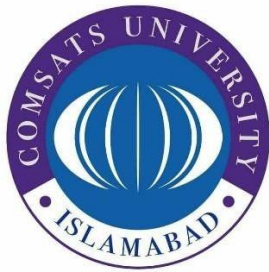


Comsats University Islamabad Attock campus



Lab MID

Submitted by:

Kulsoom bano(sp22-bcs-047)

Submitted to:

Sir Bilal Haider

Course title:

CC LAB

Deadline:

April 11, 2025

Department of computer sciences

Question1:

```
using System;

namespace ModifiedVariableProgram
{
    class Program
    {
        static void Main()
        {

            int x = 5;
            int y = 9;
            Console.Write("Enter value for z: ");
            int z = int.Parse(Console.ReadLine());
            int result = x * y + z;

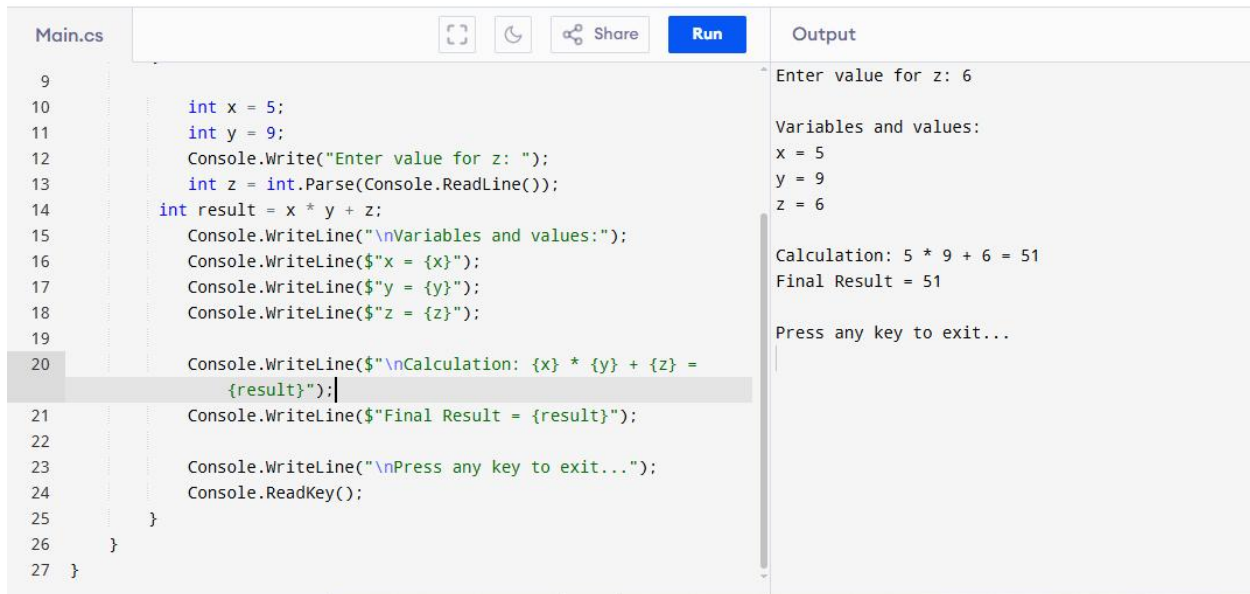
            Console.WriteLine("\nVariables and values:");
            Console.WriteLine($"x = {x}");
            Console.WriteLine($"y = {y}");
            Console.WriteLine($"z = {z}");

            Console.WriteLine($"
Calculation: {x} * {y} + {z} = {result}");
            Console.WriteLine($"Final Result = {result}");

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

```
}  
  
}
```

Output:



```
Main.cs  
9  
10     int x = 5;  
11     int y = 9;  
12     Console.Write("Enter value for z: ");  
13     int z = int.Parse(Console.ReadLine());  
14     int result = x * y + z;  
15     Console.WriteLine("\nVariables and values:");  
16     Console.WriteLine($"x = {x}");  
17     Console.WriteLine($"y = {y}");  
18     Console.WriteLine($"z = {z}");  
19  
20     Console.WriteLine($"\\nCalculation: {x} * {y} + {z} =  
    {result}");  
21     Console.WriteLine($"Final Result = {result}");  
22  
23     Console.WriteLine("\\nPress any key to exit...");  
24     Console.ReadKey();  
25 }  
26 }  
27 }
```

Output

```
Enter value for z: 6  
  
Variables and values:  
x = 5  
y = 9  
z = 6  
  
Calculation: 5 * 9 + 6 = 51  
Final Result = 51  
  
Press any key to exit...
```

Question2:

```
using System;
```

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

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

```
namespace MiniLanguageLexerConsole
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            Console.WriteLine("Mini-Language Lexical Analyzer (Console)");
```

```
            Console.WriteLine("Enter code (press Enter twice to analyze):");
```

```

// Read multi-line input until double Enter
string inputCode = ReadMultiLineInput();

// Analyze and display results
List<VariableToken> tokens = LexicalAnalyzer.AnalyzeCode(inputCode);
DisplayResults(tokens);
}

static string ReadMultiLineInput()
{
    string input = "";
    string line;
    while (!string.IsNullOrEmpty(line = Console.ReadLine()))
    {
        input += line + Environment.NewLine;
    }
    return input;
}

static void DisplayResults(List<VariableToken> tokens)
{
    Console.WriteLine("\nResults:");
    Console.WriteLine("-----");
    Console.WriteLine("| {0,-10} | {1,-12} | {2,-8} |",
        "VarName", "SpecialSymbol", "Type");

```

```

        Console.WriteLine("-----");

        foreach (var token in tokens)
        {
            Console.WriteLine("| {0,-10} | {1,-12} | {2,-8} |",
                               token.VarName, token.SpecialSymbol, token.TokenType);
        }
        Console.WriteLine("-----");
    }
}

public static class LexicalAnalyzer
{
    public static List<VariableToken> AnalyzeCode(string code)
    {
        List<VariableToken> tokens = new List<VariableToken>();
        string pattern = @"\"b(var\s+|float\s+)?([abc][a-zA-Z]*\d+)\s*=\s*([^\s;]+);$";

        foreach (Match match in Regex.Matches(code, pattern))
        {
            string varName = match.Groups[2].Value;
            string value = match.Groups[3].Value.Trim();
            string specialSymbols = Regex.Replace(value, @"\"w\\.]", "");

            if (!string.IsNullOrEmpty(specialSymbols))
            {

```

```

tokens.Add(new VariableToken
{
    VarName = varName,
    SpecialSymbol = specialSymbols,
    TokenType = match.Groups[1].Value.Contains("float") ? "Float" : "Integer"
});
}
}
return tokens;
}
}

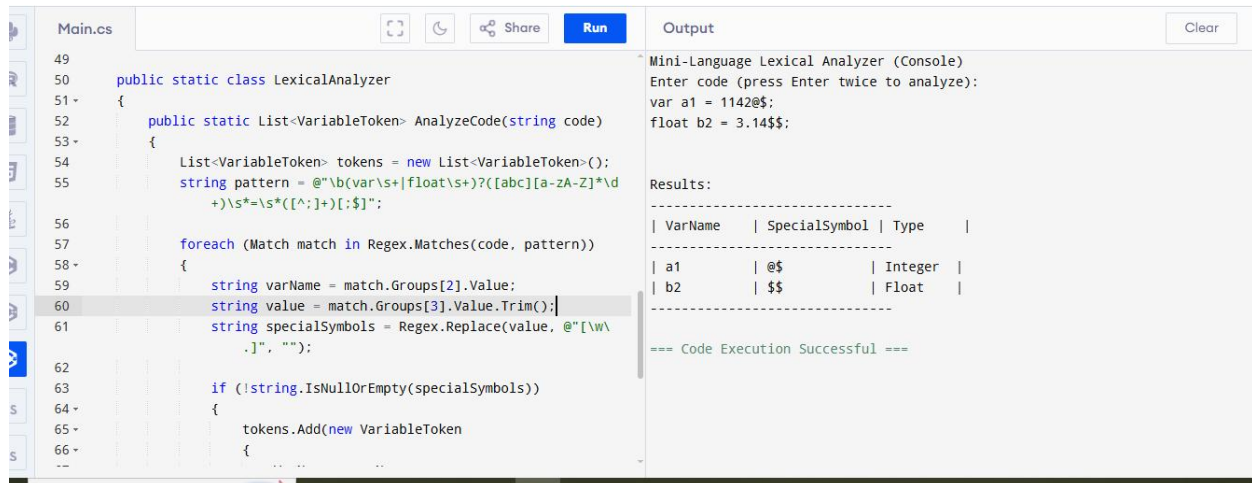
```

```

public class VariableToken
{
    public string VarName { get; set; }
    public string SpecialSymbol { get; set; }
    public string TokenType { get; set; }
}
}

```

Output:



The screenshot shows a C# IDE with a file named 'Main.cs'. The code defines a 'LexicalAnalyzer' class with a static method 'AnalyzeCode' that takes a string 'code' and returns a list of 'VariableToken' objects. The code uses a regular expression to parse the input string 'var a1 = 1142@;\$; float b2 = 3.14\$;\$;'. The output window shows the results of the analysis, including a table of tokens and a message indicating successful execution.

```
49
50 public static class LexicalAnalyzer
51 {
52     public static List<VariableToken> AnalyzeCode(string code)
53     {
54         List<VariableToken> tokens = new List<VariableToken>();
55         string pattern = @"(var\s+|float\s+)?([a-zA-Z]*\d
56         +)\s*=\s*([^\s;]+);";
57         foreach (Match match in Regex.Matches(code, pattern))
58         {
59             string varName = match.Groups[2].Value;
60             string value = match.Groups[3].Value.Trim();
61             string specialSymbols = Regex.Replace(value, @"[\w\
62             .]", "");
63             if (!string.IsNullOrEmpty(specialSymbols))
64             {
65                 tokens.Add(new VariableToken
66                 {
67                     ...
68                 })
69             }
70         }
71         return tokens;
72     }
73 }
```

Output:

```
Mini-Language Lexical Analyzer (Console)
Enter code (press Enter twice to analyze):
var a1 = 1142@;$;
float b2 = 3.14$;$;

Results:
-----
| VarName | SpecialSymbol | Type |
-----
| a1      | @$           | Integer |
| b2      | $$           | Float   |
-----

=== Code Execution Successful ===
```

Question3:

Question3:
using System;
using System.Collections.Generic;

namespace SymbolTableWithPalindromeCheck

```
{
    class Program
    {
        static void Main(string[] args)
        {
            SymbolTable symbolTable = new SymbolTable();
            int lineNumber = 1;

            Console.WriteLine("Symbol Table with Palindrome Check");
            Console.WriteLine("Enter variable declarations (empty line to exit):");
            Console.WriteLine("Format: <type> <name> = <value>; Example: int val33 = 999;");

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

                if (string.IsNullOrEmpty(input))
                    break;

                try
                {
                    // Parse the input
                }
            }
        }
    }
}
```

```

string[] parts = input.Split(new[] { ' ', '=', ';' }, StringSplitOptions.RemoveEmptyEntries);

if (parts.Length != 3)
{
    Console.WriteLine("Invalid format. Use: <type> <name> = <value>;");
    continue;
}

string type = parts[0];
string name = parts[1];
string value = parts[2];

// Add to symbol table if valid palindrome substring exists
var result = symbolTable.AddVariable(name, type, value, lineNumber);
if (result.success)
{
    Console.WriteLine($"Added: {name} (palindrome: '{result.palindrome}')");
}
else
{
    Console.WriteLine($"Rejected: {name} (no palindrome substring ≥3)");
}
}
catch (Exception ex)
{
    Console.WriteLine($"Error: {ex.Message}");
}

lineNumber++;
}

// Display symbol table
Console.WriteLine("\nSymbol Table Contents:");
Console.WriteLine("-----");
Console.WriteLine("| Line # | Variable Name | Type   | Value   | Palindrome |");
Console.WriteLine("-----");
symbolTable.Display();
Console.WriteLine("-----");
}
}

class SymbolTable
{
    private readonly List<SymbolEntry> _entries = new List<SymbolEntry>();

    public (bool success, string palindrome) AddVariable(string name, string type, string value, int
lineNumber)
    {
        string palindrome = FindPalindromeSubstring(name, 3);
        if (palindrome != null)
        {

```



```

        _entries.Add(new SymbolEntry(name, type, value, lineNumber, palindrome));
        return (true, palindrome);
    }
    return (false, null);
}

public void Display()
{
    foreach (var entry in _entries)
    {
        Console.WriteLine($"{entry.LineNumber,-6} | {entry.Name,-13} | {entry.Type,-6} |
{entry.Value,-9} | {entry.Palindrome,-10} |");
    }
}

private string FindPalindromeSubstring(string s, int minLength)
{
    for (int len = s.Length; len >= minLength; len--)
    {
        for (int i = 0; i <= s.Length - len; i++)
        {
            string substring = s.Substring(i, len);
            if (IsPalindrome(substring))
            {
                return substring;
            }
        }
    }
    return null;
}

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

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

class SymbolEntry
{

```

```

    public string Name { get; }
    public string Type { get; }
    public string Value { get; }
    public int LineNumber { get; }
    public string Palindrome { get; }

    public SymbolEntry(string name, string type, string value, int lineNumber, string palindrome)
    {
        Name = name;
        Type = type;
        Value = value;
        LineNumber = lineNumber;
        Palindrome = palindrome;
    }
}
}

```

Output:

The screenshot shows the Programiz C# Online Compiler interface. On the left, the 'Main.cs' file is open, displaying the `SymbolEntry` class with its properties and constructor. The code is as follows:

```

123 }
124 }
125
126 class SymbolEntry
127 {
128     public string Name { get; }
129     public string Type { get; }
130     public string Value { get; }
131     public int LineNumber { get; }
132     public string Palindrome { get; }
133
134     public SymbolEntry(string name, string type, string value,
135                         int lineNumber, string palindrome)
136     {
137         Name = name;
138         Type = type;
139         Value = value;
140         LineNumber = lineNumber;
141         Palindrome = palindrome;
142     }
143 }

```

On the right, the 'Output' pane shows the results of running the program. It displays a prompt for variable declarations and the output of several test cases:

```

Symbol Table with Palindrome Check
Enter variable declarations (empty line to exit):
Format: <type> <name> = <value>; Example: int val33 = 999;
[Line 1]> int val33 = 999;
Rejected: val33 (no palindrome substring ?3)
[Line 2]> int val333 = 999;
Added: val333 (palindrome: '333')
[Line 3]> int wow = 123;
Added: wow (palindrome: 'wow')
[Line 4]> int radar = 456;
Added: radar (palindrome: 'radar')
[Line 5]> |

```

Question4:

```

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

```

```

class Program
{
    static void Main()
    {
        // Step 1: Take input from user
        Console.Write("Enter number of non-terminals: ");
        int n = int.Parse(Console.ReadLine());
    }
}

```

```

var grammar = new Dictionary<string, List<List<string>>>>();

for (int i = 0; i < n; i++)
{
    Console.WriteLine($"Enter production for non-terminal {i + 1}: ");
    string input = Console.ReadLine();
    var parts = input.Split(new[] { ">" }, StringSplitOptions.RemoveEmptyEntries);
    string lhs = parts[0].Trim();
    string[] rhsOptions = parts[1].Split('|');

    var productions = new List<List<string>>>();
    foreach (string option in rhsOptions)
    {
        List<string> symbols = option.Trim().Split(' ').ToList();
        productions.Add(symbols);
    }

    grammar[lhs] = productions;
}

var firstSets = ComputeFirstSets(grammar);
var followSets = ComputeFollowSets(grammar, firstSets);

Console.WriteLine("\nFIRST Sets:");
foreach (var kvp in firstSets)
{
    Console.WriteLine($"FIRST({kvp.Key}) = {{ {string.Join(" ", kvp.Value)} }}");
}

Console.WriteLine("\nFOLLOW Sets:");
foreach (var kvp in followSets)
{
    Console.WriteLine($"FOLLOW({kvp.Key}) = {{ {string.Join(" ", kvp.Value)} }}");
}

static Dictionary<string, HashSet<string>> ComputeFirstSets(Dictionary<string, List<List<string>>>
grammar)
{
    var first = new Dictionary<string, HashSet<string>>();

    foreach (var nt in grammar.Keys)
        first[nt] = new HashSet<string>();

    bool changed;

    do
    {
        changed = false;

```

```

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

            if (!grammar.ContainsKey(symbol)) // Terminal
            {
                if (first[nt].Add(symbol))
                    changed = true;
                break;
            }

            foreach (var f in first[symbol])
            {
                if (f != "ε" && first[nt].Add(f))
                    changed = true;
            }

            if (!first[symbol].Contains("ε"))
                break;

            if (i == production.Count - 1 && first[nt].Add("ε"))
                changed = true;
        }
    }
} while (changed);

return first;
}

static Dictionary<string, HashSet<string>> ComputeFollowSets(Dictionary<string,
List<List<string>>> grammar, Dictionary<string, HashSet<string>> firstSets)
{
    var follow = new Dictionary<string, HashSet<string>>();

    foreach (var nt in grammar.Keys)
        follow[nt] = new HashSet<string>();

    string startSymbol = grammar.Keys.First();
    follow[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 symbol = production[i];
            if (!grammar.ContainsKey(symbol))
                continue;

            bool epsilonInAll = true;

            for (int j = i + 1; j < production.Count; j++)
            {
                string next = production[j];
                if (!grammar.ContainsKey(next))
                {
                    if (follow[symbol].Add(next))
                        changed = true;
                    epsilonInAll = false;
                    break;
                }

                foreach (var f in firstSets[next])
                {
                    if (f != "ε" && follow[symbol].Add(f))
                        changed = true;
                }

                if (!firstSets[next].Contains("ε"))
                {
                    epsilonInAll = false;
                    break;
                }
            }

            if (epsilonInAll)
            {
                foreach (var f in follow[lhs])
                {
                    if (follow[symbol].Add(f))
                        changed = true;
                }
            }
        }
    }
}


} while (changed);

return follow;

```

```
}  
}
```

Output:


C# Online Compiler

Programiz PRO >

Main.cs

```
143         epsilonInAll = false;  
144         break;  
145     }  
146 }  
147  
148 if (epsilonInAll)  
149 {  
150     foreach (var f in follow[lhs])  
151     {  
152         if (follow[symbol].Add(f))  
153             changed = true;  
154     }  
155 }  
156 }  
157 }  
158 }  
159  
160 } while (changed);  
161  
162 return follow;
```

Output

```
Enter number of non-terminals: 3  
Enter production for non-terminal 1: E -> T X  
Enter production for non-terminal 2: X -> + T X | ??  
Enter production for non-terminal 3: T -> int | ( E )  
  
FIRST Sets:  
FIRST(E) = { int, ( }  
FIRST(X) = { +, ?? }  
FIRST(T) = { int, ( }  
  
FOLLOW Sets:  
FOLLOW(E) = { $, ) }  
FOLLOW(X) = { $, ) }  
FOLLOW(T) = { +, ?? }  
  
=== Code Execution Successful ===
```