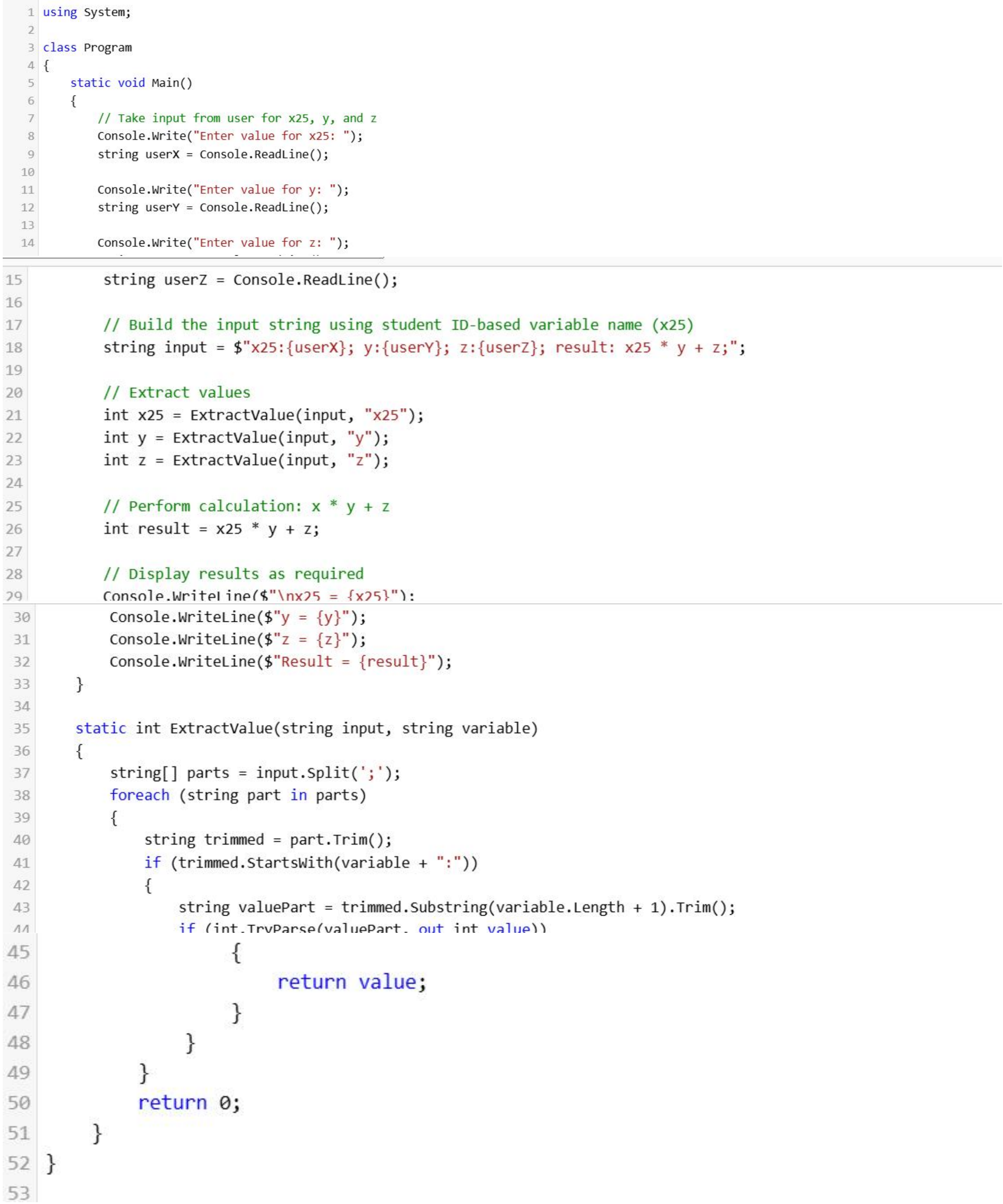
COMSATS UNIVERSITY ISLAMABAD ATTOCK CAMPUS

LAB MID

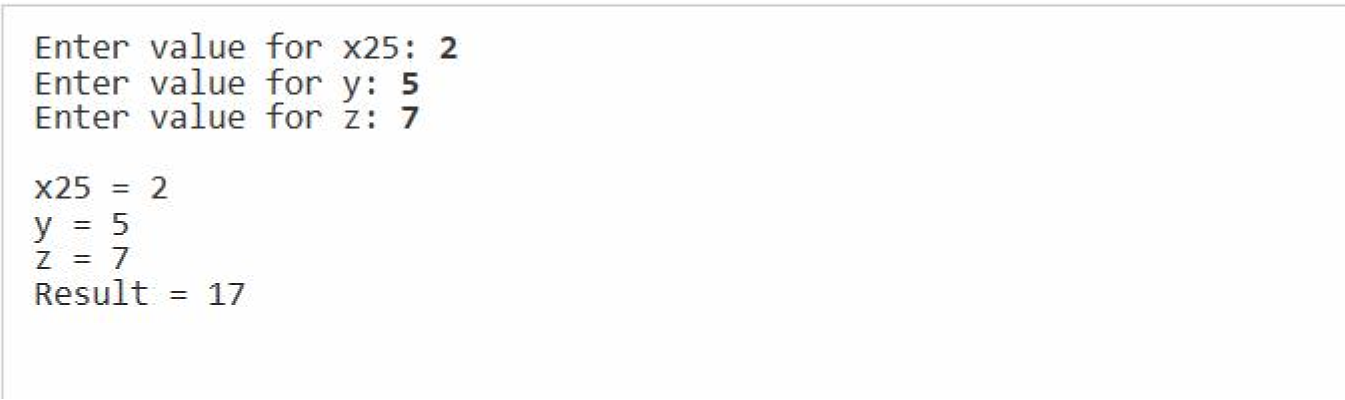
NAME:
SARA ARSHAD
REG.NO:
SP22-BCS-025
SUBMITTED TO:
SIR BILAL BUKHARI
DATE:
11-4-2025

Q1:



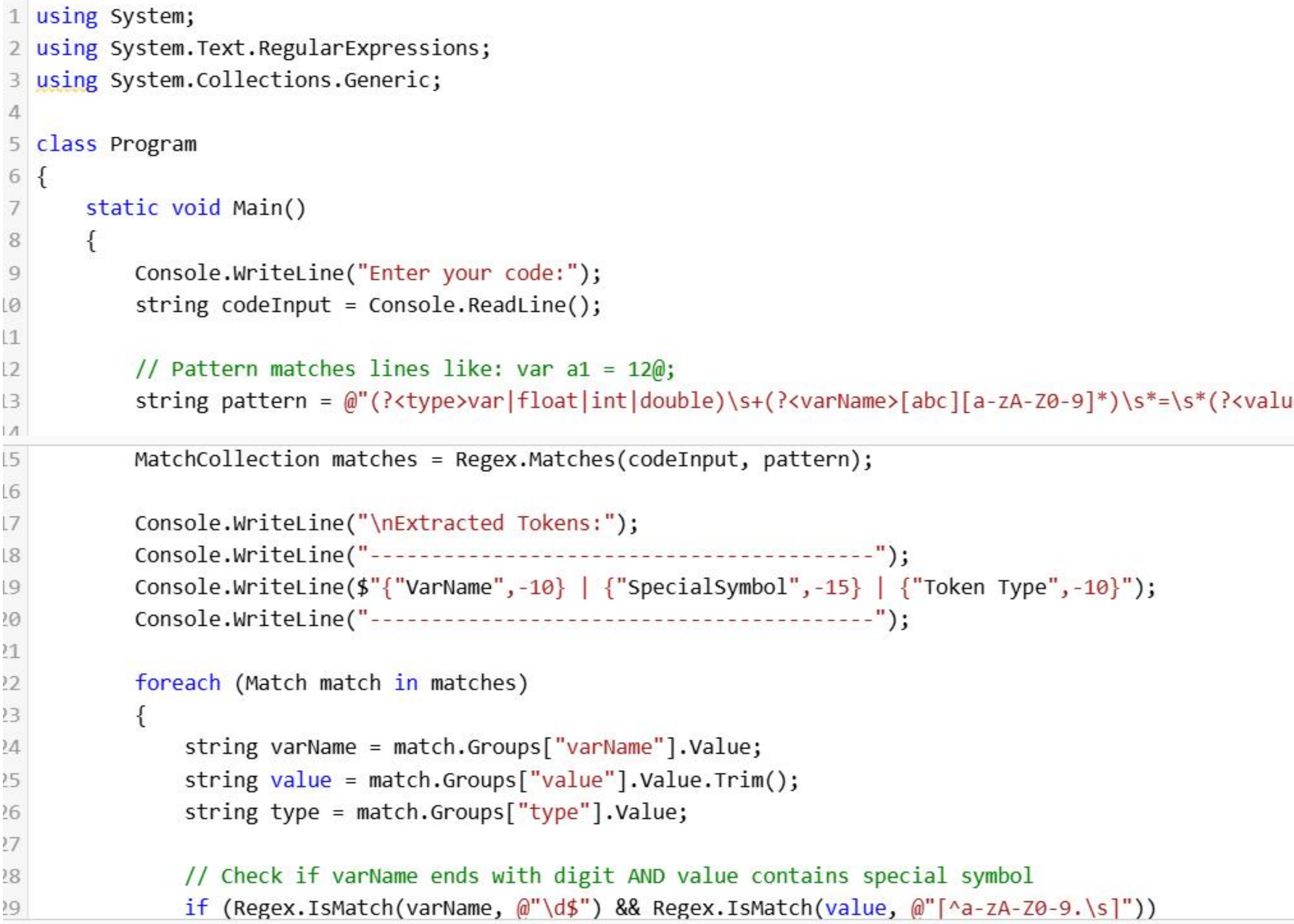
```
using System;

class Program
{
    static void Main()
    {
        // Take input from user for x25, y, and z
        Console.Write("Enter value for x25: ");
        string userX = Console.ReadLine();

        Console.Write("Enter value for y: ");
        string userY = Console.ReadLine();

        Console.Write("Enter value for z: ");
        string userZ = Console.ReadLine();

        // Build the input string using student ID-based variable name (x25)
        string input = $"x25:{userX}; y:{userY}; z:{userZ}; result: x25 * y + z;";

        // Extract values
        int x25 = ExtractValue(input, "x25");
        int y = ExtractValue(input, "y");
        int z = ExtractValue(input, "z");

        // Perform calculation: x * y + z
        int result = x25 * y + z;

        // Display results as required
        Console.WriteLine($"\nx25 = {x25}");
        Console.WriteLine($"y = {y}");
        Console.WriteLine($"z = {z}");
        Console.WriteLine($"Result = {result}");
    }

    static int ExtractValue(string input, string variable)
    {
        string[] parts = input.Split(';');
        foreach (string part in parts)
        {
            string trimmed = part.Trim();
            if (trimmed.StartsWith(variable + ":"))
            {
                string valuePart = trimmed.Substring(variable.Length + 1).Trim();
                if (int.TryParse(valuePart, out int value))
                {
                    return value;
                }
            }
        }
        return 0;
    }
}
```

Output:

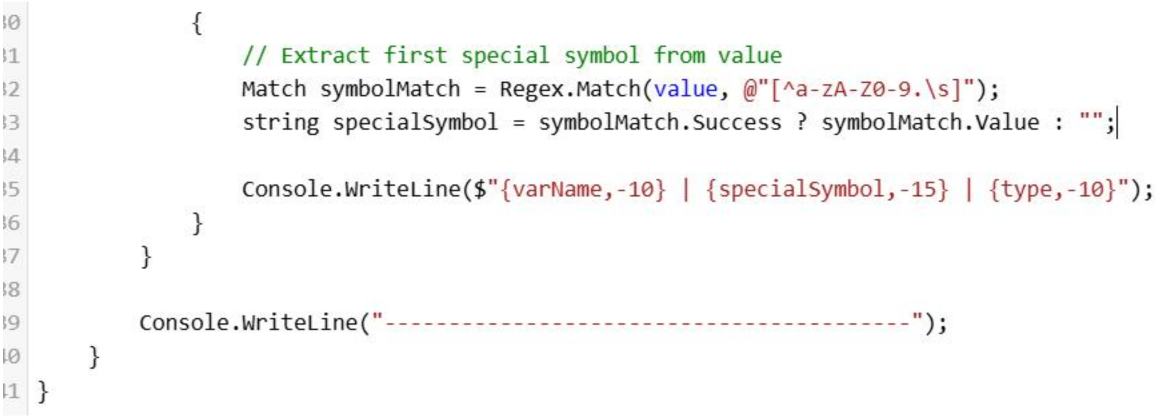


```
Enter value for x25: 2
Enter value for y: 5
Enter value for z: 7

x25 = 2
y = 5
z = 7
Result = 17
```

Q2:

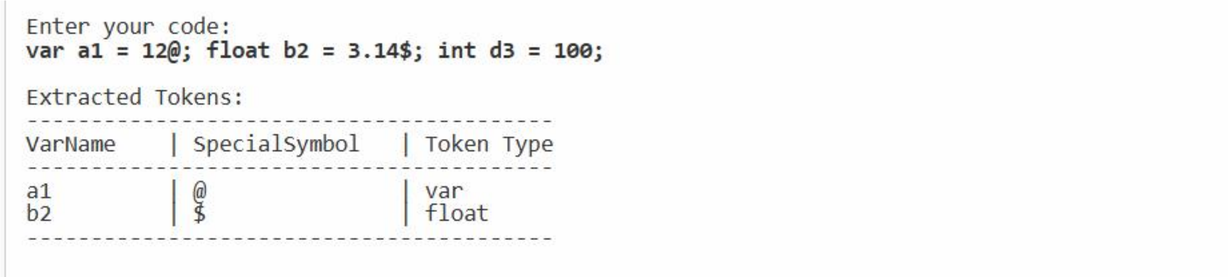


```csharp
using System;
using System.Text.RegularExpressions;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        Console.WriteLine("Enter your code:");
        string codeInput = Console.ReadLine();

        // Pattern matches lines like: var a1 = 12@;
        string pattern = @"(?<type>var|float|int|double)\s+(?<varName>[abc][a-zA-Z0-9]*)\s*=\s*(?<valu

        MatchCollection matches = Regex.Matches(codeInput, pattern);

        Console.WriteLine("\nExtracted Tokens:");
        Console.WriteLine("-----------------------------------------");
        Console.WriteLine($"{"VarName",-10} | {"SpecialSymbol",-15} | {"Token Type",-10}");
        Console.WriteLine("-----------------------------------------");

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

            // Check if varName ends with digit AND value contains special symbol
            if (Regex.IsMatch(varName, @"\d$") && Regex.IsMatch(value, @"[^a-zA-Z0-9.\s]"))
```



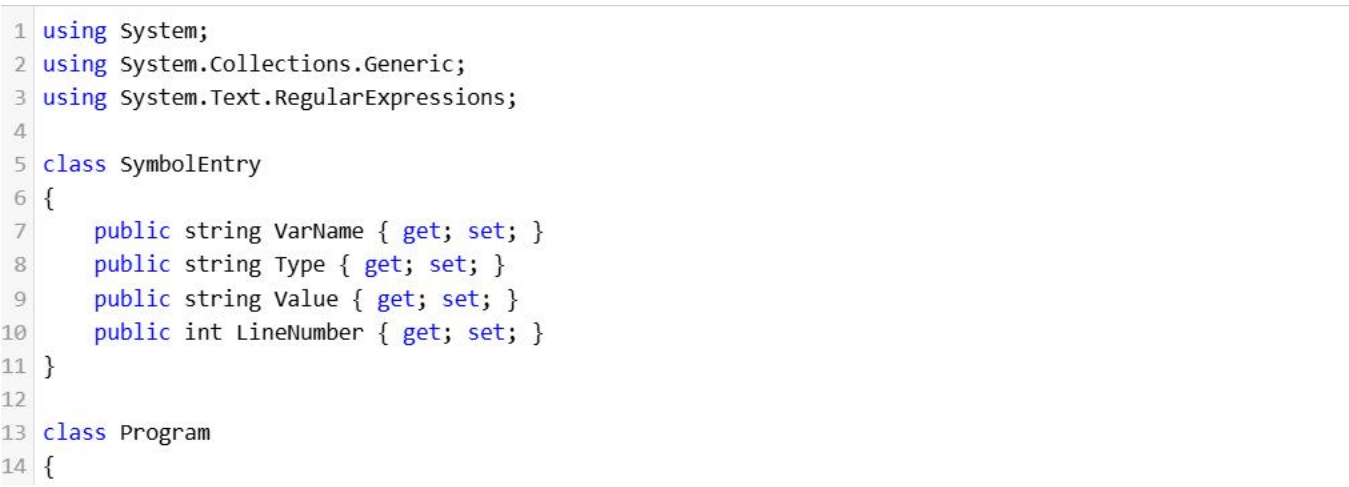
```
            {
                // Extract first special symbol from value
                Match symbolMatch = Regex.Match(value, @"[^a-zA-Z0-9.\s]");
                string specialSymbol = symbolMatch.Success ? symbolMatch.Value : "";

                Console.WriteLine($"{varName,-10} | {specialSymbol,-15} | {type,-10}");
            }
        }

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

OUTPUT:

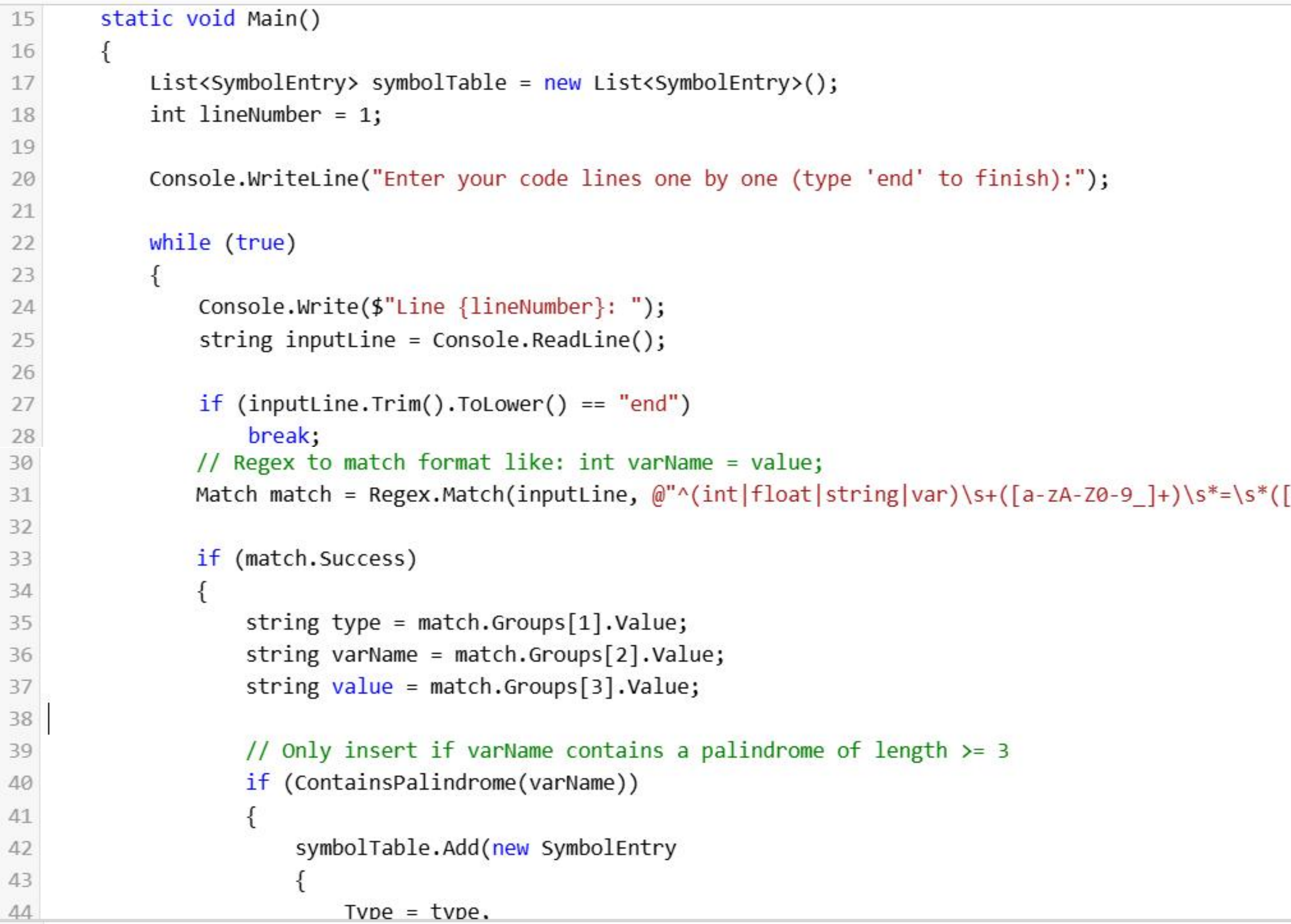


```
Enter your code:
var a1 = 12@; float b2 = 3.14$; int d3 = 100;

Extracted Tokens:
-----------------------------------------
VarName    | SpecialSymbol   | Token Type
-----------------------------------------
a1         | @               | var
b2         | $               | float
-----------------------------------------
```

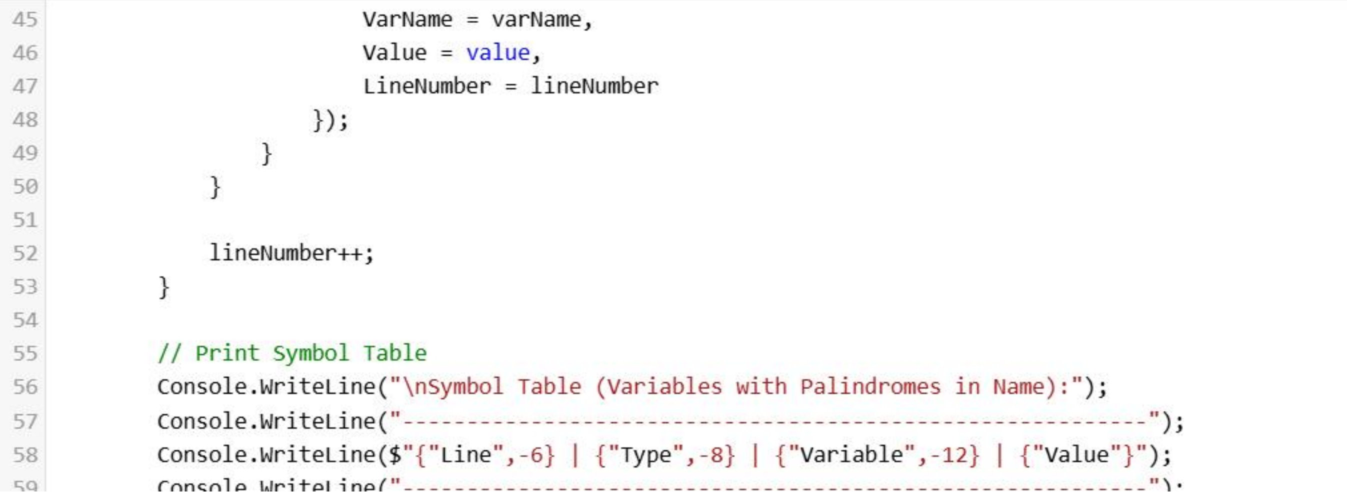
Q3:



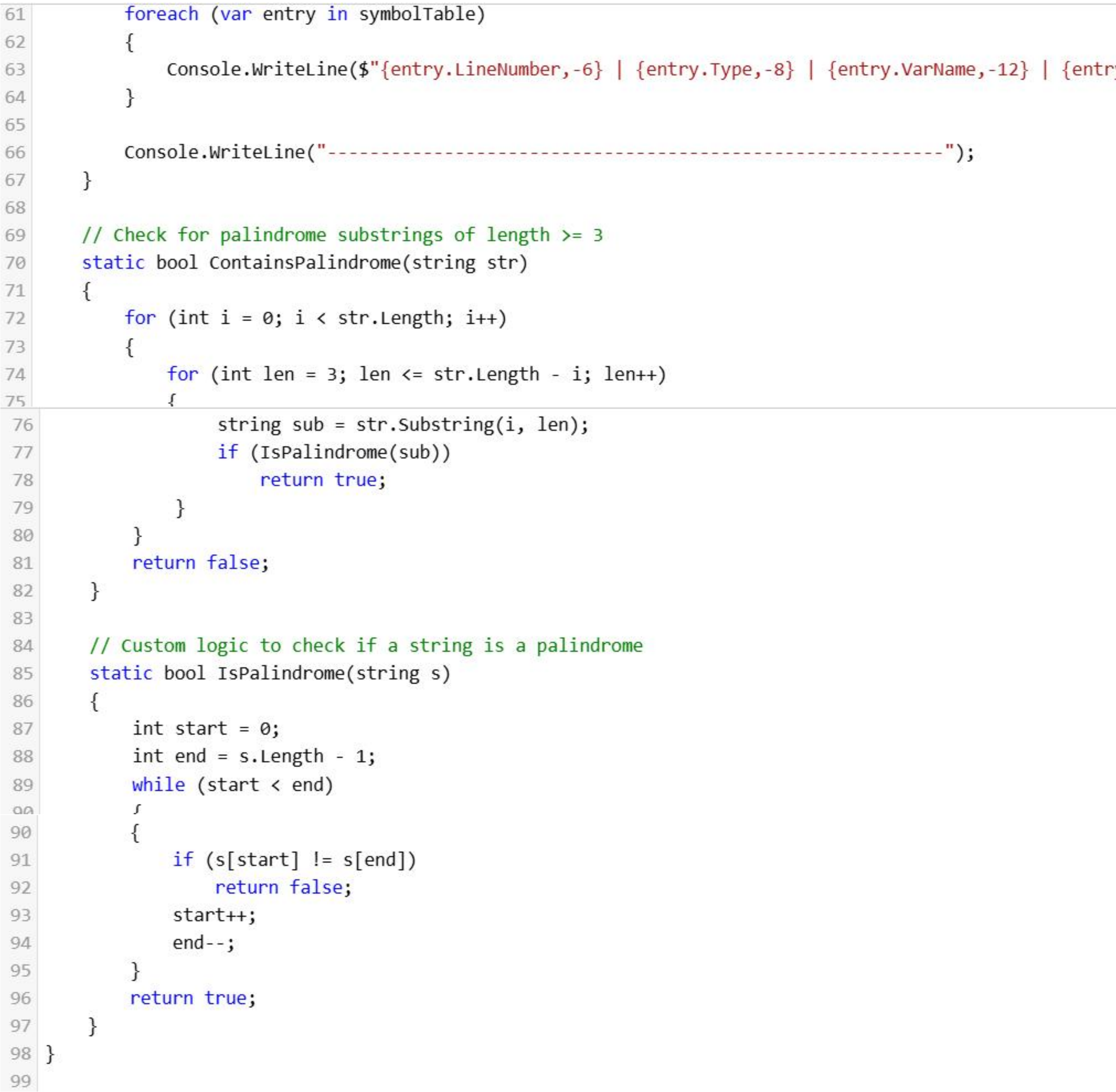
```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

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

class Program
{
```



```
    static void Main()
    {
        List<SymbolEntry> symbolTable = new List<SymbolEntry>();
        int lineNumber = 1;

        Console.WriteLine("Enter your code lines one by one (type 'end' to finish):");

        while (true)
        {
            Console.Write($"Line {lineNumber}: ");
            string inputLine = Console.ReadLine();

            if (inputLine.Trim().ToLower() == "end")
                break;
            // Regex to match format like: int varName = value;
            Match match = Regex.Match(inputLine, @"^(int|float|string|var)\s+([a-zA-Z0-9_]+)\s*=\s*([

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

                // Only insert if varName contains a palindrome of length >= 3
                if (ContainsPalindrome(varName))
                {
                    symbolTable.Add(new SymbolEntry
                    {
                        Type = type,
```

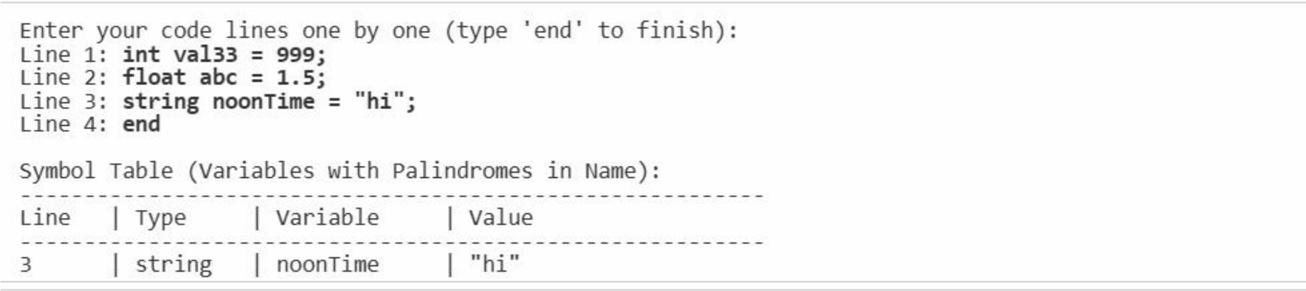


```
                        VarName = varName,
                        Value = value,
                        LineNumber = lineNumber
                    });
                }
            }

            lineNumber++;
        }

        // Print Symbol Table
        Console.WriteLine("\nSymbol Table (Variables with Palindromes in Name):");
        Console.WriteLine("----------------------------------------------------------");
        Console.WriteLine($"{"Line",-6} | {"Type",-8} | {"Variable",-12} | {"Value"}");
        Console.WriteLine("----------------------------------------------------------");
```



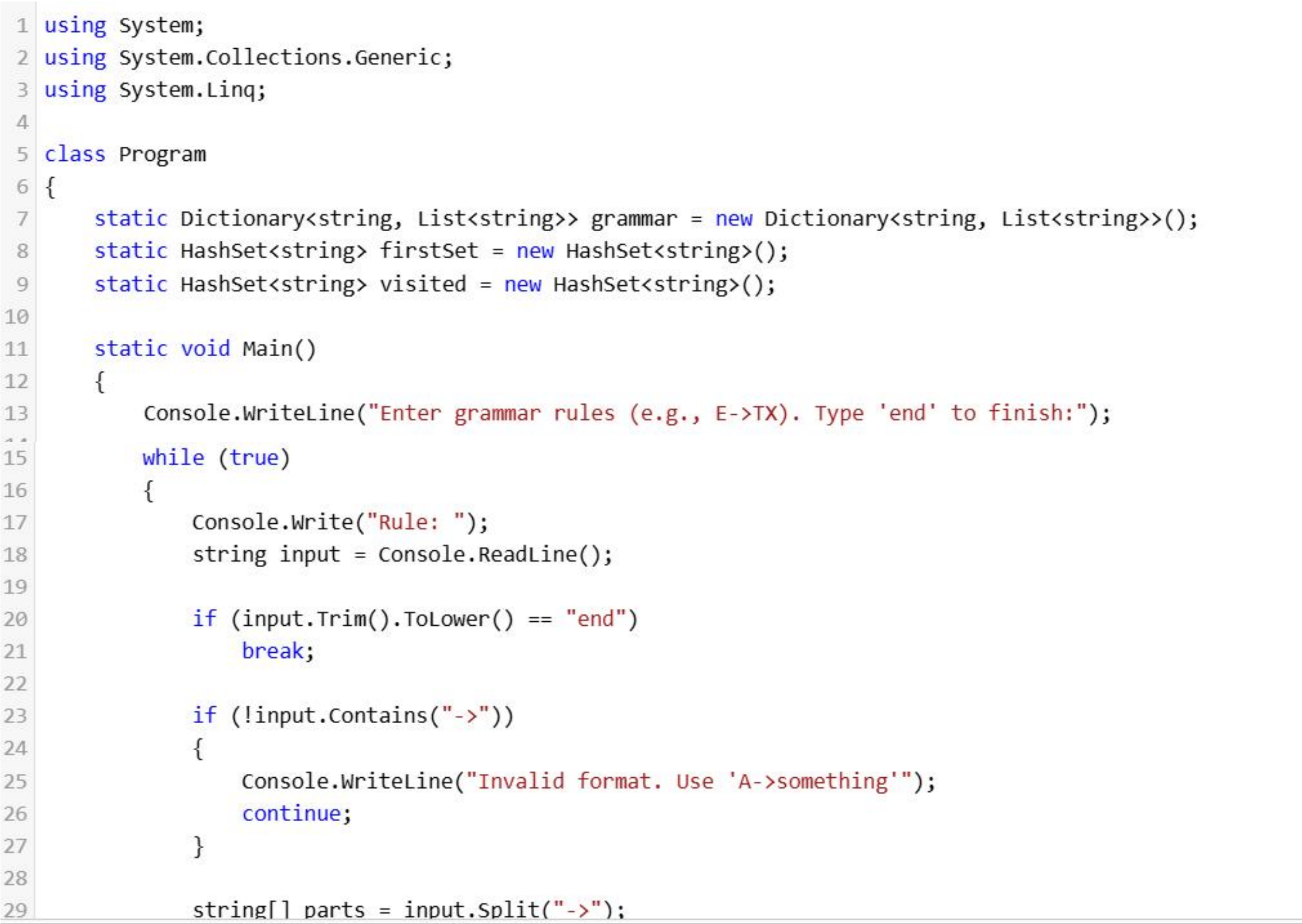
```
        foreach (var entry in symbolTable)
        {
            Console.WriteLine($"{entry.LineNumber,-6} | {entry.Type,-8} | {entry.VarName,-12} | {entr
        }

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

    // Check for palindrome substrings of length >= 3
    static bool ContainsPalindrome(string str)
    {
        for (int i = 0; i < str.Length; i++)
        {
            for (int len = 3; len <= str.Length - i; len++)
            {
                string sub = str.Substring(i, len);
                if (IsPalindrome(sub))
                    return true;
            }
        }
        return false;
    }

    // Custom logic to check if a string is a palindrome
    static bool IsPalindrome(string s)
    {
        int start = 0;
        int end = s.Length - 1;
        while (start < end)
        {
            if (s[start] != s[end])
                return false;
            start++;
            end--;
        }
        return true;
    }
}
```

Output:

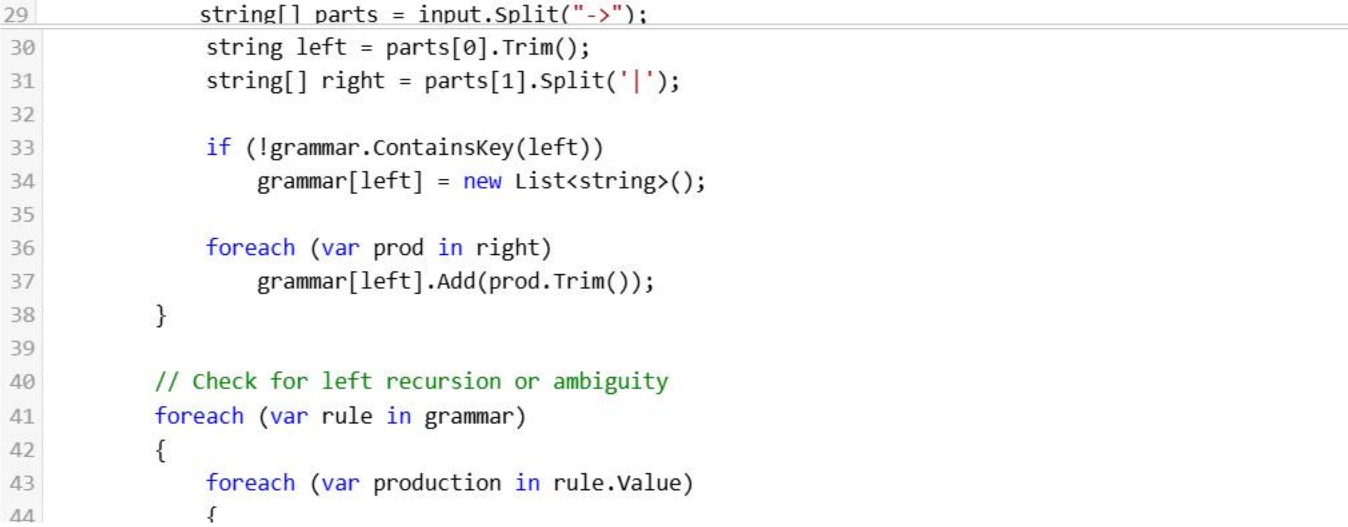


```
Enter your code lines one by one (type 'end' to finish):
Line 1: int val33 = 999;
Line 2: float abc = 1.5;
Line 3: string noonTime = "hi";
Line 4: end

Symbol Table (Variables with Palindromes in Name):
-----------------------------------------------------------
Line   | Type     | Variable     | Value
-----------------------------------------------------------
3      | string   | noonTime     | "hi"
```

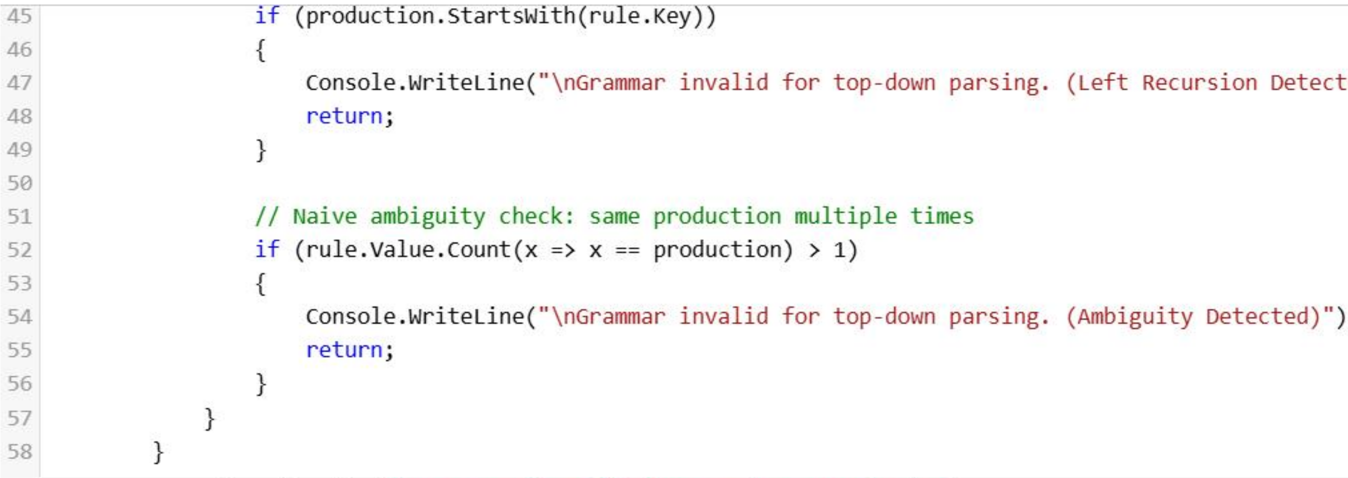
Q4:



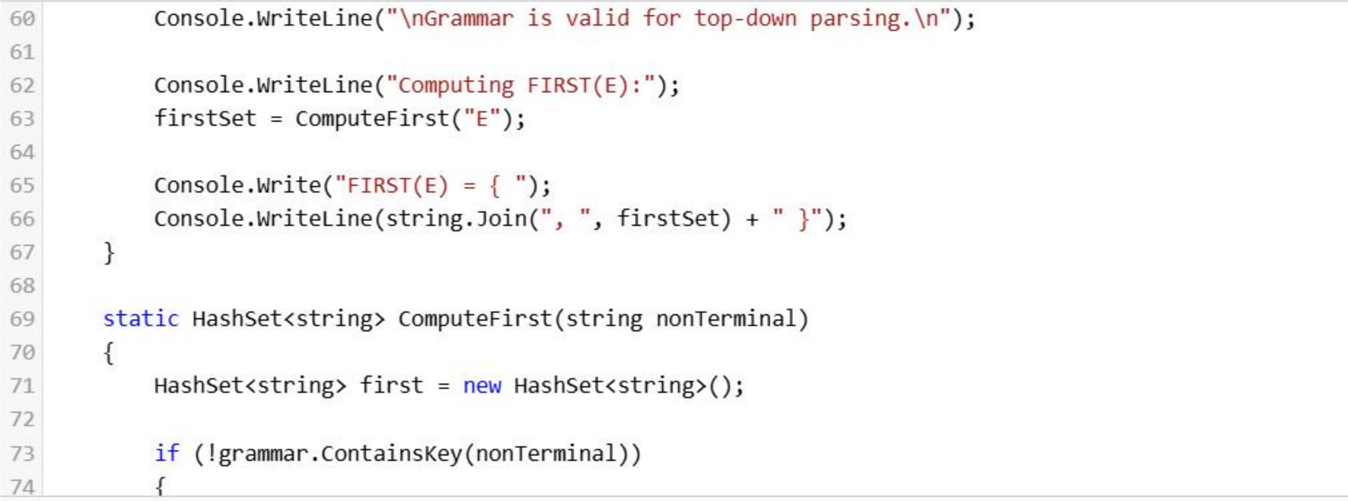
```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static Dictionary<string, List<string>> grammar = new Dictionary<string, List<string>>();
    static HashSet<string> firstSet = new HashSet<string>();
    static HashSet<string> visited = new HashSet<string>();

    static void Main()
    {
        Console.WriteLine("Enter grammar rules (e.g., E->TX). Type 'end' to finish:");
        while (true)
        {
            Console.Write("Rule: ");
            string input = Console.ReadLine();

            if (input.Trim().ToLower() == "end")
                break;

            if (!input.Contains("->"))
            {
                Console.WriteLine("Invalid format. Use 'A->something'");
                continue;
            }

            string[] parts = input.Split("->");
```



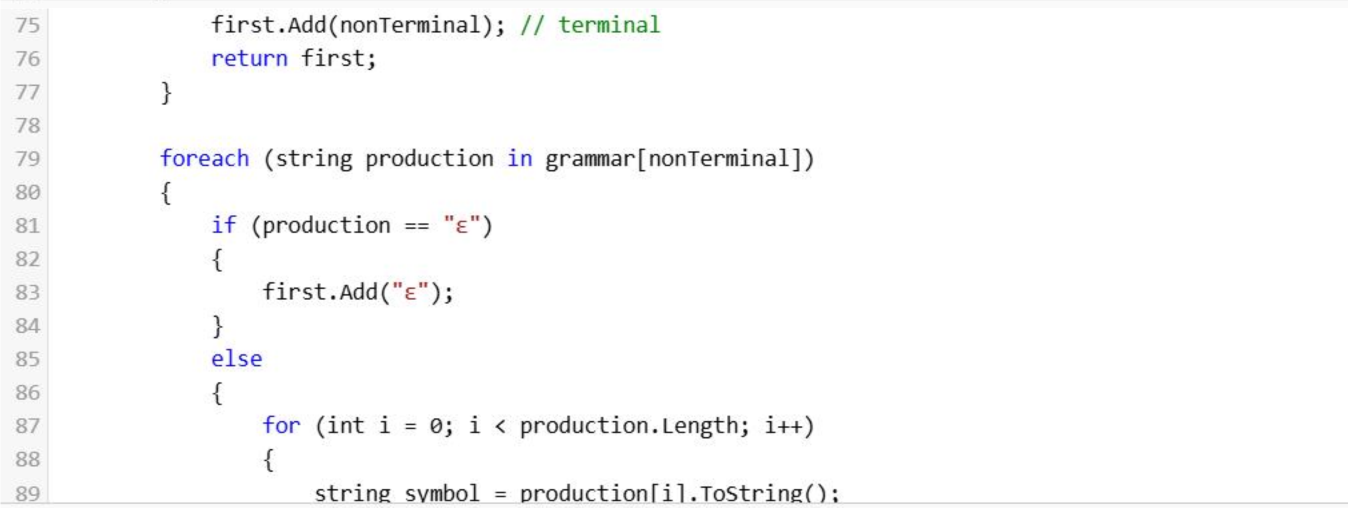
```
            string left = parts[0].Trim();
            string[] right = parts[1].Split('|');

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

            foreach (var prod in right)
                grammar[left].Add(prod.Trim());
        }

        // Check for left recursion or ambiguity
        foreach (var rule in grammar)
        {
            foreach (var production in rule.Value)
            {
```



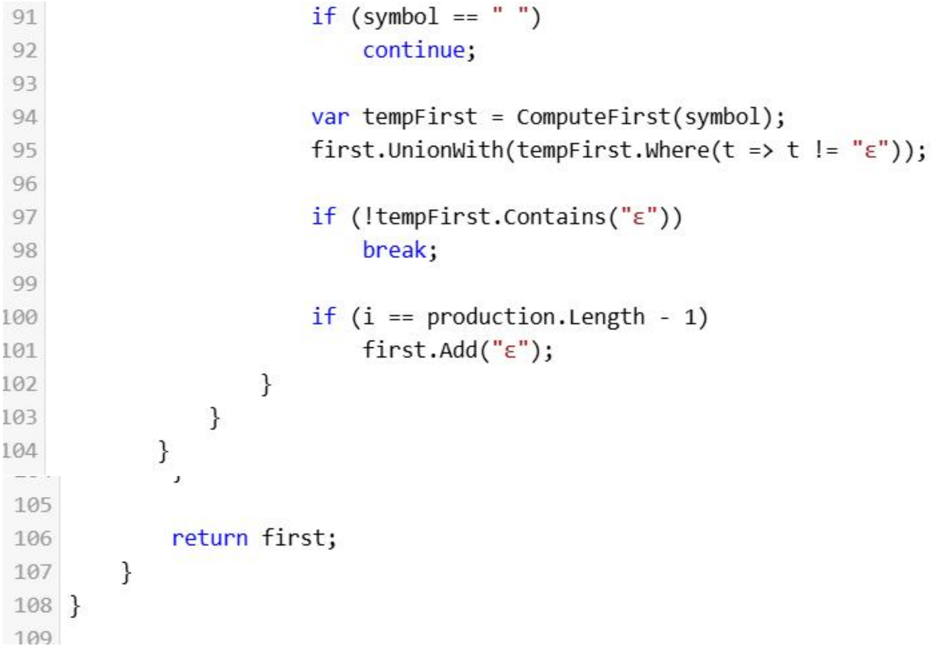
```
                if (production.StartsWith(rule.Key))
                {
                    Console.WriteLine("\nGrammar invalid for top-down parsing. (Left Recursion Detect
                    return;
                }

                // Naive ambiguity check: same production multiple times
                if (rule.Value.Count(x => x == production) > 1)
                {
                    Console.WriteLine("\nGrammar invalid for top-down parsing. (Ambiguity Detected)")
                    return;
                }
            }
        }
```



```
        Console.WriteLine("\nGrammar is valid for top-down parsing.\n");

        Console.WriteLine("Computing FIRST(E):");
        firstSet = ComputeFirst("E");

        Console.Write("FIRST(E) = { ");
        Console.WriteLine(string.Join(", ", firstSet) + " }");
    }

    static HashSet<string> ComputeFirst(string nonTerminal)
    {
        HashSet<string> first = new HashSet<string>();

        if (!grammar.ContainsKey(nonTerminal))
        {
```

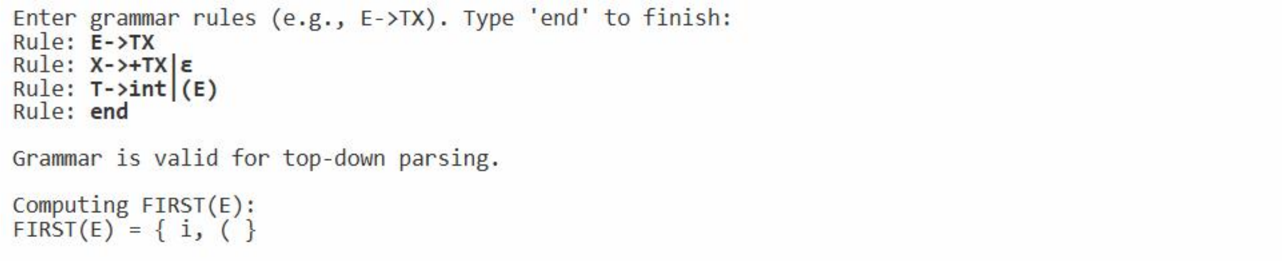


```
            first.Add(nonTerminal); // terminal
            return first;
        }

        foreach (string production in grammar[nonTerminal])
        {
            if (production == "ε")
            {
                first.Add("ε");
            }
            else
            {
                for (int i = 0; i < production.Length; i++)
                {
                    string symbol = production[i].ToString();
```



```
                    if (symbol == " ")
                        continue;

                    var tempFirst = ComputeFirst(symbol);
                    first.UnionWith(tempFirst.Where(t => t != "ε"));

                    if (!tempFirst.Contains("ε"))
                        break;

                    if (i == production.Length - 1)
                        first.Add("ε");
                }
            }
        }

        return first;
    }
}
```

Output:



```
Enter grammar rules (e.g., E->TX). Type 'end' to finish:
Rule: E->TX
Rule: X->+TX|ε
Rule: T->int|(E)
Rule: end

Grammar is valid for top-down parsing.

Computing FIRST(E):
FIRST(E) = { i, ( }
```