# **COMSATS UNIVERSITY ISLAMABAD**

*ATTOCK CAMPUS*



Submitted To

Sir Bilal Haider Bukhari

Submitted By

MUHAMMAD UMAR FAROOQ

(SP22-BCS-040)

11th April, 2025

**Question 4:**

using System;

using System.Collections.Generic;

using System.Linq;

namespace GrammarAnalyzer

{

    class Program

    {

        // *Dictionary to hold the grammar rules.*

        // *Key: Non-terminal (like E), Value: List of productions (each as list of strings)*

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

        static void Main(string[] args)

        {

            Console.WriteLine("Enter grammar rules (format: A->a B | ε). Enter 'done' to finish:");

            // *Reading grammar rules from the user until they type 'done'*

            while (true)

            {

                Console.Write("> ");

                string? input = Console.ReadLine();

                if (string.IsNullOrEmpty(input)) continue;

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

                // *Check if rule format is valid*

                if (!input.Contains("->"))

                {

                    Console.WriteLine("Invalid format. Use A->B C | d");

                    continue;

                }

                var parts = input.Split("->");

                string lhs = parts[0].Trim(); // *Left-hand side non-terminal*

                var rhs = parts[1].Split('|')

                                  .Select(p => p.Trim().Split(' ').ToList()) // *Split each production into symbols*

                                  .ToList();

                // *Initialize grammar entry if not already present*

                if (!grammar.ContainsKey(lhs))

                    grammar[lhs] = new List<List<string>>();

                // *Loop through each production and add to the grammar*

                foreach (var prod in rhs)

                {

                    // *Check for ambiguous duplicate production*

                    if (grammar[lhs].Any(existing => existing.SequenceEqual(prod)))

                    {

                        Console.WriteLine("Grammar invalid for top-down parsing. (Ambiguity found)");

                        return;

                    }

                    // *Check for left recursion: e.g. A -> A...*

                    if (prod[0] == lhs)

                    {

                        Console.WriteLine("Grammar invalid for top-down parsing. (Left recursion found)");

                        return;

                    }

                    grammar[lhs].Add(prod); // *Add valid production*

                }

            }

            // *Must have a rule for E to compute FIRST(E)*

            if (!grammar.ContainsKey("E"))

            {

                Console.WriteLine("No rule defined for E.");

                return;

            }

            // *Compute and display the FIRST set for non-terminal E*

            Console.WriteLine("\nComputing FIRST(E)...");

            var firstE = ComputeFirst("E");

            Console.WriteLine("FIRST(E): { " + string.Join(", ", firstE) + " }");

        }

        // *Recursive function to compute FIRST of a symbol*

        static HashSet<string> ComputeFirst(string symbol)

        {

            HashSet<string> result = new HashSet<string>();

            // *Base case: If it's a terminal symbol*

            if (!grammar.ContainsKey(symbol))

            {

                result.Add(symbol); // *Add terminal to FIRST*

                return result;

            }

            // *Process each production of the non-terminal*

            foreach (var production in grammar[symbol])

            {

                // *Handle epsilon productions using common variants*

                if (production[0] == "ε" || production[0] == "e" || production[0] == "eps")

                {

                    result.Add("ε");

                    continue;

                }

                // *For each symbol in the production, compute its FIRST*

                foreach (var sym in production)

                {

                    var firstOfSym = ComputeFirst(sym);

                    // *Add everything except ε*

                    result.UnionWith(firstOfSym.Where(s => s != "ε"));

                    // *If ε not in FIRST(sym), stop processing this production*

                    if (!firstOfSym.Contains("ε"))

                        break;

                    // *If it's the last symbol and all before had ε, then ε is in FIRST of this production*

                    else if (sym == production.Last())

                        result.Add("ε");

                }

            }

            return result;

        }

    }

}

**Output:**

