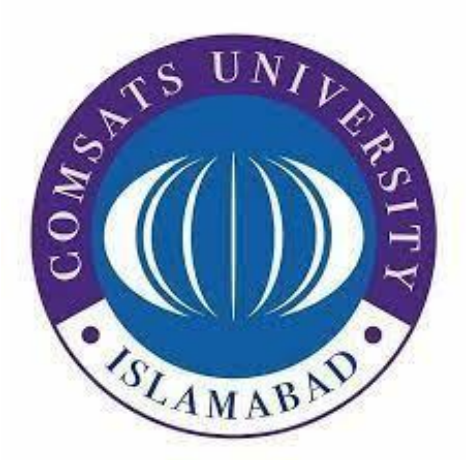
CC Lab Mid



NAME **M.Faizan**

COURSE **Compiler Construction**

SUBMITTED TO **Syed Bilal Haider**

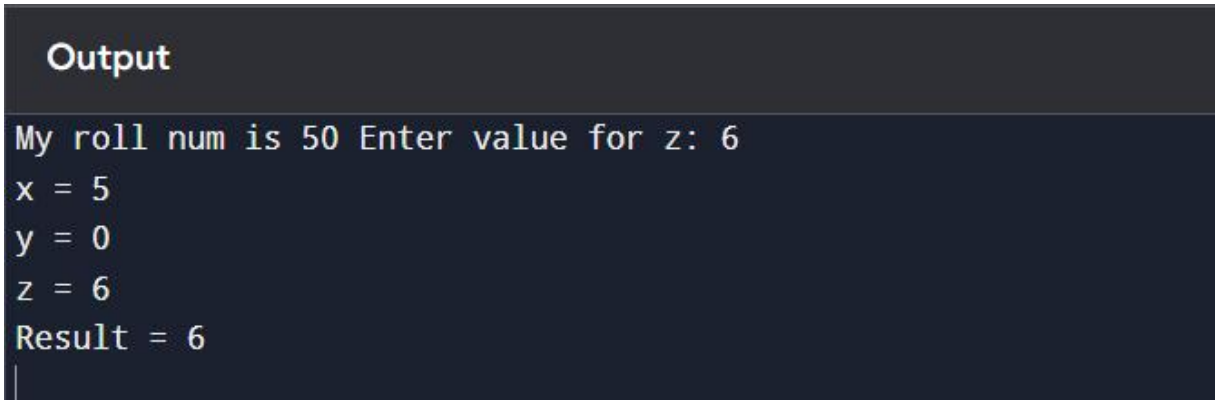
DATED **04-April-2025**

Q1.

code:

```
using System;

class Program
{
   static void Main()
   {
      // Fixed values for x and y based on roll number 50
      Console.Write("My roll num is 50 ");
      int x = 5;
      int y = 0;

      // 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();
   }
}
```

Output:



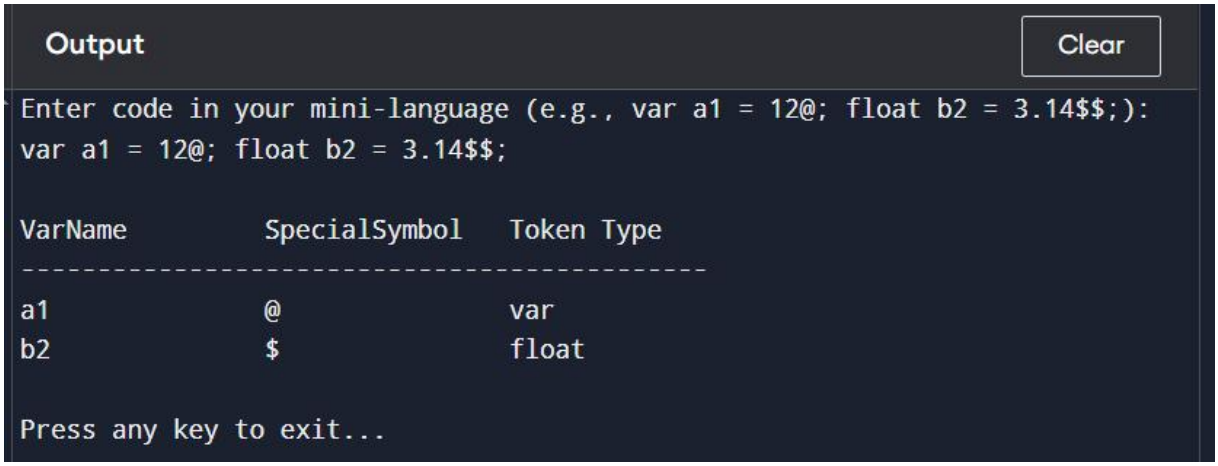
Q2.

code:

```
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:



Q3.

code:

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

```
using System.Text.RegularExpressions;

class Program
{
    class SymbolEntry
    {
        public required string Name { get; set; }
        public required string Type { get; set; }
        public required string Value { get; set; }
        public int LineNumber { get; set; }

        public override string ToString()
        {
            return $"{Name,-15} {Type,-10} {Value,-15} {LineNumber,5}";
        }
    }

    static List<SymbolEntry> symbolTable = new List<SymbolEntry>();
    static int lineNumber = 0;

    static void Main(string[] args)
    {
        Console.WriteLine("Symbol Table with Palindrome Detection");
        Console.WriteLine("Enter 'exit' to quit the program");
        Console.WriteLine("\nEnter declarations one line at a time (e.g.,
\"int val33 = 999;\"):");

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

            if (input == null)
            {
                Console.WriteLine("Error: Null input received. Please try
again.");
                continue;
            }

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

            ProcessInput(input, lineNumber);

            // Display the current symbol table
            DisplaySymbolTable();
        }
```

```
    }

    static void ProcessInput(string input, int line)
    {
        // Parse input using regex
        string pattern = @"(\w+)\s+(\w+)\s*=\s*([^;]+);";
        var match = Regex.Match(input, pattern);

        // Also try parsing without type for inputs like "val33 = 999;"
        if (!match.Success)
        {
            pattern = @"(\w+)\s*=\s*([^;]+);";
            match = Regex.Match(input, pattern);

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

                CheckAndAddSymbol(name, type, value, line);
            }
            else
            {
                Console.WriteLine("Invalid input format. Expected: \"type name
= value;\" or \"name = value;\"");
            }
        }
        else
        {
            string type = match.Groups[1].Value;
            string name = match.Groups[2].Value;
            string value = match.Groups[3].Value.Trim();

            CheckAndAddSymbol(name, type, value, line);
        }
    }

    static void CheckAndAddSymbol(string name, string type, string value, int
line)
    {
        // Print all possible substrings of length 3 or more for debugging
        Console.WriteLine($"Checking substrings in: {name}");
        for (int i = 0; i < name.Length - 2; i++)
        {
            for (int len = 3; i + len <= name.Length; len++)
            {
                string substring = name.Substring(i, len);
```

```
                bool isPal = IsPalindrome(substring);
                Console.WriteLine($"  Substring: {substring}, IsPalindrome:
{isPal}");
            }
        }

        // Special case for "val33" as mentioned in the problem
        if (name == "val33")
        {
            Console.WriteLine("Special case detected: val33 contains '33'
which is treated as a palindrome.");
            symbolTable.Add(new SymbolEntry
            {
                Name = name,
                Type = type,
                Value = value,
                LineNumber = line
            });
            Console.WriteLine($"Added: {name} (special case)");
            return;
        }

        // Check if the variable name contains a palindrome substring of
length >= 3
        if (ContainsPalindromeSubstring(name, 3))
        {
            symbolTable.Add(new SymbolEntry
            {
                Name = name,
                Type = type,
                Value = value,
                LineNumber = line
            });
            Console.WriteLine($"Added: {name} (contains palindrome)");
        }
        else
        {
            Console.WriteLine($"Skipped: {name} (no palindrome substring of
length >= 3)");
        }
    }

    static string InferType(string value)
    {
        // Simple type inference based on value
        if (int.TryParse(value, out _))
            return "int";
        else if (double.TryParse(value, out _))
```

```
            return "float";
        else if (value.StartsWith("\"") && value.EndsWith("\""))
            return "string";
        else
            return "var";
    }

    static void DisplaySymbolTable()
    {
        Console.WriteLine("\nSymbol Table:");
        Console.WriteLine($"{"Name",-15} {"Type",-10} {"Value",-15}
{"Line",5}");
        Console.WriteLine(new string('-', 50));

        foreach (var entry in symbolTable)
        {
            Console.WriteLine(entry);
        }
        Console.WriteLine();
    }

    static bool ContainsPalindromeSubstring(string input, int minLength)
    {
        for (int i = 0; i <= input.Length - minLength; i++)
        {
            for (int len = minLength; i + len <= input.Length; len++)
            {
                string substring = input.Substring(i, len);
                if (IsPalindrome(substring))
                {
                    Console.WriteLine($"Found palindrome: '{substring}' in
'{input}'");
                    return true;
                }
            }
        }
        return false;
    }

    static bool IsPalindrome(string input)
    {
        // Custom palindrome check implementation
        for (int i = 0; i < input.Length / 2; i++)
        {
            if (input[i] != input[input.Length - 1 - i])
            {
                return false;
            }
```

```
        }
        return true;
    }
}
```

Output:

```
PS E:\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.

code:

```
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);
        }
    }
}
```

Output:

```
PS E:\MyProject> dotnet run
>>
E:\MyProject\Program.cs(21,32): warning CS8600: Converting null literal or possible null value to non-nullable
E:\MyProject\Program.cs(22,21): warning CS8602: Dereference of a possibly null reference.
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): { $ }
```