using System;

using System.Collections.Generic;

using System.Linq;

class CFGProcessor

{

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

static Dictionary<string, HashSet<string>> firstSet = new();

static Dictionary<string, HashSet<string>> followSet = new();

static HashSet<string> nonTerminals = new();

static HashSet<string> terminals = new();

static string startingSymbol = "";

static void Main()

{

Console.WriteLine("Enter grammar rules (e.g., E -> T X | e). Type 'exit' to finish.\n");

// Accept grammar rules

while (true)

{

Console.Write("Rule: ");

string input = Console.ReadLine()?.Trim();

if (input?.ToLower() == "exit") break;

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

{

Console.WriteLine("Invalid format. Use 'A -> alpha | beta'");

continue;

}

string[] parts = input.Split("->", 2);

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

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

if (productions.ContainsKey(lhs))

productions[lhs].AddRange(rhs);

else

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

if (startingSymbol == "")

startingSymbol = lhs;

nonTerminals.Add(lhs);

}

IdentifyTerminals();

if (CheckLeftRecursion() || CheckAmbiguity())

{

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

return;

}

GenerateFirstSets();

GenerateFollowSets();

Console.WriteLine("\n=== FIRST SETS ===");

PrintTable(firstSet);

Console.WriteLine("\n=== FOLLOW SETS ===");

PrintTable(followSet);

}

static void IdentifyTerminals()

{

foreach (var (lhs, rules) in productions)

{

foreach (var rule in rules)

{

var symbols = rule.Split(' ', StringSplitOptions.RemoveEmptyEntries);

foreach (var sym in symbols)

{

if (!productions.ContainsKey(sym) && sym != "e")

terminals.Add(sym);

}

}

}

}

static bool CheckLeftRecursion()

{

foreach (var (lhs, rules) in productions)

{

foreach (var rule in rules)

{

var firstSymbol = rule.Split(' ', StringSplitOptions.RemoveEmptyEntries).FirstOrDefault();

if (firstSymbol == lhs)

return true;

}

}

return false;

}

static bool CheckAmbiguity()

{

foreach (var (lhs, rules) in productions)

{

var seen = new HashSet<string>();

foreach (var rule in rules)

{

if (!seen.Add(rule)) return true;

}

}

return false;

}

static void GenerateFirstSets()

{

foreach (var nt in nonTerminals)

firstSet[nt] = new HashSet<string>();

bool changed;

do

{

changed = false;

foreach (var (nt, rules) in productions)

{

foreach (var rule in rules)

{

var symbols = rule.Split(' ', StringSplitOptions.RemoveEmptyEntries);

if (symbols.Length == 0) continue;

bool canHaveEpsilon = true;

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

{

string symbol = symbols[i];

if (symbol == "e")

{

changed |= firstSet[nt].Add("e");

canHaveEpsilon = false;

}

else if (terminals.Contains(symbol))

{

changed |= firstSet[nt].Add(symbol);

canHaveEpsilon = false;

}

else if (nonTerminals.Contains(symbol))

{

foreach (var item in firstSet[symbol])

{

if (item != "e")

changed |= firstSet[nt].Add(item);

}

canHaveEpsilon = firstSet[symbol].Contains("e");

}

}

if (canHaveEpsilon)

changed |= firstSet[nt].Add("e");

}

}

} while (changed);

}

static void GenerateFollowSets()

{

foreach (var nt in nonTerminals)

followSet[nt] = new HashSet<string>();

followSet[startingSymbol].Add("$");

bool changed;

do

{

changed = false;

foreach (var (lhs, rules) in productions)

{

foreach (var rule in rules)

{

var symbols = rule.Split(' ', StringSplitOptions.RemoveEmptyEntries);

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

{

string current = symbols[i];

if (!nonTerminals.Contains(current)) continue;

bool epsilonPossible = true;

for (int j = i + 1; j < symbols.Length && epsilonPossible; j++)

{

string next = symbols[j];

epsilonPossible = false;

if (terminals.Contains(next))

{

changed |= followSet[current].Add(next);

break;

}

foreach (var item in firstSet[next])

{

if (item == "e") epsilonPossible = true;

else changed |= followSet[current].Add(item);

}

}

if (epsilonPossible || i == symbols.Length - 1)

{

foreach (var f in followSet[lhs])

changed |= followSet[current].Add(f);

}

}

}

}

} while (changed);

}

static void PrintTable(Dictionary<string, HashSet<string>> setDict)

{

Console.WriteLine("{0,-12} | {1}", "Non-Terminal", "Set");

Console.WriteLine(new string('-', 40));

foreach (var nt in nonTerminals)

{

string items = string.Join(", ", setDict[nt]);

Console.WriteLine($"{nt,-12} | {{ {items} }}");

}

}

}

