**JAMIL ASKARI**

**Sp22-bcs-007**

using System;

using System.Collections.Generic;

using System.Linq;

class GrammarAnalyzer

{

private Dictionary<string, List<string>> rules = new Dictionary<string, List<string>>();

private string startSymbol = "E"; // Hardcoded as per the question

public void AddRule(string rule)

{

var parts = rule.Split(new[] { "→", "->" }, StringSplitOptions.RemoveEmptyEntries);

if (parts.Length != 2)

{

Console.WriteLine($"Invalid rule format: {rule}");

return;

}

string lhs = parts[0].Trim();

string[] rhsProductions = parts[1].Split('|').Select(p => p.Trim()).ToArray();

if (!rules.ContainsKey(lhs))

{

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

}

foreach (var production in rhsProductions)

{

rules[lhs].Add(production);

}

}

public bool HasLeftRecursion()

{

foreach (var rule in rules)

{

foreach (var production in rule.Value)

{

string[] symbols = production.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

if (symbols.Length > 0 && symbols[0] == rule.Key)

{

return true;

}

}

}

return false;

}

public Dictionary<string, HashSet<string>> ComputeFirstSets()

{

var first = new Dictionary<string, HashSet<string>>();

// Initialize first sets

foreach (var nonTerminal in rules.Keys)

{

first[nonTerminal] = new HashSet<string>();

}

bool changed;

do

{

changed = false;

foreach (var rule in rules)

{

string A = rule.Key;

foreach (var production in rule.Value)

{

string[] symbols = production.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string symbol in symbols)

{

if (symbol == "ε")

{

if (first[A].Add("ε"))

changed = true;

break;

}

else if (!rules.ContainsKey(symbol)) // Terminal

{

if (first[A].Add(symbol))

changed = true;

break;

}

else // Non-terminal

{

var prevCount = first[A].Count;

first[A].UnionWith(first[symbol].Where(s => s != "ε"));

if (first[A].Count != prevCount)

changed = true;

if (!first[symbol].Contains("ε"))

break;

}

}

// If all symbols can produce ε, add ε to FIRST(A)

bool allEpsilon = true;

foreach (string symbol in symbols)

{

if (symbol == "ε") continue;

if (!rules.ContainsKey(symbol)) // Terminal

{

allEpsilon = false;

break;

}

else if (!first[symbol].Contains("ε"))

{

allEpsilon = false;

break;

}

}

if (allEpsilon && first[A].Add("ε"))

changed = true;

}

}

} while (changed);

return first;

}

public Dictionary<string, HashSet<string>> ComputeFollowSets(Dictionary<string, HashSet<string>> first)

{

var follow = new Dictionary<string, HashSet<string>>();

// Initialize follow sets

foreach (var nonTerminal in rules.Keys)

{

follow[nonTerminal] = new HashSet<string>();

}

follow[startSymbol].Add("$"); // Add $ to start symbol

bool changed;

do

{

changed = false;

foreach (var rule in rules)

{

string A = rule.Key;

foreach (var production in rule.Value)

{

string[] symbols = production.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);

for (int i = 0; i < symbols.Length; i++)

{

if (!rules.ContainsKey(symbols[i])) continue;

string B = symbols[i];

// Case 1: B is followed by something

if (i < symbols.Length - 1)

{

string nextSymbol = symbols[i + 1];

if (!rules.ContainsKey(nextSymbol)) // Terminal

{

if (follow[B].Add(nextSymbol))

changed = true;

}

else // Non-terminal

{

var prevCount = follow[B].Count;

follow[B].UnionWith(first[nextSymbol].Where(s => s != "ε"));

if (follow[B].Count != prevCount)

changed = true;

// If nextSymbol can produce ε, add FOLLOW(A) to FOLLOW(B)

if (first[nextSymbol].Contains("ε"))

{

prevCount = follow[B].Count;

follow[B].UnionWith(follow[A]);

if (follow[B].Count != prevCount)

changed = true;

}

}

}

// Case 2: B is last in production

if (i == symbols.Length - 1)

{

var prevCount = follow[B].Count;

follow[B].UnionWith(follow[A]);

if (follow[B].Count != prevCount)

changed = true;

}

}

}

}

} while (changed);

return follow;

}

}

class Program

{

static void Main()

{

GrammarAnalyzer analyzer = new GrammarAnalyzer();

// Hardcode the grammar rules as per the question

analyzer.AddRule("E → T X");

analyzer.AddRule("X → + T X | ε");

analyzer.AddRule("T → int | ( E )");

if (analyzer.HasLeftRecursion())

{

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

return;

}

try

{

var firstSets = analyzer.ComputeFirstSets();

var followSets = analyzer.ComputeFollowSets(firstSets);

Console.WriteLine("FIRST Sets:");

foreach (var set in firstSets)

{

Console.WriteLine($"FIRST({set.Key}) = {{ {string.Join(", ", set.Value)} }}");

}

Console.WriteLine("\nFOLLOW Sets:");

foreach (var set in followSets)

{

Console.WriteLine($"FOLLOW({set.Key}) = {{ {string.Join(", ", set.Value)} }}");

}

}

catch (Exception ex)

{

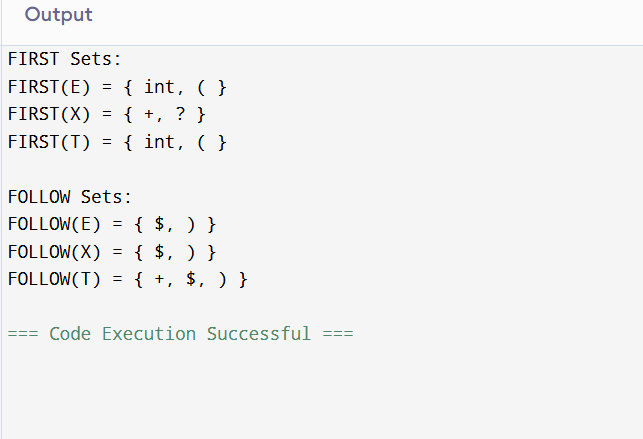
Console.WriteLine($"Error: {ex.Message}");

}

}

}

**OUTPUT**

****