Comsats University Islamabad, Attock Campus

Lab MID

Name:

Muhammad Anas

Registration no:

Sp22-Bcs-042

Submitted To:

Syed Bilal Haider

Date:

11th April ,2025

Task 1:

Code:

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

namespace ComplexStringEvaluator
{
    class EvaluatorEngine
    {
        static void Main(string[] args)
        {

            var formula = "x:4; y:2; z:userinput; result: x * y + z;";

            Console.WriteLine("Formula to evaluate: " + formula);

            // Parse the expression into components
            var tokenizer = new FormulaTokenizer(formula);
            var variableSet = tokenizer.BuildVariableSet();

            // Dynamic variable collection
            Dictionary<string, int> variables = new Dictionary<string, int>();

            // Add predefined variables
            variables["x"] = tokenizer.ExtractConstant("x");
            variables["y"] = tokenizer.ExtractConstant("y");

            // Handle user input for z
            Console.WriteLine($"Input value for z:");
            variables["z"] = Convert.ToInt32(Console.ReadLine());

            // Process the equation
            string equationStructure = tokenizer.GetComputationStructure();
            int finalValue = ComputeExpression(variables, equationStructure);

            // Display summary
            Console.WriteLine("\n--- Computation Results ---");
            foreach (var entry in variables)
            {
                Console.WriteLine($"{entry.Key} = {entry.Value}");
```

```
            }
            Console.WriteLine($"Final Outcome = {finalValue}");

            Console.WriteLine("\nTerminate with any key...");
            Console.ReadKey();
        }

        static int ComputeExpression(Dictionary<string, int> vars, string expr)
        {
            // This simplistic approach assumes the expression is x * y + z
            return vars["x"] * vars["y"] + vars["z"];
        }
    }

    class FormulaTokenizer
    {
        private readonly string _inputExpression;

        public FormulaTokenizer(string expression)
        {
            _inputExpression = expression;
        }

        public List<string> BuildVariableSet()
        {
            return new List<string> { "x", "y", "z" };
        }

        public int ExtractConstant(string varName)
        {
            var pattern = varName + @":(\d+)";
            var match = Regex.Match(_inputExpression, pattern);

            if (match.Success)
            {
                return int.Parse(match.Groups[1].Value);
            }

            // Fallback values for x and y
            if (varName == "x") return 4;
            if (varName == "y") return 2;

            return 0;
        }
```

```
        public string GetComputationStructure()
        {
            var opPattern = @"result:\s*(.+?);";
            var match = Regex.Match(_inputExpression, opPattern);

            if (match.Success)
            {
                return match.Groups[1].Value.Trim();
            }

            return "x * y + z"; // Default computation
        }
    }
}
```

Output:

```
C:\Users\Muhammad Anas\Desktop\CC Lab MID\MyProject>dotnet run
Formula to evaluate: x:4; y:2; z:userinput; result: x * y + z;
Input value for z:
2

--- Computation Results ---
x = 4
y = 2
z = 2
Final Outcome = 10

Terminate with any key...
```

Task 2:

Code:

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

namespace MiniLanguageAnalyzer
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            Console.WriteLine("Mini-Language Variable Analyzer");
            Console.WriteLine("================================");
            Console.WriteLine("Enter code sample (e.g., var a1 = 12@; float b2 =
3.14$$;):");

            string inputCode = Console.ReadLine();

        // If input is empty, use the example from the question
            if (string.IsNullOrWhiteSpace(inputCode))
            {
                inputCode = "var a1 = 12@; float b2 = 3.14$$;";
                Console.WriteLine("Using example input: " + inputCode);
            }
            string pattern = @"([abc]\w*\d+)\s*=\s*([^;]*?[^\w\s.][^;]*?);";

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

            // Create list to store results
            List<VariableInfo> variables = new List<VariableInfo>();

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

                // Extract special symbols
                string specialSymbols = ExtractSpecialSymbols(value);

                // Determine token type
                string tokenType = DetermineTokenType(value);

                if (!string.IsNullOrEmpty(specialSymbols))
                {
                    variables.Add(new VariableInfo(varName, specialSymbols,
tokenType));
                }
            }

            // Display results in a table
            DisplayResultsTable(variables);

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

```
        static string ExtractSpecialSymbols(string value)
        {
            StringBuilder specialChars = new StringBuilder();

            foreach (char c in value)
            {
                if (!char.IsLetterOrDigit(c) && !char.IsWhiteSpace(c) && c !=
'.')
                {
                    specialChars.Append(c);
                }
            }

            return specialChars.ToString();
        }

        static string DetermineTokenType(string value)
        {
            // Simple token type determination
            if (value.Contains("."))
            {
                return "FloatLiteral";
            }
            else if (int.TryParse(value.TrimEnd(value.Where(c =>
!char.IsDigit(c)).ToArray()), out _))
            {
                return "IntegerLiteral";
            }
            else
            {
                return "Unknown";
            }
        }

        static void DisplayResultsTable(List<VariableInfo> variables)
        {
            if (variables.Count == 0)
            {
                Console.WriteLine("\nNo matching variables found.");
                return;
            }

            // Table header
            Console.WriteLine("\nAnalysis Results:");
```

```
            Console.WriteLine("-----------------------------------------------------
-------");
            Console.WriteLine("|  VarName  |  SpecialSymbol  |    Token
Type    |");
            Console.WriteLine("-----------------------------------------------------
-------");

            // Table rows
            foreach (var variable in variables)
            {
                Console.WriteLine($"|  {variable.Name,-
8}  |  {variable.SpecialSymbols,-13}  |  {variable.TokenType,-16}  |");
            }

            Console.WriteLine("-----------------------------------------------------
-------");
        }}
    class VariableInfo
    {
        public string Name { get; }
        public string SpecialSymbols { get; }
        public string TokenType { get; }

        public VariableInfo(string name, string specialSymbols, string tokenType)
        {
            Name = name;
            SpecialSymbols = specialSymbols;
            TokenType = tokenType;
        } }}
```

Output:

```
C:\Users\Muhammad Anas\Desktop\CC Lab MID\task2>dotnet run
C:\Users\Muhammad Anas\Desktop\CC Lab MID\task2\Program.cs(16,32): warning CS8600: Converting null literal or possible
null value to non-nullable type. [C:\Users\Muhammad Anas\Desktop\CC Lab MID\task2\task2.csproj]
Mini-Language Variable Analyzer
==============================
Enter code sample (e.g., var a1 = 12@; float b2 = 3.14$$;):
var a1 = 32@; float b2 = 5.99$$$; float c3 = 4.33$;

Analysis Results:
-----------------------------------------------------
|  VarName  |  SpecialSymbol  |    Token Type    |
-----------------------------------------------------
|  a1       |  @              |  IntegerLiteral    |
|  b2       |  $$$            |  FloatLiteral      |
|  c3       |  $              |  FloatLiteral      |
-----------------------------------------------------

Press any key to exit...
```

Task 3:

Code:

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

namespace PatternSequenceValidator
{
    // Record for tracking lexical elements
    public record LexicalItem
    {
        public string Identifier { get; init; }
        public string Category { get; init; }
        public string Content { get; init; }
        public int Position { get; init; }

        public string FormatInfo() => $"{Identifier} -> {Category} -> {Content} @
line {Position}";
    }

    class Analyzer
    {
        // Storage for valid lexical elements
        private readonly List<LexicalItem> _lexicalStorage = new();
        private int _positionTracker = 1;

        static void Main()
        {
            var analyzer = new Analyzer();
            Console.WriteLine("Mirrored Sequence Pattern Recognition Engine");
            Console.WriteLine("-----------------------------------------");
            Console.WriteLine("Input patterns for analysis (type 'TERMINATE' to
finish):");

            string userInput;
            while ((userInput = Console.ReadLine()) != "TERMINATE")
            {
                analyzer.EvaluateSequence(userInput);
                analyzer._positionTracker++;
            }
```

```
            // Present collected data
            analyzer.GenerateReport();

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

        void EvaluateSequence(string sequence)
        {
            // Extract components using expression pattern
            var patternMatch = Regex.Match(sequence,
@"(\w+)\s+(\w+)\s*=\s*([^;]+);");

            if (patternMatch.Success)
            {
                string category = patternMatch.Groups[1].Value;
                string identifier = patternMatch.Groups[2].Value;
                string content = patternMatch.Groups[3].Value.Trim();

                // Verify if identifier contains mirror patterns
                string mirrorSegment = DetectMirrorSequence(identifier, 3);

                if (!string.IsNullOrEmpty(mirrorSegment))
                {
                    // Record the verified item
                    _lexicalStorage.Add(new LexicalItem
                    {
                        Identifier = identifier,
                        Category = category,
                        Content = content,
                        Position = _positionTracker
                    });

                    Console.WriteLine($"Validated: {identifier} (mirror pattern
detected: '{mirrorSegment}')");
                }
                else
                {
                    Console.WriteLine($"Rejected: {identifier} (no mirror
sequence of minimum length 3)");
                }
            }
            else
            {
```

```
                Console.WriteLine("Invalid syntax. Expected format: \"category
identifier = content;\"");
            }
        }

        string DetectMirrorSequence(string input, int minimumLength)
        {
            // Algorithm to identify mirror sequence patterns
            for (int startPosition = 0; startPosition < input.Length;
startPosition++)
            {
                // Evaluate all potential sequence lengths
                for (int sequenceLength = minimumLength; startPosition +
sequenceLength <= input.Length; sequenceLength++)
                {
                    bool isMirrorPattern = true;
                    string candidateSequence = input.Substring(startPosition,
sequenceLength);

                    // Verify if sequence reads the same forward and backward
                    for (int charIndex = 0; charIndex < sequenceLength / 2;
charIndex++)
                    {
                        if (candidateSequence[charIndex] !=
candidateSequence[sequenceLength - charIndex - 1])
                        {
                            isMirrorPattern = false;
                            break;
                        }
                    }

                    if (isMirrorPattern)
                    {
                        return candidateSequence;
                    }
                }
            }

            return null;
        }

        void GenerateReport()
        {
            Console.WriteLine("\nValidated Pattern Repository:");
```

```
            Console.WriteLine("================================================="
);
            Console.WriteLine("| Identifier     | Category  | Content    | Pos
|");
            Console.WriteLine("================================================="
);

            foreach (var item in _lexicalStorage)
            {
                Console.WriteLine($"| {item.Identifier,-14} | {item.Category,-9}
| {item.Content,-10} | {item.Position,-3} |");
            }

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

            if (!_lexicalStorage.Any())
            {
                Console.WriteLine("Repository is empty. No valid patterns
found.");
            }
        }
    }
}
```

Output:

```
C:\Users\Muhammad Anas\Desktop\CC Lab MID\task3>dotnet run
Mirrored Sequence Pattern Recognition Engine
-----------------------------------------
Input patterns for analysis (type 'TERMINATE' to finish):
int a333= 554;
Validated: a333 (mirror pattern detected: '333')
TERMINATE

Validated Pattern Repository:
=================================================
| Identifier     | Category  | Content    | Pos |
=================================================
| a333           | int       | 554        | 1   |
=================================================

Press any key to finish...
```

Task 4:

Code:

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

namespace GrammarAnalyzer
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Grammar FIRST and FOLLOW Sets Calculator");
            Console.WriteLine("=====================================");
            Console.WriteLine("Enter grammar rules (one per line). Enter an empty
line to finish input.");
            Console.WriteLine("Format: non-terminal -> sequence1 | sequence2 |
...");
            Console.WriteLine("Example: E -> T X");
            Console.WriteLine("         X -> + T X | ε");
            Console.WriteLine("         T -> int | ( E )");

            // Read grammar rules from user
            List<string> inputRules = new List<string>();
            string input;

            while (!string.IsNullOrWhiteSpace(input = Console.ReadLine()))
            {
                inputRules.Add(input);
            }

            // Parse rules and check for left recursion
            var grammar = new Grammar();

            if (!grammar.ParseRules(inputRules))
            {
                Console.WriteLine("Error parsing grammar rules. Please check the
format.");
                return;
            }
```

```
            // Check for left recursion
            if (grammar.HasLeftRecursion())
            {
                Console.WriteLine("Grammar invalid for top-down parsing.");
                return;
            }

            // Compute FIRST sets
            var firstSets = grammar.ComputeFirstSets();
            Console.WriteLine("\nFIRST Sets:");
            PrintSets(firstSets);

            // Compute FOLLOW sets
            var followSets = grammar.ComputeFollowSets(firstSets);
            Console.WriteLine("\nFOLLOW Sets:");
            PrintSets(followSets);

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

        static void PrintSets(Dictionary<string, HashSet<string>> sets)
        {
            foreach (var entry in sets)
            {
                Console.Write($"FIRST({entry.Key}) = {{ ");
                Console.Write(string.Join(", ", entry.Value));
                Console.WriteLine(" }");
            }
        }
    }

    class Grammar
    {
        // Dictionary to store the production rules
        private Dictionary<string, List<List<string>>> rules = new
Dictionary<string, List<List<string>>>();
        private HashSet<string> nonTerminals = new HashSet<string>();
        private HashSet<string> terminals = new HashSet<string>();
        private string startSymbol = null;

        // Parse the input rules
        public bool ParseRules(List<string> inputRules)
        {
            if (inputRules.Count == 0)
```

```
                return false;

            // Regular expression to match grammar rules
            var rulePattern = new Regex(@"^(\w+)\s*->\s*(.+)$");

            foreach (var inputRule in inputRules)
            {
                var match = rulePattern.Match(inputRule);
                if (!match.Success)
                    return false;

                string nonTerminal = match.Groups[1].Value.Trim();
                nonTerminals.Add(nonTerminal);

                // Set the first non-terminal as the start symbol
                if (startSymbol == null)
                    startSymbol = nonTerminal;

                // Split alternatives (separated by |)
                string rightSide = match.Groups[2].Value;
                string[] alternatives = rightSide.Split('|');

                List<List<string>> productions = new List<List<string>>();

                foreach (var alt in alternatives)
                {
                    // Split the alternative into symbols
                    List<string> symbols = new List<string>();
                    string[] tokens = alt.Trim().Split(' ',
StringSplitOptions.RemoveEmptyEntries);

                    foreach (var token in tokens)
                    {
                        if (token == "ε")
                        {
                            symbols.Add("ε"); // Epsilon
                        }
                        else
                        {
                            symbols.Add(token);
                            // If it's not a non-terminal, it's a terminal
                            if (!nonTerminals.Contains(token))
                                terminals.Add(token);
                        }
                    }
```

```
                productions.Add(symbols);
            }

            // Add or update the rule
            if (rules.ContainsKey(nonTerminal))
                rules[nonTerminal].AddRange(productions);
            else
                rules[nonTerminal] = productions;
        }

        // Verify all symbols in the rules are defined
        foreach (var entry in rules)
        {
            foreach (var production in entry.Value)
            {
                foreach (var symbol in production)
                {
                    if (symbol != "ε" && !nonTerminals.Contains(symbol) &&
!terminals.Contains(symbol))
                    {
                        Console.WriteLine($"Warning: Symbol '{symbol}' is
used but not defined.");
                    }
                }
            }
        }

        return true;
    }

    // Check if the grammar has left recursion
    public bool HasLeftRecursion()
    {
        foreach (var entry in rules)
        {
            string nonTerminal = entry.Key;

            // Direct left recursion
            foreach (var production in entry.Value)
            {
                if (production.Count > 0 && production[0] == nonTerminal)
                {
                    Console.WriteLine($"Direct left recursion found in rule:
{nonTerminal} -> {string.Join(" ", production)}");
```

```
                        return true;
                    }
                }

                // Indirect left recursion (simplified check)
                HashSet<string> visited = new HashSet<string>();
                if (HasIndirectLeftRecursion(nonTerminal, nonTerminal, visited))
                {
                    Console.WriteLine($"Indirect left recursion detected
involving {nonTerminal}");
                    return true;
                }
            }

            return false;
        }

        private bool HasIndirectLeftRecursion(string original, string current,
HashSet<string> visited)
        {
            if (visited.Contains(current))
                return false;

            visited.Add(current);

            // Check each production of current non-terminal
            if (rules.ContainsKey(current))
            {
                foreach (var production in rules[current])
                {
                    if (production.Count > 0)
                    {
                        string first = production[0];

                        // If first symbol is the original non-terminal, we found
indirect left recursion
                        if (first == original && current != original)
                            return true;

                        // If first symbol is a non-terminal, continue checking
                        if (nonTerminals.Contains(first) && first != current)
                        {
                            if (HasIndirectLeftRecursion(original, first, new
HashSet<string>(visited)))
                                return true;
```

```
                        }
                    }
                }
            }

            return false;
        }

        // Compute FIRST sets for all non-terminals
        public Dictionary<string, HashSet<string>> ComputeFirstSets()
        {
            Dictionary<string, HashSet<string>> firstSets = new
Dictionary<string, HashSet<string>>();

            // Initialize FIRST sets
            foreach (var terminal in terminals)
            {
                firstSets[terminal] = new HashSet<string> { terminal };
            }

            foreach (var nonTerminal in nonTerminals)
            {
                firstSets[nonTerminal] = new HashSet<string>();
            }

            bool changed;
            do
            {
                changed = false;

                foreach (var entry in rules)
                {
                    string nonTerminal = entry.Key;

                    foreach (var production in entry.Value)
                    {
                        // If production is empty or epsilon, add epsilon to
FIRST set
                        if (production.Count == 0 || (production.Count == 1 &&
production[0] == "ε"))
                        {
                            if (firstSets[nonTerminal].Add("ε"))
                                changed = true;
                            continue;
                        }
```

```
                    // Process each symbol in the production
                    bool allCanDeriveEpsilon = true;
                    for (int i = 0; i < production.Count; i++)
                    {
                        string symbol = production[i];

                        if (symbol == "ε")
                            continue;

                        // Add FIRST(symbol) - {ε} to FIRST(nonTerminal)
                        bool epsilonInFirst = false;
                        foreach (var terminal in firstSets[symbol])
                        {
                            if (terminal != "ε")
                            {
                                if (firstSets[nonTerminal].Add(terminal))
                                    changed = true;
                            }
                            else
                            {
                                epsilonInFirst = true;
                            }
                        }

                        // If the symbol cannot derive epsilon, stop here
                        if (!epsilonInFirst)
                        {
                            allCanDeriveEpsilon = false;
                            break;
                        }

                        // If we've reached the last symbol and all can
derive epsilon
                        if (i == production.Count - 1 && allCanDeriveEpsilon)
                        {
                            if (firstSets[nonTerminal].Add("ε"))
                                changed = true;
                        }
                    }
                }
            }
        } while (changed);

        return firstSets;
```

```
        }

        // Compute FOLLOW sets given the FIRST sets
        public Dictionary<string, HashSet<string>>
ComputeFollowSets(Dictionary<string, HashSet<string>> firstSets)
        {
            Dictionary<string, HashSet<string>> followSets = new
Dictionary<string, HashSet<string>>();

            // Initialize FOLLOW sets for all non-terminals
            foreach (var nonTerminal in nonTerminals)
            {
                followSets[nonTerminal] = new HashSet<string>();
            }

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

            bool changed;
            do
            {
                changed = false;

                foreach (var entry in rules)
                {
                    string nonTerminal = entry.Key;

                    foreach (var production in entry.Value)
                    {
                        for (int i = 0; i < production.Count; i++)
                        {
                            string symbol = production[i];

                            // Skip terminals and epsilon
                            if (!nonTerminals.Contains(symbol))
                                continue;

                            // Compute what follows the symbol in this production
                            bool allRemainderCanDeriveEpsilon = true;

                            // Process all symbols after the current one
                            for (int j = i + 1; j < production.Count; j++)
                            {
                                string nextSymbol = production[j];
```

```
                            // Skip epsilon
                            if (nextSymbol == "ε")
                                continue;

                            // Add FIRST(nextSymbol) - {ε} to FOLLOW(symbol)
                            bool epsilonInNext = false;
                            foreach (var terminal in firstSets[nextSymbol])
                            {
                                if (terminal != "ε")
                                {
                                    if (followSets[symbol].Add(terminal))
                                        changed = true;
                                }
                                else
                                {
                                    epsilonInNext = true;
                                }
                            }

                            // If nextSymbol cannot derive epsilon, stop here
                            if (!epsilonInNext)
                            {
                                allRemainderCanDeriveEpsilon = false;
                                break;
                            }
                        }

                        // If all remaining symbols can derive epsilon or
we're at the end
                        // Add FOLLOW(nonTerminal) to FOLLOW(symbol)
                        if (allRemainderCanDeriveEpsilon)
                        {
                            foreach (var terminal in followSets[nonTerminal])
                            {
                                if (followSets[symbol].Add(terminal))
                                    changed = true;
                            }
                        }
                    }    }
                }
            } while (changed);
            return followSets;
        }
    }
}
```

Output:

```
C:\Users\Muhammad Anas\Desktop\CC Lab MID\task4\Program.cs(76,38): warning CS8625: Cannot convert null literal to non-n
ullable reference type. [C:\Users\Muhammad Anas\Desktop\CC Lab MID\task4\task4.csproj]
Grammar FIRST and FOLLOW Sets Calculator
====================================
Enter grammar rules (one per line). Enter an empty line to finish input.
Format: non-terminal -> sequence1 | sequence2 | ...
Example: E -> T X
         X -> + T X | ε
         T -> int | ( E )
E -> int | a


FIRST Sets:
FIRST(int) = { int }
FIRST(a) = { a }
FIRST(E) = { int, a }

FOLLOW Sets:
FIRST(E) = { $ }

Press any key to exit...
```