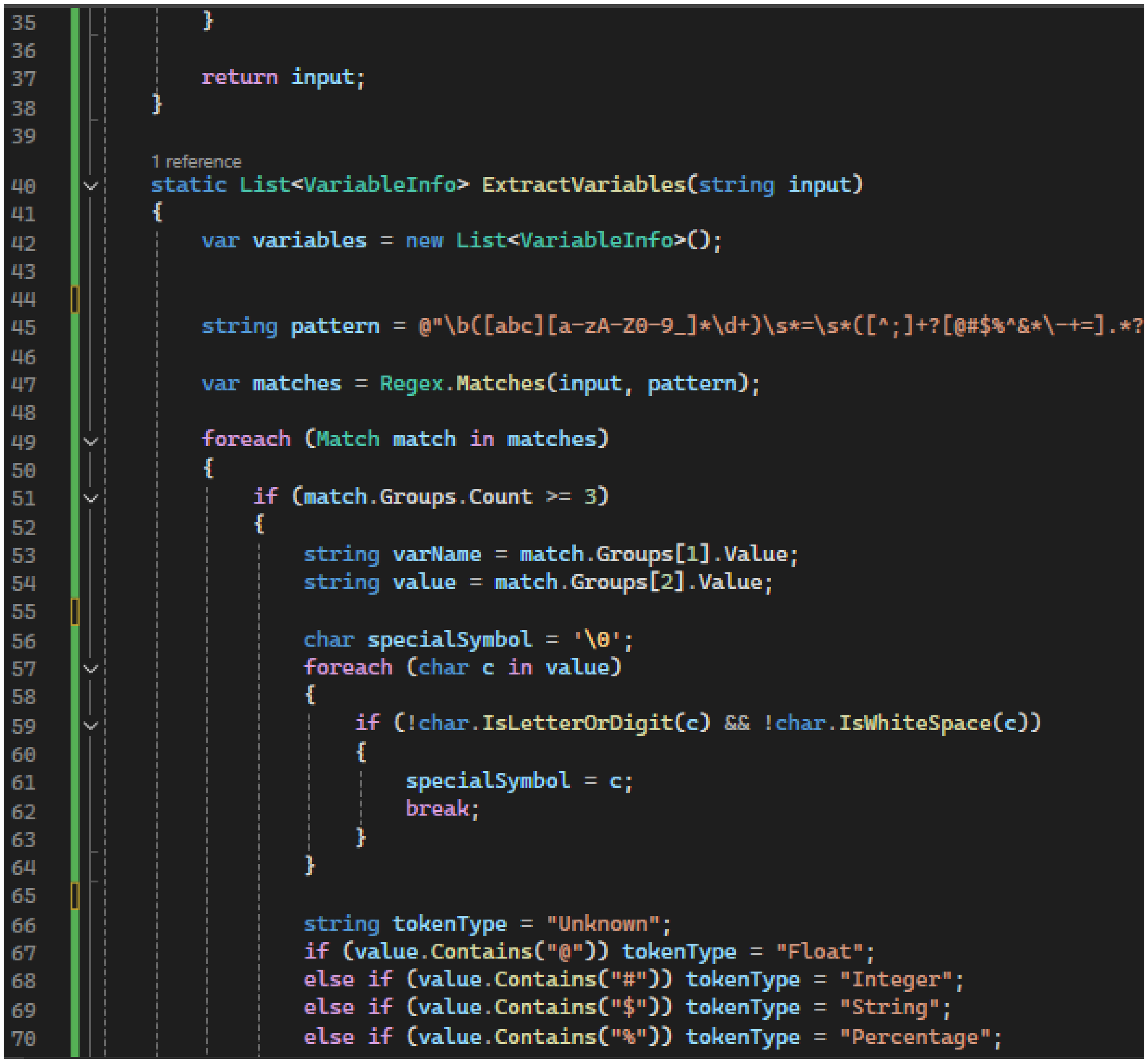

C:\WINDOWS\system32\cmd.exe

```
Enter value for x: 3
Enter value for y: 6

Output:
x = 3
y = 6
z = 4
Result = 22
Press any key to continue . . .
```

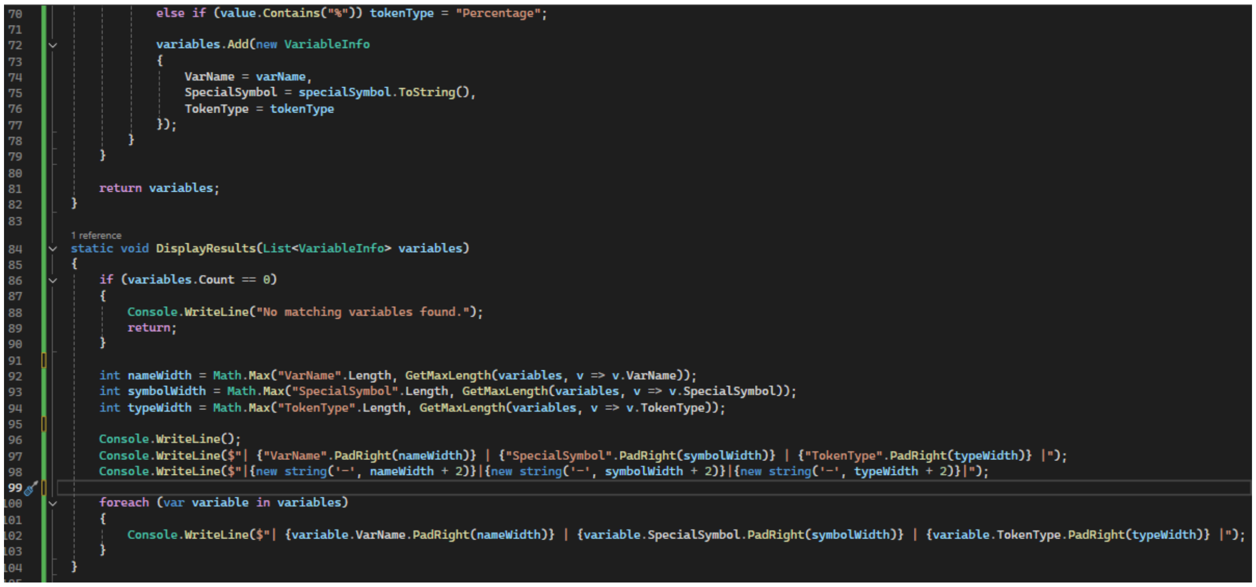
QUESTION NO 2



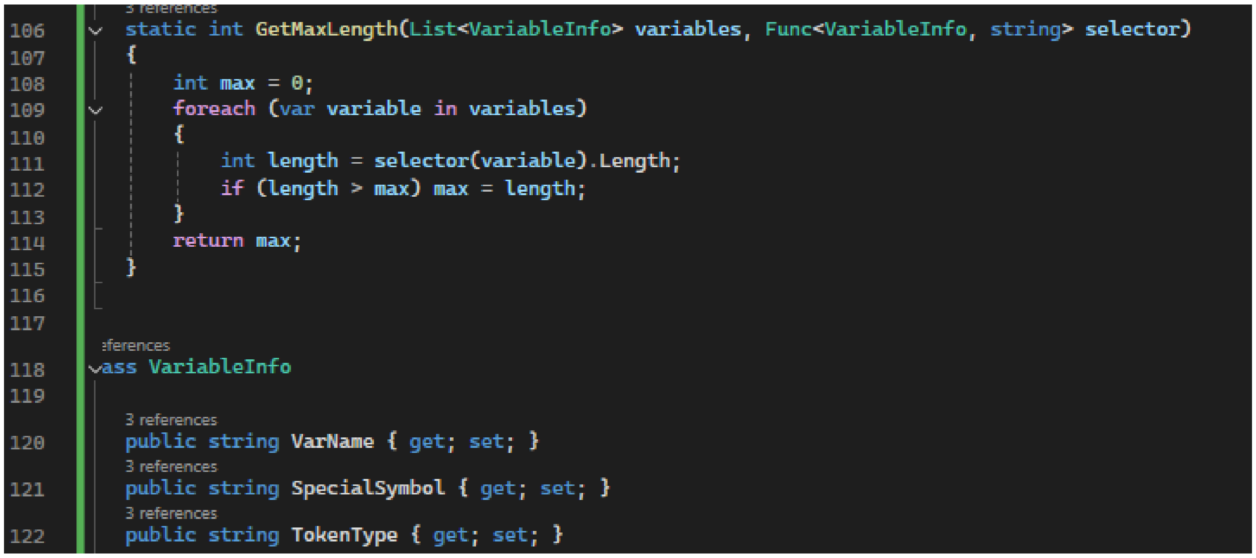
```csharp
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("Enter your code (press Enter twice to finish):");
        string input = ReadMultilineInput();

        var variables = ExtractVariables(input);

        DisplayResults(variables);
    }

    1 reference
    static string ReadMultilineInput()
    {
        string input = "";
        string line;
        int emptyLineCount = 0;

        while ((line = Console.ReadLine()) != null)
        {
            if (string.IsNullOrWhiteSpace(line))
            {
                emptyLineCount++;
                if (emptyLineCount >= 1) break;
            }
            else
            {
                input += line + Environment.NewLine;
                emptyLineCount = 0;
            }
```



```
    }

    return input;
}

1 reference
static List<VariableInfo> ExtractVariables(string input)
{
    var variables = new List<VariableInfo>();


    string pattern = @"\b([abc][a-zA-Z0-9_]*\d+)\s*=\s*([^;]+?[@#$%^&*\-+=].*?

    var matches = Regex.Matches(input, pattern);

    foreach (Match match in matches)
    {
        if (match.Groups.Count >= 3)
        {
            string varName = match.Groups[1].Value;
            string value = match.Groups[2].Value;

            char specialSymbol = '\0';
            foreach (char c in value)
            {
                if (!char.IsLetterOrDigit(c) && !char.IsWhiteSpace(c))
                {
                    specialSymbol = c;
                    break;
                }
            }

            string tokenType = "Unknown";
            if (value.Contains("@")) tokenType = "Float";
            else if (value.Contains("#")) tokenType = "Integer";
            else if (value.Contains("$")) tokenType = "String";
            else if (value.Contains("%")) tokenType = "Percentage";
```

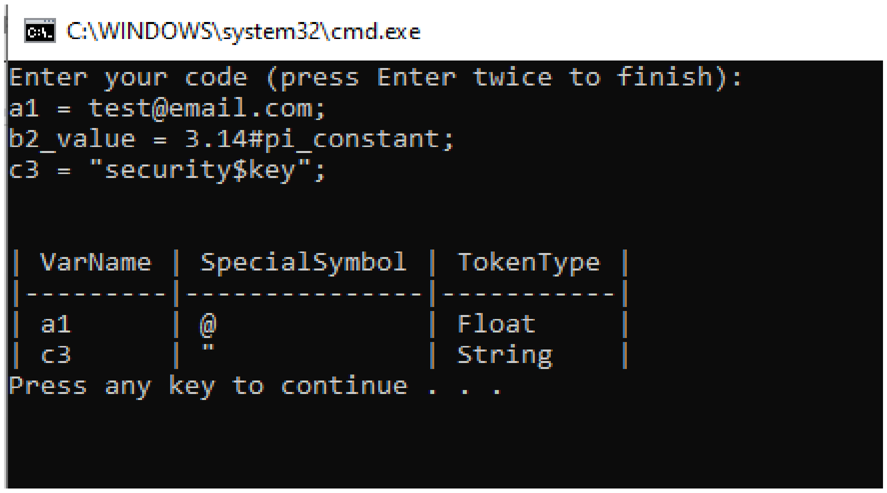


```
                else if (value.Contains("%")) tokenType = "Percentage";

                variables.Add(new VariableInfo
                {
                    VarName = varName,
                    SpecialSymbol = specialSymbol.ToString(),
                    TokenType = tokenType
                });
            }
        }

        return variables;
    }

    1 reference
    static void DisplayResults(List<VariableInfo> variables)
    {
        if (variables.Count == 0)
        {
            Console.WriteLine("No matching variables found.");
            return;
        }

        int nameWidth = Math.Max("VarName".Length, GetMaxLength(variables, v => v.VarName));
        int symbolWidth = Math.Max("SpecialSymbol".Length, GetMaxLength(variables, v => v.SpecialSymbol));
        int typeWidth = Math.Max("TokenType".Length, GetMaxLength(variables, v => v.TokenType));

        Console.WriteLine();
        Console.WriteLine($"| {"VarName".PadRight(nameWidth)} | {"SpecialSymbol".PadRight(symbolWidth)} | {"TokenType".PadRight(typeWidth)} |");
        Console.WriteLine($"|{new string('-', nameWidth + 2)}|{new string('-', symbolWidth + 2)}|{new string('-', typeWidth + 2)}|");

        foreach (var variable in variables)
        {
            Console.WriteLine($"| {variable.VarName.PadRight(nameWidth)} | {variable.SpecialSymbol.PadRight(symbolWidth)} | {variable.TokenType.PadRight(typeWidth)} |");
        }
    }
```



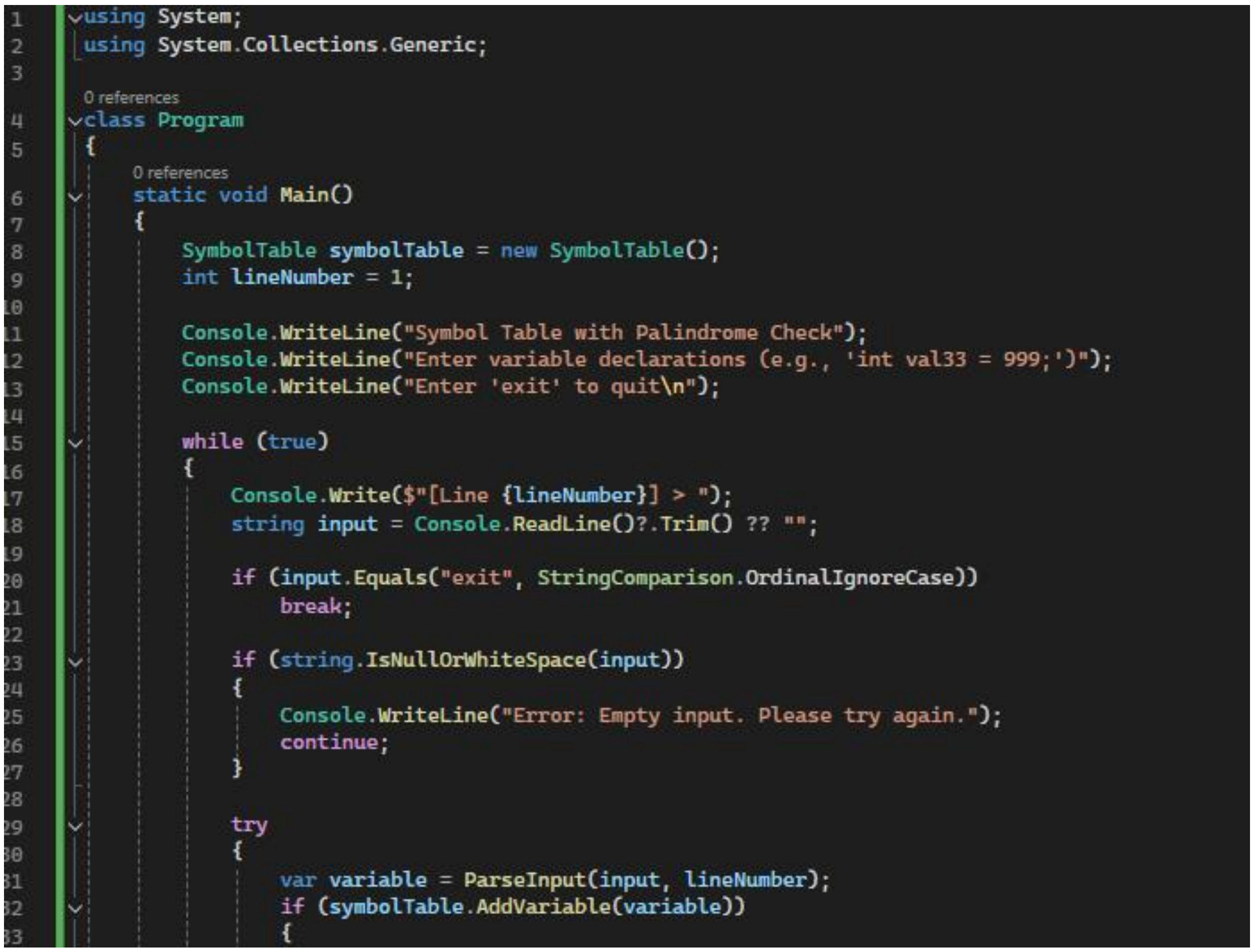
```
    3 references
    static int GetMaxLength(List<VariableInfo> variables, Func<VariableInfo, string> selector)
    {
        int max = 0;
        foreach (var variable in variables)
        {
            int length = selector(variable).Length;
            if (length > max) max = length;
        }
        return max;
    }


eferences
ass VariableInfo

    3 references
    public string VarName { get; set; }
    3 references
    public string SpecialSymbol { get; set; }
    3 references
    public string TokenType { get; set; }
```

OUTPUT

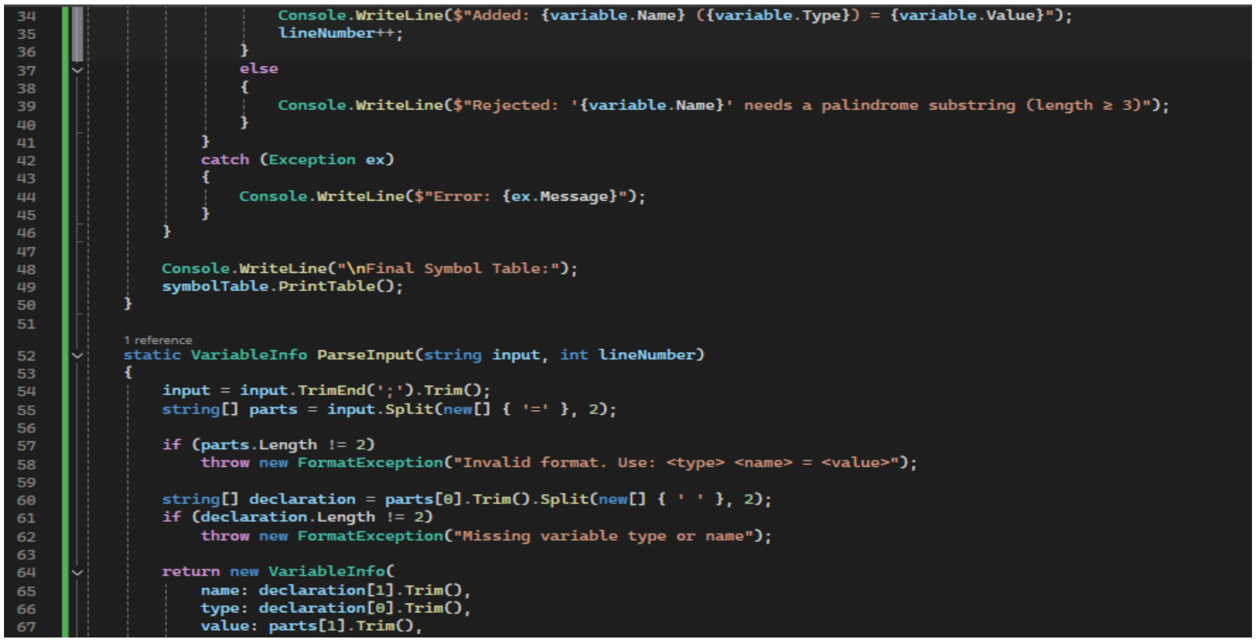


```
C:\WINDOWS\system32\cmd.exe
Enter your code (press Enter twice to finish):
a1 = test@email.com;
b2_value = 3.14#pi_constant;
c3 = "security$key";


| VarName | SpecialSymbol | TokenType |
|---------|---------------|-----------|
| a1      | @             | Float     |
| c3      | "             | String    |
Press any key to continue . . .
```

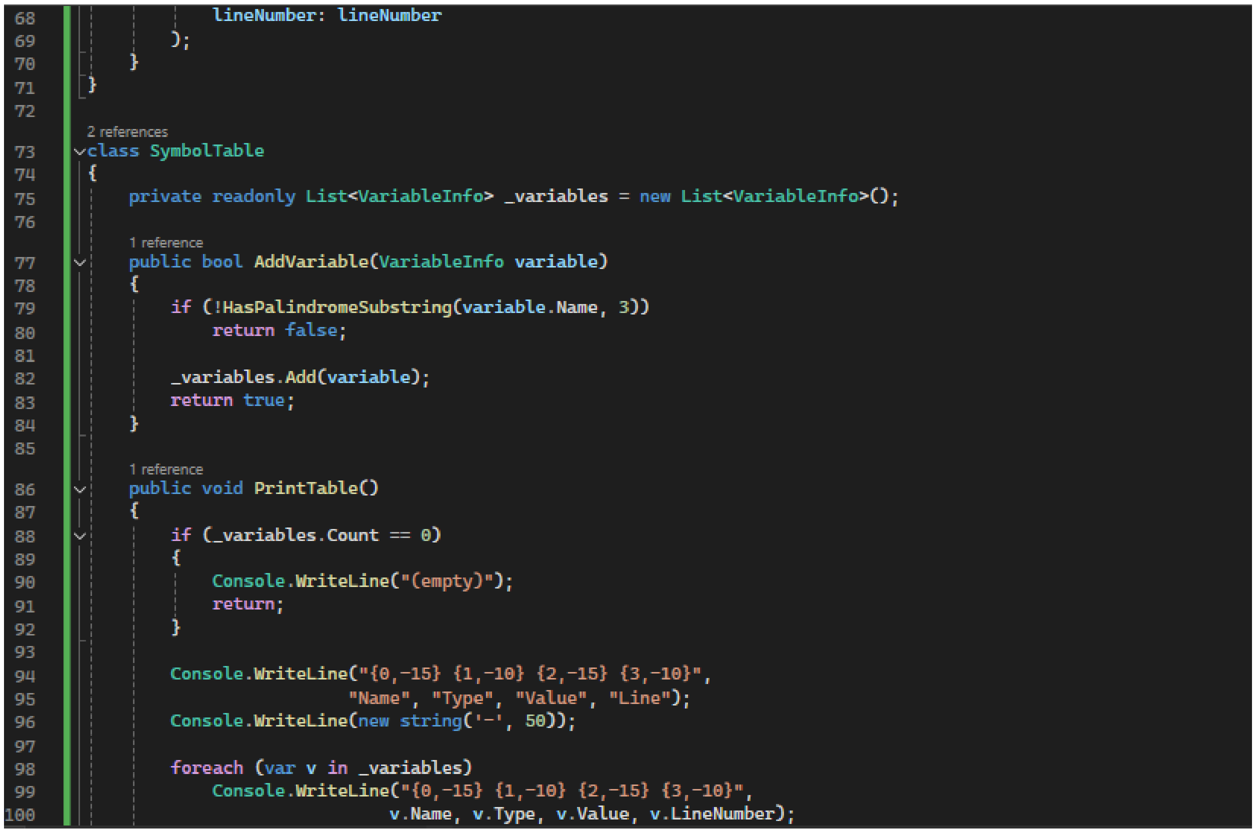
QUESTION NO 3



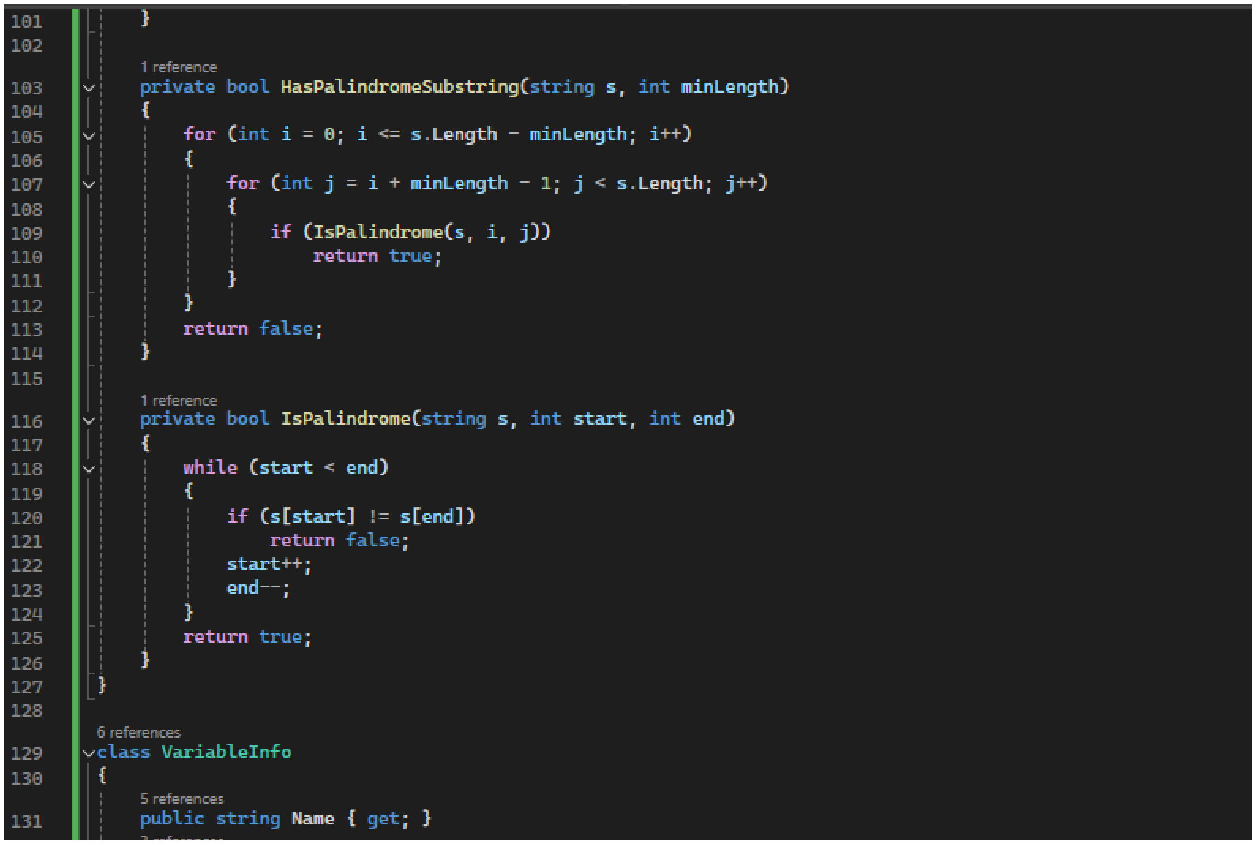
```csharp
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        SymbolTable symbolTable = new SymbolTable();
        int lineNumber = 1;

        Console.WriteLine("Symbol Table with Palindrome Check");
        Console.WriteLine("Enter variable declarations (e.g., 'int val33 = 999;')");
        Console.WriteLine("Enter 'exit' to quit\n");

        while (true)
        {
            Console.Write($"[Line {lineNumber}] > ");
            string input = Console.ReadLine()?.Trim() ?? "";

            if (input.Equals("exit", StringComparison.OrdinalIgnoreCase))
                break;

            if (string.IsNullOrWhiteSpace(input))
            {
                Console.WriteLine("Error: Empty input. Please try again.");
                continue;
            }

            try
            {
                var variable = ParseInput(input, lineNumber);
                if (symbolTable.AddVariable(variable))
                {
```



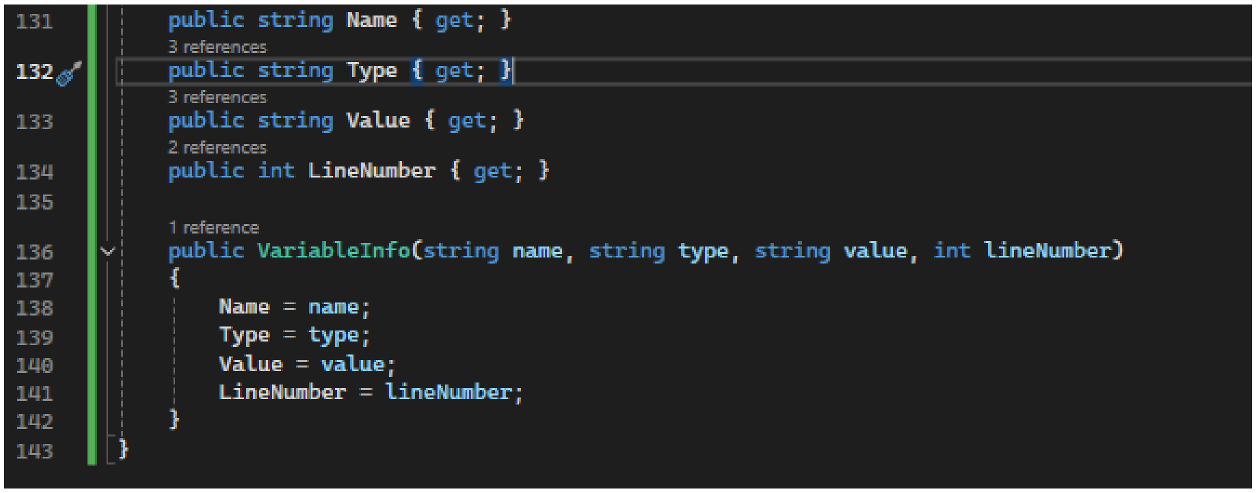
```csharp
                    Console.WriteLine($"Added: {variable.Name} ({variable.Type}) = {variable.Value}");
                    lineNumber++;
                }
                else
                {
                    Console.WriteLine($"Rejected: '{variable.Name}' needs a palindrome substring (length ≥ 3)");
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error: {ex.Message}");
            }
        }

        Console.WriteLine("\nFinal Symbol Table:");
        symbolTable.PrintTable();
    }

    static VariableInfo ParseInput(string input, int lineNumber)
    {
        input = input.TrimEnd(';').Trim();
        string[] parts = input.Split(new[] { '=' }, 2);

        if (parts.Length != 2)
            throw new FormatException("Invalid format. Use: <type> <name> = <value>");

        string[] declaration = parts[0].Trim().Split(new[] { ' ' }, 2);
        if (declaration.Length != 2)
            throw new FormatException("Missing variable type or name");

        return new VariableInfo(
            name: declaration[1].Trim(),
            type: declaration[0].Trim(),
            value: parts[1].Trim(),
```



```
            lineNumber: lineNumber
        );
    }
}

class SymbolTable
{
    private readonly List<VariableInfo> _variables = new List<VariableInfo>();

    public bool AddVariable(VariableInfo variable)
    {
        if (!HasPalindromeSubstring(variable.Name, 3))
            return false;

        _variables.Add(variable);
        return true;
    }

    public void PrintTable()
    {
        if (_variables.Count == 0)
        {
            Console.WriteLine("(empty)");
            return;
        }

        Console.WriteLine("{0,-15} {1,-10} {2,-15} {3,-10}",
                         "Name", "Type", "Value", "Line");
        Console.WriteLine(new string('-', 50));

        foreach (var v in _variables)
            Console.WriteLine("{0,-15} {1,-10} {2,-15} {3,-10}",
                             v.Name, v.Type, v.Value, v.LineNumber);
```

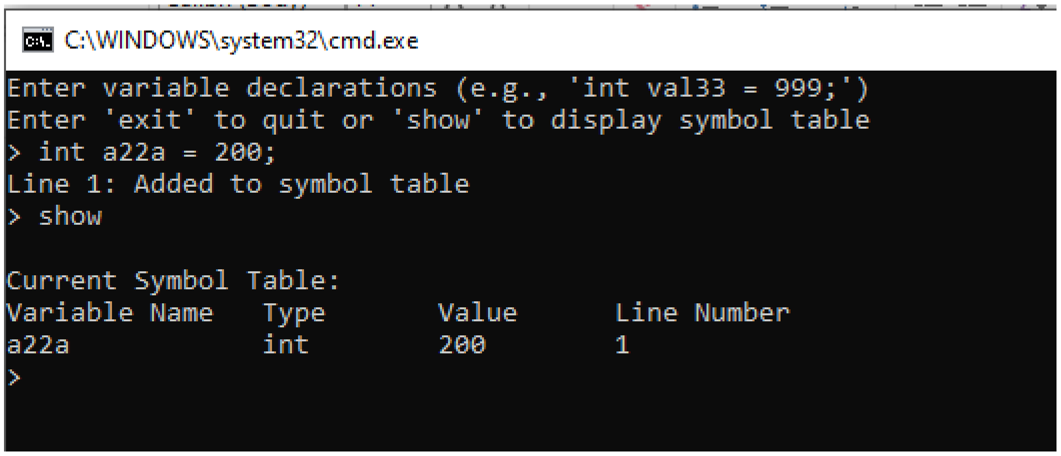


```
    }

    private bool HasPalindromeSubstring(string s, int minLength)
    {
        for (int i = 0; i <= s.Length - minLength; i++)
        {
            for (int j = i + minLength - 1; j < s.Length; j++)
            {
                if (IsPalindrome(s, i, j))
                    return true;
            }
        }
        return false;
    }

    private bool IsPalindrome(string s, int start, int end)
    {
        while (start < end)
        {
            if (s[start] != s[end])
                return false;
            start++;
            end--;
        }
        return true;
    }
}

class VariableInfo
{
    public string Name { get; }
```



```
    public string Name { get; }
    3 references
    public string Type { get; }
    3 references
    public string Value { get; }
    2 references
    public int LineNumber { get; }

    1 reference
    public VariableInfo(string name, string type, string value, int lineNumber)
    {
        Name = name;
        Type = type;
        Value = value;
        LineNumber = lineNumber;
    }
}
```

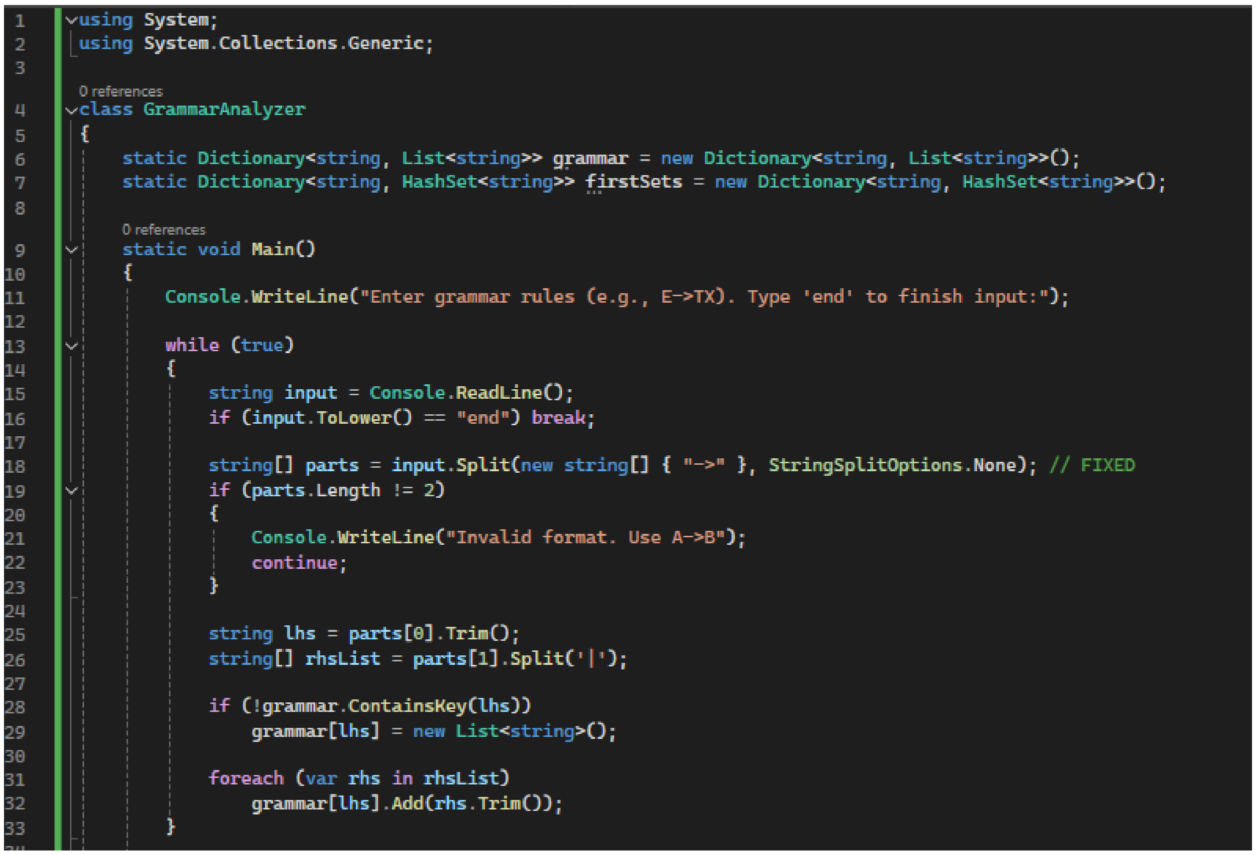
OUTPUT



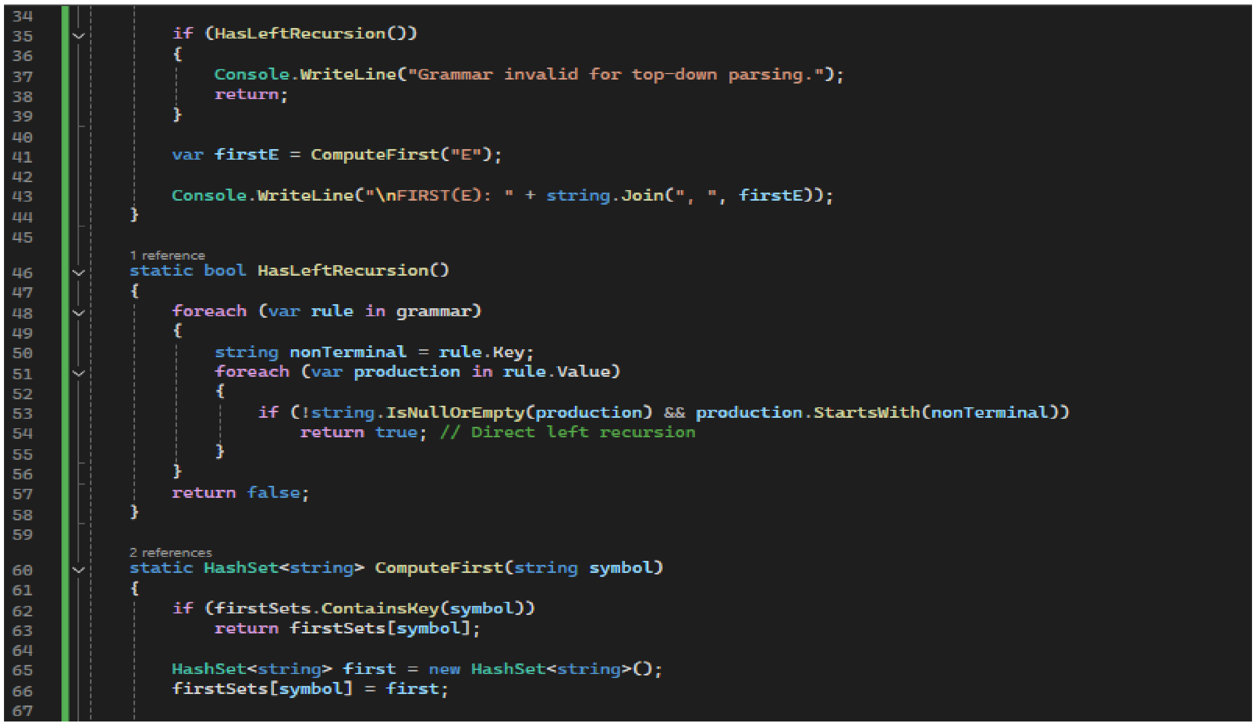
C:\WINDOWS\system32\cmd.exe

```
Enter variable declarations (e.g., 'int val33 = 999;')
Enter 'exit' to quit or 'show' to display symbol table
> int a22a = 200;
Line 1: Added to symbol table
> show

Current Symbol Table:
Variable Name    Type       Value      Line Number
a22a             int        200        1
>
```

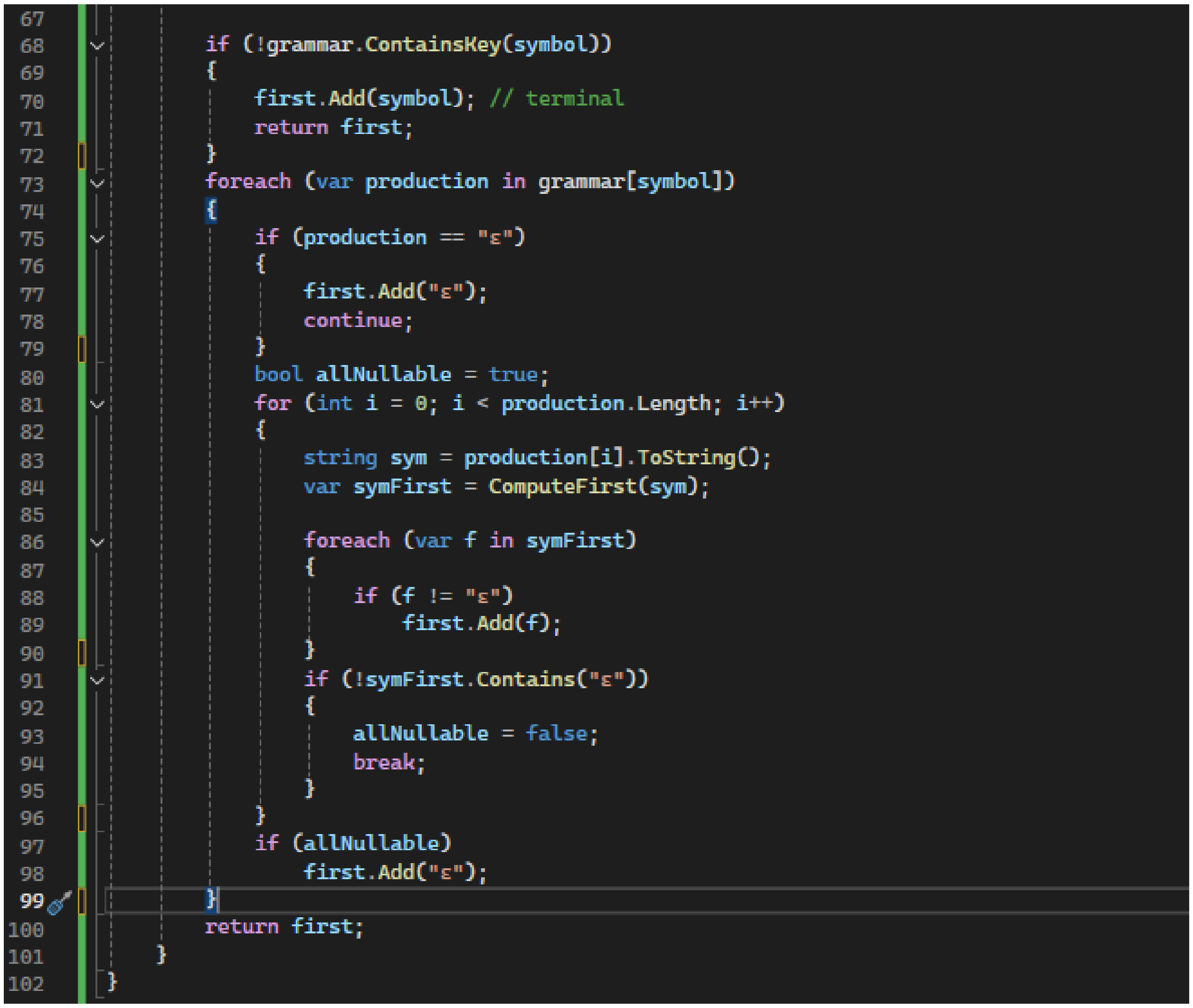
QUESTION 4



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

class GrammarAnalyzer
{
    static Dictionary<string, List<string>> grammar = new Dictionary<string, List<string>>();
    static Dictionary<string, HashSet<string>> firstSets = new Dictionary<string, HashSet<string>>();

    static void Main()
    {
        Console.WriteLine("Enter grammar rules (e.g., E->TX). Type 'end' to finish input:");

        while (true)
        {
            string input = Console.ReadLine();
            if (input.ToLower() == "end") break;

            string[] parts = input.Split(new string[] { "->" }, StringSplitOptions.None); // FIXED
            if (parts.Length != 2)
            {
                Console.WriteLine("Invalid format. Use A->B");
                continue;
            }

            string lhs = parts[0].Trim();
            string[] rhsList = parts[1].Split('|');

            if (!grammar.ContainsKey(lhs))
                grammar[lhs] = new List<string>();

            foreach (var rhs in rhsList)
                grammar[lhs].Add(rhs.Trim());
        }
```

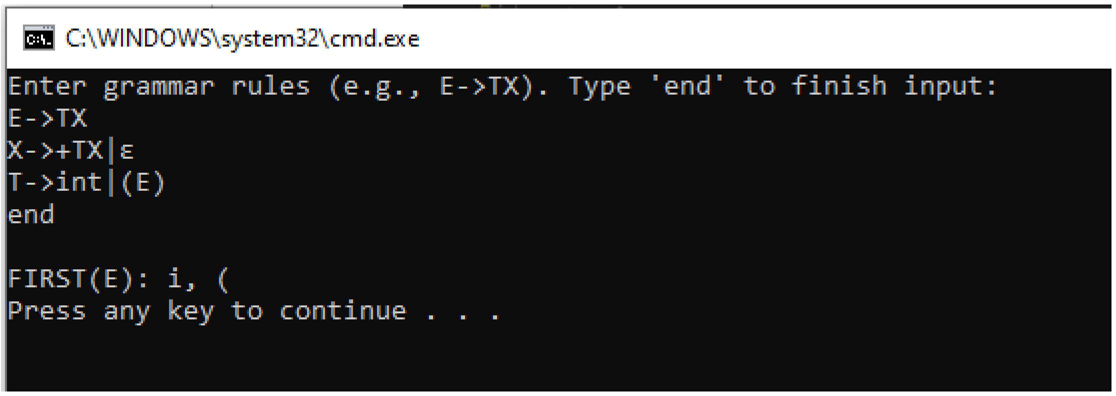


```csharp

        if (HasLeftRecursion())
        {
            Console.WriteLine("Grammar invalid for top-down parsing.");
            return;
        }

        var firstE = ComputeFirst("E");

        Console.WriteLine("\nFIRST(E): " + string.Join(", ", firstE));
    }

    static bool HasLeftRecursion()
    {
        foreach (var rule in grammar)
        {
            string nonTerminal = rule.Key;
            foreach (var production in rule.Value)
            {
                if (!string.IsNullOrEmpty(production) && production.StartsWith(nonTerminal))
                    return true; // Direct left recursion
            }
        }
        return false;
    }

    static HashSet<string> ComputeFirst(string symbol)
    {
        if (firstSets.ContainsKey(symbol))
            return firstSets[symbol];

        HashSet<string> first = new HashSet<string>();
        firstSets[symbol] = first;

```



```
            if (!grammar.ContainsKey(symbol))
            {
                first.Add(symbol); // terminal
                return first;
            }
            foreach (var production in grammar[symbol])
            {
                if (production == "ε")
                {
                    first.Add("ε");
                    continue;
                }
                bool allNullable = true;
                for (int i = 0; i < production.Length; i++)
                {
                    string sym = production[i].ToString();
                    var symFirst = ComputeFirst(sym);

                    foreach (var f in symFirst)
                    {
                        if (f != "ε")
                            first.Add(f);
                    }
                    if (!symFirst.Contains("ε"))
                    {
                        allNullable = false;
                        break;
                    }
                }
                if (allNullable)
                    first.Add("ε");
            }
            return first;
        }
}
```

OUTPUT



C:\WINDOWS\system32\cmd.exe

```
Enter grammar rules (e.g., E->TX). Type 'end' to finish input:
E->TX
X->+TX|ε
T->int|(E)
end

FIRST(E): i, (
Press any key to continue . . .
```