COMSATS UNIVERSITY ISLAMABAD

Mid Term Lab

Name:

Moazzam Azam

Registration:

SP22-BCS-010

Submitted To:

Sir Bilal Haider

Subject:

Compiler Construction

Date:

April 11th, 202

Question 4

Given the grammar:

$E \rightarrow T\ X$
$X \rightarrow +\ T\ X \mid \varepsilon$
$T \rightarrow int \mid (\ E\ )$

Write a C# function to compute the FIRST and follow sets.

- The rules must be input by user at runtime (via Console).
- Only compute FIRST of E after validating that the grammar has no left recursion.
    - If left recursion or any ambiguity is found in any rule, halt and print.
      "Grammar invalid for top-down parsing."

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

class GrammarAnalyzer
{
    static void Main()
    {
        Console.WriteLine("Grammar FIRST and FOLLOW Set Calculator");
        Console.WriteLine("Enter grammar rules (e.g., 'E → T X'), one per line. Enter 'done' when finished.");

        Dictionary<string, List<string>> grammar = new Dictionary<string, List<string>>();
        HashSet<string> nonTerminals = new HashSet<string>();
        HashSet<string> terminals = new HashSet<string>();

        // Input grammar rules
        while (true)
        {
            Console.Write("> ");
            string input = Console.ReadLine().Trim();
            if (input.ToLower() == "done") break;

            if (!input.Contains("→"))
            {
                Console.WriteLine("Invalid format. Use: 'NonTerminal → Production'");
                continue;
            }

            string[] parts = input.Split(new[] { "→" }, StringSplitOptions.RemoveEmptyEntries);
            string nonTerminal = parts[0].Trim();
            string production = parts[1].Trim();

            if (!grammar.ContainsKey(nonTerminal))
            {
                grammar[nonTerminal] = new List<string>();
            }
            grammar[nonTerminal].Add(production);
            nonTerminals.Add(nonTerminal);

            // Collect terminals
            foreach (char c in production)
            {
                if (char.IsLower(c) || !char.IsLetter(c))
                {
                    terminals.Add(c.ToString());
                }
```

```
        }
    }

    // Remove non-terminals from terminals
    foreach (var nt in nonTerminals)
    {
        terminals.Remove(nt);
    }

    // Check for left recursion
    if (HasLeftRecursion(grammar))
    {
        Console.WriteLine("Grammar invalid for top-down parsing.");
        return;
    }

    // Compute FIRST sets
    Dictionary<string, HashSet<string>> firstSets = ComputeFirstSets(grammar, nonTerminals, terminals);

    // Compute FOLLOW sets
    Dictionary<string, HashSet<string>> followSets = ComputeFollowSets(grammar, nonTerminals, terminals, firstSets);

    // Display results
    Console.WriteLine("\nFIRST Sets:");
    foreach (var nt in nonTerminals.OrderBy(x => x))
    {
        Console.WriteLine($"FIRST({nt}) = {{ {string.Join(", ", firstSets[nt])} }}");
    }

    Console.WriteLine("\nFOLLOW Sets:");
    foreach (var nt in nonTerminals.OrderBy(x => x))
    {
        Console.WriteLine($"FOLLOW({nt}) = {{ {string.Join(", ", followSets[nt])} }}");
    }
}

static bool HasLeftRecursion(Dictionary<string, List<string>> grammar)
{
    foreach (var rule in grammar)
    {
        string nonTerminal = rule.Key;
```

```
    {
        string nonTerminal = rule.Key;
        foreach (string production in rule.Value)
        {
            string[] symbols = production.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
            if (symbols.Length > 0 && symbols[0] == nonTerminal)
            {
                return true;
            }
        }
    }
    return false;
}

static Dictionary<string, HashSet<string>> ComputeFirstSets(
    Dictionary<string, List<string>> grammar,
    HashSet<string> nonTerminals,
    HashSet<string> terminals)
{
    Dictionary<string, HashSet<string>> firstSets = new Dictionary<string, HashSet<string>>();

    // Initialize FIRST sets
    foreach (string nt in nonTerminals)
    {
        firstSets[nt] = new HashSet<string>();
    }

    bool changed;
    do
    {
        changed = false;
        foreach (var rule in grammar)
        {
            string nonTerminal = rule.Key;
            foreach (string production in rule.Value)
            {
                string[] symbols = production.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
                if (symbols.Length == 0) // ε production
                {
                    if (firstSets[nonTerminal].Add("ε"))
                        changed = true;
                    continue;
                }
```

```
                bool allHaveEpsilon = true;
                foreach (string symbol in symbols)
                {
                    if (terminals.Contains(symbol))
                    {
                        if (firstSets[nonTerminal].Add(symbol))
                            changed = true;
                        allHaveEpsilon = false;
                        break;
                    }
                    else if (nonTerminals.Contains(symbol))
                    {
                        int countBefore = firstSets[nonTerminal].Count;
                        firstSets[nonTerminal].UnionWith(firstSets[symbol].Except(new[] { "ε" }));
                        if (firstSets[nonTerminal].Count > countBefore)
                            changed = true;

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

                if (allHaveEpsilon && firstSets[nonTerminal].Add("ε"))
                    changed = true;
            }
        }
    } while (changed);

    return firstSets;
}

static Dictionary<string, HashSet<string>> ComputeFollowSets(
    Dictionary<string, List<string>> grammar,
    HashSet<string> nonTerminals,
    HashSet<string> terminals,
    Dictionary<string, HashSet<string>> firstSets)
```

```
{
    Dictionary<string, HashSet<string>> followSets = new Dictionary<string, HashSet<string>>();
    foreach (string nt in nonTerminals)
    {
        followSets[nt] = new HashSet<string>();
    }
    followSets[grammar.Keys.First()].Add("$"); // Add $ to start symbol

    bool changed;
    do
    {
        changed = false;
        foreach (var rule in grammar)
        {
            string nonTerminal = rule.Key;
            foreach (string production in rule.Value)
            {
                string[] symbols = production.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
                for (int i = 0; i < symbols.Length; i++)
                {
                    if (!nonTerminals.Contains(symbols[i])) continue;

                    // Case 1: A → αBβ
                    if (i < symbols.Length - 1)
                    {
                        string nextSymbol = symbols[i + 1];
                        if (terminals.Contains(nextSymbol))
                        {
                            if (followSets[symbols[i]].Add(nextSymbol))
                                changed = true;
                        }
                        else
                        {
                            int countBefore = followSets[symbols[i]].Count;
                            followSets[symbols[i]].UnionWith(firstSets[nextSymbol].Except(new[] { "ε" }));
                            if (followSets[symbols[i]].Count > countBefore)
                                changed = true;

                            // If β can derive ε, add FOLLOW(A) to FOLLOW(B)
                            if (firstSets[nextSymbol].Contains("ε"))
                            {
                                countBefore = followSets[symbols[i]].Count;
                                followSets[symbols[i]].UnionWith(followSets[nonTerminal]);
```

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

FIRST(E): i, (
Press any key to continue . . .
```