COMSATS University Islamabad

Attock Campus

Lab Mid

Compiler Construction

Name: Abdullah Qaisar

Registration: SP22-BCS-001

Submitted to: Mr Bilal Bukhari

Question 01

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

namespace CustomStringProcessor
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Custom String Processor");
            Console.WriteLine("======================");

            // The custom string to process
            string customString = "x:0; y:1; z:userinput; result: x * y + z;";
            Console.WriteLine($"Processing string: \"{customString}\"");

            // Extract variables and values
            // Using student roll number 01: x = 0, y = 1
            double x01 = 0; // first digit of roll number
            double y = 1;   // second digit of roll number
            double z = 0;

            // Get user input for z (since it's marked as "userinput" in the
string)
            Console.Write("\nEnter value for z: ");
            string zInput = Console.ReadLine();
            z = Convert.ToDouble(zInput);

            // Extract the formula from the custom string
            Match formulaMatch = Regex.Match(customString, @"result: (.*?);");
            string formulaStr = "";
            if (formulaMatch.Success)
            {
                formulaStr = formulaMatch.Groups[1].Value;
                Console.WriteLine($"\nFormula: {formulaStr}");
            }

            // Perform the operation: result = x * y + z
            double result = x01 * y + z;

            // Display original variables with values and final result
```

```
            Console.WriteLine("\nOriginal Variables:");
            Console.WriteLine($"x = {x01}");
            Console.WriteLine($"y = {y}");
            Console.WriteLine($"z = {z}");
            Console.WriteLine($"Result = {result}");

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

```
PS C:\Users\abdul\Desktop\c#\ques1> dotnet run
Custom String Processor
======================
Processing string: "x:0; y:1; z:userinput; result: x * y + z;"

Enter value for z: 15

Formula: x * y + z

Original Variables:
x = 0
y = 1
z = 15
Result = 15

Press any key to exit...
```

Question 02

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

namespace MiniLanguageParser
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Mini-Language Variable Extractor");
            Console.WriteLine("================================");
            Console.WriteLine("Enter your code (type 'END' on a new line when
finished):");

            // Collect input code from user
            string inputCode = "";
            string line;
            while ((line = Console.ReadLine()) != "END")
            {
                inputCode += line + Environment.NewLine;
            }

            // Define the regex pattern:
            // - Variable declaration (var or float)
            // - Variable name starts with a, b, or c
            // - Variable name ends with digits
            // - Value contains a non-alphanumeric special character
            string pattern = @"(var|float)\s+([abc][a-zA-Z0-
9]*\d+)\s*=\s*([^;]*?[^a-zA-Z0-9\s][^;]*);";

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

            // Create a list to store the extracted variables
            var variables = new List<(string VarName, string SpecialSymbol,
string TokenType)>();

            foreach (Match match in matches)
            {
                string tokenType = match.Groups[1].Value;
                string varName = match.Groups[2].Value;
```

```
                string fullValue = match.Groups[3].Value;


                // Extract special symbol from the value
                string specialSymbol = Regex.Match(fullValue, @"[^a-zA-Z0-
9\s]").Value;

                variables.Add((varName, specialSymbol, tokenType));
            }

            // Display the results in a table
            Console.WriteLine("\nResults:");
            Console.WriteLine("==================================================
=====");
            Console.WriteLine("| {0,-15} | {1,-15} | {2,-15} |", "VarName",
"SpecialSymbol", "Token Type");
            Console.WriteLine("==================================================
=====");

            foreach (var variable in variables)
            {
                Console.WriteLine("| {0,-15} | {1,-15} | {2,-15} |",
                    variable.VarName,
                    variable.SpecialSymbol,
                    variable.TokenType);
            }
            Console.WriteLine("==================================================
=====");

            if (variables.Count == 0)
            {
                Console.WriteLine("No matching variables found.");
            }

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

```
PS C:\Users\abdul\Desktop\c#\ques2> dotnet run
Mini-Language Variable Extractor
================================
Enter your code (type 'END' on a new line when finished):
var a1=123!; var b1=321@; float c1=3.1$;
END

Results:
=====================================================
| VarName          | SpecialSymbol    | Token Type      |
=====================================================
| a1               | !                | var             |
| b1               | @                | var             |
| c1               | .                | float           |
=====================================================

Press any key to exit...
```

Question 03

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

namespace SymbolTablePalindrome
{
    // Class to represent a symbol table entry
    class Symbol
    {
        public string Name { get; set; }
        public string Type { get; set; }
        public string Value { get; set; }
        public int LineNumber { get; set; }

        public override string ToString()
        {
            return $"{Name} | {Type} | {Value} | {LineNumber}";
        }
    }

    class Program
    {
        // Symbol table to store variables
        static List<Symbol> symbolTable = new List<Symbol>();
        static int lineCounter = 1;

        static void Main(string[] args)
        {
            Console.WriteLine("Symbol Table with Palindrome Validation");
            Console.WriteLine("======================================");
            Console.WriteLine("Enter variable declarations one line at a time
(type 'EXIT' to quit):");

            string input;
            while ((input = Console.ReadLine()) != "EXIT")
            {
                ProcessLine(input);
                lineCounter++;
            }

            // Display the symbol table
```

```
            DisplaySymbolTable();

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

        static void ProcessLine(string line)
        {
            // Pattern to match variable declarations like "int val33 = 999;"
            var match = Regex.Match(line, @"(\w+)\s+(\w+)\s*=\s*([^;]+);");

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

                // Check if the variable NAME contains a palindrome substring
                string palindromeFound = FindPalindromeSubstring(name, 3);

                if (!string.IsNullOrEmpty(palindromeFound))
                {
                    // Add to symbol table
                    symbolTable.Add(new Symbol
                    {
                        Name = name,
                        Type = type,
                        Value = value,
                        LineNumber = lineCounter
                    });

                    Console.WriteLine($"Added: {name} (contains palindrome:
'{palindromeFound}')");
                }
                else
                {
                    Console.WriteLine($"Skipped: {name} (no palindrome substring
of length >= 3)");
                }
            }
            else
            {
                Console.WriteLine("Invalid declaration format. Expected: \"type
name = value;\"");
            }
```

```
        }

        static string FindPalindromeSubstring(string str, int minLength)
        {
            // Custom implementation to find palindrome substrings
            for (int i = 0; i < str.Length; i++)
            {
                // Try all possible substring lengths starting from minLength
                for (int len = minLength; i + len <= str.Length; len++)
                {
                    bool isPalindrome = true;
                    string substr = str.Substring(i, len);

                    // Check if this substring is a palindrome
                    for (int j = 0; j < len / 2; j++)
                    {
                        if (substr[j] != substr[len - j - 1])
                        {
                            isPalindrome = false;
                            break;
                        }
                    }

                    if (isPalindrome)
                    {
                        return substr;
                    }
                }
            }

            return null;
        }

        static void DisplaySymbolTable()
        {
            Console.WriteLine("\nSymbol Table Contents:");
            Console.WriteLine("================================================
===");
            Console.WriteLine("| Variable Name | Type      | Value      | Line
No. |");
            Console.WriteLine("================================================
===");

            foreach (var symbol in symbolTable)
            {
```

```
                Console.WriteLine($"| {symbol.Name,-13} | {symbol.Type,-9} |
{symbol.Value,-10} | {symbol.LineNumber,-8} |");
            }

            Console.WriteLine("=============================================================
===");

            if (symbolTable.Count == 0)
            {
                Console.WriteLine("No valid entries in symbol table.");
            }
        }
    }
}
```

```
PS C:\Users\abdul\Desktop\c#\ques3> dotnet run
Symbol Table with Palindrome Validation
=====================================
Enter variable declarations one line at a time (type 'EXIT' to quit):
var a333=123;
Added: a333 (contains palindrome: '333')
int a431=999;
Skipped: a431 (no palindrome substring of length >= 3)
float a414=132;
Added: a414 (contains palindrome: '414')
EXIT

Symbol Table Contents:
=====================================================
| Variable Name | Type      | Value      | Line No. |
=====================================================
| a333          | var       | 123        | 1        |
| a414          | float     | 132        | 3        |
=====================================================

Press any key to exit...
```

Question 04

```
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);
        }
    }
}
```

```
PS C:\Users\abdul\Desktop\c#\ques4> dotnet run
Enter grammar rules (format: A->a B | ε). Enter 'done' to finish:
> E -> int | T
> T -> a
> done

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

Computing FOLLOW sets...
FOLLOW(E): { $ }
FOLLOW(T): { $ }

FIRST(E): { int, a }
FOLLOW(E): { $ }
```