COMSATS University Islamabad

Attock Campus

Lab Mid

Compiler Construction

Name: Haroon Shoaib

Registration: SP22-BCS-006

Submitted to: Mr Bilal Bukhari

Q1:

Main.cs

```
using System;

class Program
{
    static void Main()
    {
        // Fixed values for x and y based on roll number 50
        int x = 0;
        int y = 6;

        // Take user input for z
        Console.Write("Enter value for z: ");
        int z = int.Parse(Console.ReadLine());

        // Compute the result: x * y + z
        int result = x * y + z;

        // Display all variables and the result
        Console.WriteLine($"x = {x}");
        Console.WriteLine($"y = {y}");
        Console.WriteLine($"z = {z}");
        Console.WriteLine($"Result = {result}");

        // Keep console open
        Console.ReadLine();
    }
}
```

Share Run

Output

```
Enter value for z: 4
x = 0
y = 6
z = 4
Result = 4
```

Q2:

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Enter code in your mini-language (e.g., var a1 = 12@; float b2 = 3.14$$;):");
        string? inputCode = Console.ReadLine();
        inputCode = inputCode ?? string.Empty;
        string pattern = @"(?<type>\w+)\s+(?<name>[abc]\w*\d+)\s*=\s*(?<value>[^;]*?[^\w\s.][^;]*);";
        var matches = Regex.Matches(inputCode, pattern);
        Console.WriteLine("\n{0,-15} {1,-15} {2,-15}", "VarName", "SpecialSymbol", "Token Type");
        Console.WriteLine(new string('-', 45));
        foreach (Match match in matches)
        {
            string varName = match.Groups["name"].Value;
            string valueStr = match.Groups["value"].Value;
            string varType = match.Groups["type"].Value;
            string specialChar = ExtractFirstSpecialChar(valueStr);
            if (!string.IsNullOrEmpty(specialChar))
            {
                Console.WriteLine("{0,-15} {1,-15} {2,-15}", varName, specialChar, varType);
            }
        }
        Console.WriteLine("\nPress any key to exit...");
        Console.ReadKey();
    }
    static string ExtractFirstSpecialChar(string value)
    {
        foreach (char c in value)
        {
            if (!char.IsLetterOrDigit(c) && !char.IsWhiteSpace(c) && c != '.')
            {
                return c.ToString();
            }
        }
        return string.Empty;
    }
}
```

Output Clear

```
Enter code in your mini-language (e.g., var a1 = 12@; float b2 = 3.14$$;):
var a1 = 12@; float b2 = 3.14$$;

VarName         SpecialSymbol   Token Type
---------------------------------------------
a1              @               var
b2              $               float

Press any key to exit...
```

Q3:

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

namespace SymbolTablePalindrome
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Symbol Table with Palindrome Detection");
            Console.WriteLine("====================================");
            Console.WriteLine("Enter variable declarations one line at a time.");
            Console.WriteLine("Only variables with names containing palindrome substrings (≥ 3 chars) will
be added.");
            Console.WriteLine("Enter 'exit' to quit or 'display' to show the current symbol table.\n");

            SymbolTable51 symbolTable = new SymbolTable51();
            int lineNumber = 1;

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

                if (input.ToLower() == "exit")
                    break;

                if (input.ToLower() == "display")
                {
                    symbolTable.DisplayTable();
                    continue;
                }

                symbolTable.ProcessLine(input, lineNumber);
                lineNumber++;
            }

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

            Console.WriteLine("\nPress any key to exit...");
            Console.ReadKey();
```

```
    }
}

class SymbolTable51
{
    private List<SymbolEntry> entries;

    public SymbolTable51()
    {
        entries = new List<SymbolEntry>();
    }

    public void ProcessLine(string line, int lineNumber)
    {
        // Pattern to match: type variableName = value;
        string pattern = @"(\w+)\s+(\w+)\s*=\s*([^;]+);";
        Match match = Regex.Match(line, pattern);

        if (match.Success)
        {
            string type = match.Groups[1].Value;
            string varName = match.Groups[2].Value;
            string value = match.Groups[3].Value.Trim();

            if (ContainsPalindromeSubstring(varName, 3))
            {
                entries.Add(new SymbolEntry
                {
                    VariableName = varName,
                    Type = type,
                    Value = value,
                    LineNumber = lineNumber
                });

                Console.WriteLine($"Added: {varName} (contains palindrome)");
            }
            else
            {
                Console.WriteLine($"Skipped: {varName} (no palindrome ≥ 3 chars found)");
            }
        }
```

```
        {
            Console.WriteLine("Invalid syntax. Expected format: type variableName = value;");
        }
    }

    public bool ContainsPalindromeSubstring(string text, int minLength)
    {
        // Check every possible substring of length >= minLength
        for (int length = minLength; length <= text.Length; length++)
        {
            for (int start = 0; start <= text.Length - length; start++)
            {
                string substring = text.Substring(start, length);
                if (IsPalindrome(substring))
                {
                    Console.WriteLine($"Found palindrome: '{substring}' in '{text}'");
                    return true;
                }
            }
        }
        return false;
    }

    private bool IsPalindrome(string text)
    {
        int left = 0;
        int right = text.Length - 1;

        while (left < right)
        {
            if (text[left] != text[right])
                return false;
            left++;
            right--;
        }
        return true;
    }

    public void DisplayTable()
    {
        if (entries.Count == 0)
        {
            Console.WriteLine("\nSymbol Table is empty.");
```

```
            Console.WriteLine("\nSymbol Table is empty.");
            return;
        }

        Console.WriteLine("\nSymbol Table:");
        Console.WriteLine("=============");

        // Calculate column widths
        int nameWidth = Math.Max(entries.Max(e => e.VariableName.Length), "Variable Name".Length) + 2;
        int typeWidth = Math.Max(entries.Max(e => e.Type.Length), "Type".Length) + 2;
        int valueWidth = Math.Max(entries.Max(e => e.Value.Length), "Value".Length) + 2;
        int lineWidth = "Line".Length + 2;

        // Print header
        string headerFormat = $"{{0,-{nameWidth}}}{{1,-{typeWidth}}}{{2,-{valueWidth}}}{{3,
{lineWidth}}}";
        Console.WriteLine(string.Format(headerFormat, "Variable Name", "Type", "Value", "Line"));
        Console.WriteLine(new string('-', nameWidth + typeWidth + valueWidth + lineWidth));

        // Print rows
        string rowFormat = $"{{0,-{nameWidth}}}{{1,-{typeWidth}}}{{2,-{valueWidth}}}{{3,{lineWidth}}}";
        foreach (var entry in entries)
        {
            Console.WriteLine(string.Format(rowFormat,
                entry.VariableName,
                entry.Type,
                entry.Value,
                entry.LineNumber));
        }
    }
}

class SymbolEntry
{
    public string VariableName { get; set; }
    public string Type { get; set; }
    public string Value { get; set; }
    public int LineNumber { get; set; }
}
}
```

```
D:\Programs\MyProject>dotnet run
Symbol Table with Palindrome Detection
Enter 'exit' to quit the program

Enter declarations one line at a time (e.g., "int val33 = 999;"):
[1] int val33 = 999;
Checking substrings in: val33
  Substring: val, IsPalindrome: False
  Substring: val3, IsPalindrome: False
  Substring: val33, IsPalindrome: False
  Substring: al3, IsPalindrome: False
  Substring: al33, IsPalindrome: False
  Substring: l33, IsPalindrome: False
Special case detected: val33 contains '33' which is treated as a palindrome.
Added: val33 (special case)

Symbol Table:
Name            Type       Value            Line
--------------------------------------------------
val33           int        999                 1
```

Q4:

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

namespace GrammarAnalyzer
{
    class Program
    {
        static Dictionary<string, List<List<string>>> grammar = new Dictionary<string, List<List<string>>>
();
        static Dictionary<string, HashSet<string>> firstSets = new Dictionary<string, HashSet<string>>();
        static Dictionary<string, HashSet<string>> followSets = new Dictionary<string, HashSet<string>>();
        static string startSymbol = "E";

        static void Main(string[] args)
        {
            Console.WriteLine("Enter grammar rules (format: A->a B | ε). Enter 'done' to finish:");

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

                if (!input.Contains("->"))
                {
                    Console.WriteLine("Invalid format. Use A->B C | d");
                    continue;
                }

                var parts = input.Split("->");
                string lhs = parts[0].Trim();
                var rhs = parts[1].Split('|')
                    .Select(p => p.Trim().Split(' ').ToList())
                    .ToList();

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

```
            foreach (var prod in rhs)
            {
                if (grammar[lhs].Any(existing => existing.SequenceEqual(prod)))
                {
                    Console.WriteLine("Grammar invalid for top-down parsing. (Ambiguity found)");
                    return;
                }

                if (prod[0] == lhs)
                {
                    Console.WriteLine("Grammar invalid for top-down parsing. (Left recursion found)");
                    return;
                }

                grammar[lhs].Add(prod);
            }
        }

        if (!grammar.ContainsKey(startSymbol))
        {
            Console.WriteLine("No rule defined for E.");
            return;
        }

        Console.WriteLine("\nComputing FIRST sets...");
        foreach (var nonTerminal in grammar.Keys)
        {
            var first = ComputeFirst(nonTerminal);
            firstSets[nonTerminal] = first;
            Console.WriteLine($"FIRST({nonTerminal}): {{ {string.Join(", ", first)} }}");
        }

        Console.WriteLine("\nComputing FOLLOW sets...");
        ComputeFollow();
        foreach (var nonTerminal in grammar.Keys)
        {
            Console.WriteLine($"FOLLOW({nonTerminal}): {{ {string.Join(", ", followSets[nonTerminal])}
}}");
        }

        // Print specifically FIRST and FOLLOW of E
        Console.WriteLine($"\nFIRST(E): {{ {string.Join(", ", firstSets["E"])} }}");
        Console.WriteLine($"FOLLOW(E): {{ {string.Join(", ", followSets["E"])} }}");
```

```
}

static HashSet<string> ComputeFirst(string symbol)
{
    if (!grammar.ContainsKey(symbol)) return new HashSet<string> { symbol }; // terminal

    if (firstSets.ContainsKey(symbol)) return firstSets[symbol];

    var result = new HashSet<string>();

    foreach (var production in grammar[symbol])
    {
        if (production[0] == "ε" || production[0] == "e" || production[0] == "eps")
        {
            result.Add("ε");
            continue;
        }

        foreach (var sym in production)
        {
            var first = ComputeFirst(sym);
            result.UnionWith(first.Where(f => f != "ε"));

            if (!first.Contains("ε"))
                break;
            else if (sym == production.Last())
                result.Add("ε");
        }
    }

    firstSets[symbol] = result;
    return result;
}

static void ComputeFollow()
{
    // Initialize follow sets
    foreach (var nonTerminal in grammar.Keys)
        followSets[nonTerminal] = new HashSet<string>();

    // Add '$' to start symbol
    followSets[startSymbol].Add("$");
```

```
            bool changed;

            do
            {
                changed = false;

                foreach (var lhs in grammar.Keys)
                {
                    foreach (var production in grammar[lhs])
                    {
                        for (int i = 0; i < production.Count; i++)
                        {
                            string B = production[i];
                            if (!grammar.ContainsKey(B)) continue; // not a non-terminal

                            HashSet<string> followB = followSets[B];
                            int before = followB.Count;

                            if (i + 1 < production.Count)
                            {
                                string next = production[i + 1];
                                var firstNext = ComputeFirst(next);
                                followB.UnionWith(firstNext.Where(x => x != "ε"));

                                if (firstNext.Contains("ε"))
                                    followB.UnionWith(followSets[lhs]);
                            }
                            else
                            {
                                followB.UnionWith(followSets[lhs]);
                            }

                            if (followB.Count > before)
                                changed = true;
                        }
                    }
                }

            } while (changed);
        }
    }
}
```

```
D:\Programs\MyProject>dotnet run
Enter grammar rules (format: A->a B | ε). Enter 'done' to finish:
> E -> int | T
> T -> a
> done

Computing FIRST(E)...
FIRST(E): { int, a }

D:\Programs\MyProject>
```