**CC Lab Mid**



**NAME M.Faizan**

**COURSE Compiler Construction**

**SUBMITTED TO Syed Bilal Haider**

**DATED 04-April-2025**

**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:**

