Министерство Образования Республики Беларусь

Учреждение образования

«Гомельский государственный технический университет

имени П.О.Сухого»

Кафедра: «Информационные технологии»

по курсу: «Методы трансляции»

Лабораторная работа №5

«Построение матрицы предшествования»

Выполнил: студент группы ИП-32

Суховенко Э.С.

Допуск к защите: Проверила: доцент

Дата защиты: Кравченко Ольга Алексеевна

Гомель 2022 г.

**Лабораторная работа №5**

**Построение матрицы предшествования**

Цель работы: разработка алгоритма и программы построения матрицы предшествования для заданной КС-грамматики.

Вариант 12

1. *S→B*
2. *B→W*
3. *B→B-W*
4. *W→X*
5. *X→Z*
6. *X→X^Z*
7. *Z→N*
8. *Z→(C*
9. *C→B)*
10. *N→L*
11. *N→NL*

12-37. *L→a*|*b*|…|*z*

Расшифровка КС-грамматики:

L – буква латинского алфавита (*a*|*b*|…|*z)*

N ­– несколько букв латинского алфавита без пробелов (слово)

Z – слово или выражение состоящее из слов, знаков ‘^’ и ‘-’ содержащее в себе правильную скобочную последовательность (число скобок открывающий скобок и закрывающих одинаково, а также для каждой открывающей скобки есть закрывающая и она находится справа)

X – правильное мат выражение, которое может содержать знак ‘^’ (возведение в степень), т.е a^b, где a, b – мат выражение со скобками и ‘-’ & ‘-’ или просто a

W – правильное мат выражение с вычитанием, возведением в степень и скобками

B – правильное мат выражение имеющее вид a или a – b, где a, b – правильное мат выражения.

S – законченное мат выражение состоящее из ‘-’, ‘^’, ‘(‘, ‘)’, ‘a’ – ‘z’

Примеры строк, которые можно вывести по заданной грамматике (все слова – переменные, содержащие вещественные числа):

Множество левых символов:

L(L)={a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}

L(N)={L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, N}

L(C)={B, W, X, Z, N, L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, (}

L(Z)={N, L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, (}

L(X)={Z, N, L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, (, X}

L(W)={X, Z, N, L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, (}

L(B)={W, X, Z, N, L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, (, B}

Множество правых символов:

R(L)={a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}

R(N)={L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}

R(C)={)}

R(Z)={N, L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, C, )}

R(X)={Z, N, L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, C, )}

R(W)={X, Z, N, L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, C, )}

R(B)={W, X, Z, N, L, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, C, )}

Правила простых грамматик предшествования:

Ai = Aj, если Ǝ(U→xAiAjy) – существует правило, в правой части которого символы стоят рядом.

Ai <. Aj, если Ǝ(U→xAiAky) & AjL(Ak) – существует правило, в правой части которого за символом Ai стоит символ Ak, множеству левых символов которого принадлежит Aj.

Ai .> Aj, если Ǝ(U→xAkAjy) & AiR(Ak) ˅ Ǝ(U→xAkAly) & AiR(Ak) & AjL(Al) – 1) существует правило, в правой части которого перед символом Aj стоит символ Ak, множеству правых символов которого принадлежит Ai; 2) существует правило, в правой части которого перед символом Aj стоит символ Ak, множеству правых символов которого принадлежит Ai.

**Код программы:**

Form1.cs

using lab5.LabClasses;

namespace lab5

{

public partial class Form1 : Form

{

Grammar grammar = new();

public Form1()

{

InitializeComponent();

tbGrammar.Select(0, 0);

//tbPrecendenceMatrix.Font = new Font(FontFamily.GenericMonospace, tbPrecendenceMatrix.Font.Size);

}

//Считывание правил грамматики

private void ReadRules()

{

string[] rulesStr = tbGrammar.Text.Split("\r\n");

foreach (string ruleStr in rulesStr)

{

string[] ruleParts = ruleStr.Split('→').ToList().Select(r => r.Replace(" ", null)).ToArray();

if (ruleParts.Length == 0 || ruleParts.Length > 2 || ruleParts[0].Length > 1 || ruleParts[1].Length == 0)

throw new Exception("Ошибка в правиле.");

if (ruleParts[0][0] != 'S')

{

grammar.AddRule(ruleParts[0][0], ruleParts[1]);

}

}

}

//Построение матрицы предшествования

private void bBuildPrecendenceMatrix\_Click(object sender, EventArgs e)

{

ReadRules();

tbLeftSymbolsSet.Text = grammar.GetSymbolsSetAsString(true);

tbRightSymbolsSet.Text = grammar.GetSymbolsSetAsString(false);

tbPrecendenceMatrix.Text = grammar.GetPrecendenceMatrixAsString();

if (grammar.IsOperatorPrecedenceGrammar())

{

lResult.Text = "Заданная КС-грамматика является простой грамматикой предшествования.";

}

else

{

lResult.Text = "Заданная КС-грамматика НЕ является простой грамматикой предшествования.";

}

}

private void label1\_Click(object sender, EventArgs e)

{

}

private void Form1\_Load(object sender, EventArgs e)

{

}

}

}

PrecendenceMatrixElement.cs

namespace lab5.LabClasses

{

internal class PrecendenceMatrixElement

{

public char Ai { get; set; }

public char Aj { get; set; }

//<

public bool DoesYieldPrecendenceTo { get; set; }

//=

public bool HasSamePrecendenceAs { get; set; }

//>

public bool DoesTakePrecendenceOver { get; set; }

public override string ToString()

{

char[] chars = new char[3] { ' ', ' ', ' ' };

int index = 0;

if (DoesYieldPrecendenceTo)

{

chars[index] = '<';//'⋖';

index++;

}

if (HasSamePrecendenceAs)

{

chars[index] = '=';//'≐';

index++;

}

if (DoesTakePrecendenceOver)

{

chars[index] = '>';//'⋗';

}

return string.Join(null, chars);

}

}

}

Grammar.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace lab5.LabClasses

{

internal class Grammar

{

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

private Dictionary<char, List<char>> leftSymbolsSet = null, rightSymbolsSet = null;

private List<List<PrecendenceMatrixElement>> precendenceMatrix = null;

//Обнуление указателей

private void ResetReferences()

{

leftSymbolsSet = null;

rightSymbolsSet = null;

precendenceMatrix = null;

}

//Добавление правила в грамматику

public void AddRule(char symbol, string rule)

{

if (rules.ContainsKey(symbol))

{

rules[symbol].Add(rule);

}

else

{

List<string> newRulesList = new();

newRulesList.Add(rule);

rules.Add(symbol, newRulesList);

}

ResetReferences();

}

//Очистка правил

public void Clear()

{

rules.Clear();

ResetReferences();

}

//Получение множества левых или правых символов для нетерминального символа

private void GetSymbolsSetIntrinsic(char symbol, List<char> symbols, bool getLeftSymbols)

{

if (rules.ContainsKey(symbol))

{

List<string> symbolRules = rules[symbol];

foreach (string symbolRule in symbolRules)

{

char nextSymbol;

if (getLeftSymbols)

{

nextSymbol = symbolRule[0];

}

else

{

nextSymbol = symbolRule[^1];

}

if (!symbols.Contains(nextSymbol))

{

symbols.Add(nextSymbol);

if (nextSymbol != symbol)

{

GetSymbolsSetIntrinsic(nextSymbol, symbols, getLeftSymbols);

}

}

}

}

}

//Получение всех множеств левых или правых символов для нетерминальных символов

private void GetSymbolsSet(bool getLeftSymbols)

{

Dictionary<char, List<char>> symbolsSet;

if (getLeftSymbols)

{

leftSymbolsSet = new();

symbolsSet = leftSymbolsSet;

}

else

{

rightSymbolsSet = new();

symbolsSet = rightSymbolsSet;

}

foreach (char symbol in rules.Keys)

{

if (symbolsSet.ContainsKey(symbol))

{

GetSymbolsSetIntrinsic(symbol, symbolsSet[symbol], getLeftSymbols);

}

else

{

List<char> newSymbols = new();

GetSymbolsSetIntrinsic(symbol, newSymbols, getLeftSymbols);

symbolsSet.Add(symbol, newSymbols);

}

}

}

//Инициализация матрицы предшествования

private void InitPrecendenceMatrix()

{

precendenceMatrix = new();

List<char> symbols = new();

foreach (char ruleSymbol in rules.Keys)

{

symbols.Add(ruleSymbol);

}

foreach (var rulesList in rules.Values)

{

foreach (var rule in rulesList)

{

foreach (char symbol in rule)

{

if (!symbols.Contains(symbol))

{

symbols.Add(symbol);

}

}

}

}

foreach (char rowSymbol in symbols)

{

List<PrecendenceMatrixElement> newRow = new();

foreach (char columnSymbol in symbols)

{

newRow.Add(new () { Ai = rowSymbol, Aj = columnSymbol });

}

precendenceMatrix.Add(newRow);

}

}

//Проверка отношения типа "=" для символов

private bool HasSamePrecendenceAs(char ai, char aj)

{

bool isItTrue = false;

string pair = $"{ai}{aj}";

foreach (var rulesList in rules.Values)

{

foreach (var rule in rulesList)

{

if (rule.Contains(pair))

{

isItTrue = true;

}

}

}

return isItTrue;

}

//Проверка отношения типа "<" для символов

private bool DoesYieldPrecendenceTo(char ai, char aj)

{

var leftSymbolsContainingAj = leftSymbolsSet.Where(symbolLeftSymbolsSet => symbolLeftSymbolsSet.Value.Contains(aj));

bool isItTrue = false;

foreach (var symbolLeftSymbolsSet in leftSymbolsContainingAj)

{

if (HasSamePrecendenceAs(ai, symbolLeftSymbolsSet.Key))

{

isItTrue = true;

}

}

return isItTrue;

}

//Проверка отношения типа ">" для символов

private bool DoesTakePrecendenceOver(char ai, char aj)

{

var rightSymbolsContainingAi = rightSymbolsSet.Where(symbolRightSymbolsSet => symbolRightSymbolsSet.Value.Contains(ai));

var leftSymbolsContainingAj = leftSymbolsSet.Where(symbolLeftSymbolsSet => symbolLeftSymbolsSet.Value.Contains(aj));

bool isItTrue = false;

//первый случай

foreach (var symbolRightSymbolsSet in rightSymbolsContainingAi)

{

if (HasSamePrecendenceAs(symbolRightSymbolsSet.Key, aj))

{

isItTrue = true;

}

}

//второй случай

if (!isItTrue)

{

foreach (var symbolRightSymbolsSet in rightSymbolsContainingAi)

{

foreach (var symbolLeftSymbolsSet in leftSymbolsContainingAj)

{

if (HasSamePrecendenceAs(symbolRightSymbolsSet.Key, symbolLeftSymbolsSet.Key))

{

isItTrue = true;

}

}

}

}

return isItTrue;

}

//Получение матрицы предшествования

private void GetPrecendenceMatrix()

{

GetSymbolsSet(true);

GetSymbolsSet(false);

InitPrecendenceMatrix();

foreach (var rowList in precendenceMatrix)

{

char rowSymbol = rowList[0].Ai;

foreach (var rowListElement in rowList)

{

char columnSymbol = rowListElement.Aj;

PrecendenceMatrixElement element = rowListElement;

element.HasSamePrecendenceAs = HasSamePrecendenceAs(rowSymbol, columnSymbol);

element.DoesYieldPrecendenceTo = DoesYieldPrecendenceTo(rowSymbol, columnSymbol);

element.DoesTakePrecendenceOver = DoesTakePrecendenceOver(rowSymbol, columnSymbol);

}

}

}

//Получение множеств левых и правых символов в текстовом виде

public string GetSymbolsSetAsString(bool getLeftSymbolsSet)

{

GetSymbolsSet(getLeftSymbolsSet);

char setSymbol = 'L';

Dictionary<char, List<char>> symbolSet = leftSymbolsSet;

if (!getLeftSymbolsSet)

{

setSymbol = 'R';

symbolSet = rightSymbolsSet;

}

StringBuilder sb = new();

var symbolSetList = symbolSet.ToList();

symbolSetList.Reverse();

foreach (var symbolSymbolsSet in symbolSetList)

{

sb.Append($"{setSymbol}({symbolSymbolsSet.Key})={{{string.Join(", ", symbolSymbolsSet.Value)}}}\r\n");

}

return sb.ToString();

}

//Получение матрицы предшествования в виде таблицы в текстовом виде

public string GetPrecendenceMatrixAsString()

{

GetPrecendenceMatrix();

StringBuilder sb = new();

int rowLength = 6 \* (precendenceMatrix.Count + 1) + 1;

//string separator = new string('-', rowLength) + "\r\n";

string separator = "";

for (int i = 0; i < rowLength; i++)

separator += ".";

separator += "\r\n";

sb.Append(separator);

sb.Append("| |");

foreach (var element in precendenceMatrix[0])

{

sb.Append($" {element.Aj} |");

}

sb.Append("\r\n");

sb.Append(separator);

foreach (var rowList in precendenceMatrix)

{

sb.Append($"| {rowList[0].Ai} |");

foreach (var columnElement in rowList)

{

sb.Append($" {columnElement} |");

}

sb.Append("\r\n");

sb.Append(separator);

}

return sb.ToString();

}

//Проверка заданной грамматики на то, является ли она ПГП

public bool IsOperatorPrecedenceGrammar()

{

bool isItTrue = true;

//первое условие

for (int i = 0; i < precendenceMatrix.Count && isItTrue; i++)

{

for (int j = 0; j < precendenceMatrix[i].Count && isItTrue; j++)

{

int count = 0;

PrecendenceMatrixElement element = precendenceMatrix[i][j];

if (element.HasSamePrecendenceAs)

{

count++;

}

if (element.DoesYieldPrecendenceTo)

{

count++;

}

if (element.DoesTakePrecendenceOver)

{

count++;

}

if (count > 1)

{

isItTrue = false;

}

}

}

if (isItTrue)

{

//второе условие

foreach (var outerRulesList in rules.Values)

{

foreach (var outerRulesListElement in outerRulesList)

{

foreach (var innerRulesList in rules.Values)

{

if (outerRulesList != innerRulesList)

{

foreach (var innerRulesListElement in innerRulesList)

{

if (outerRulesListElement == innerRulesListElement)

{

isItTrue = false;

}

}

}

}

}

}

}

return isItTrue;

}

}

}

Результат

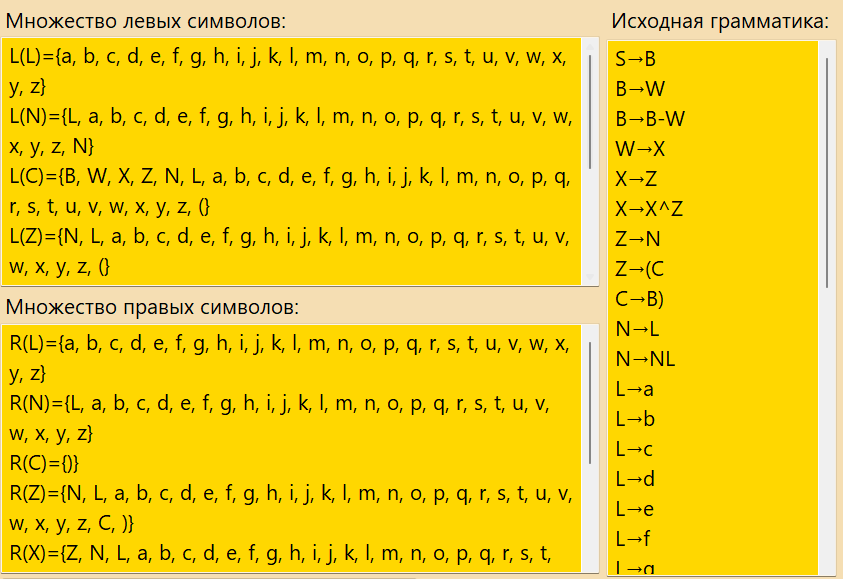
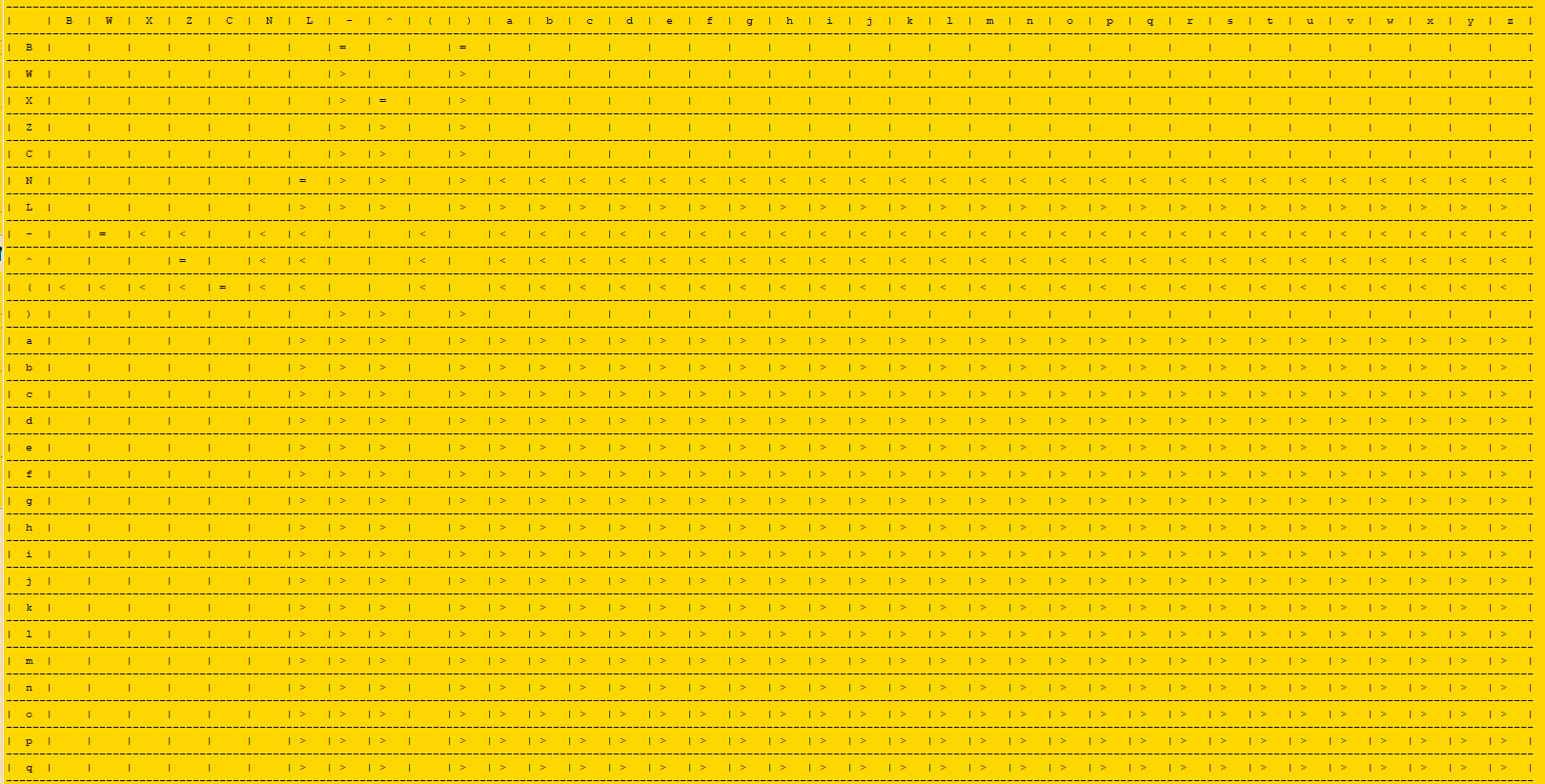


Рисунок 1



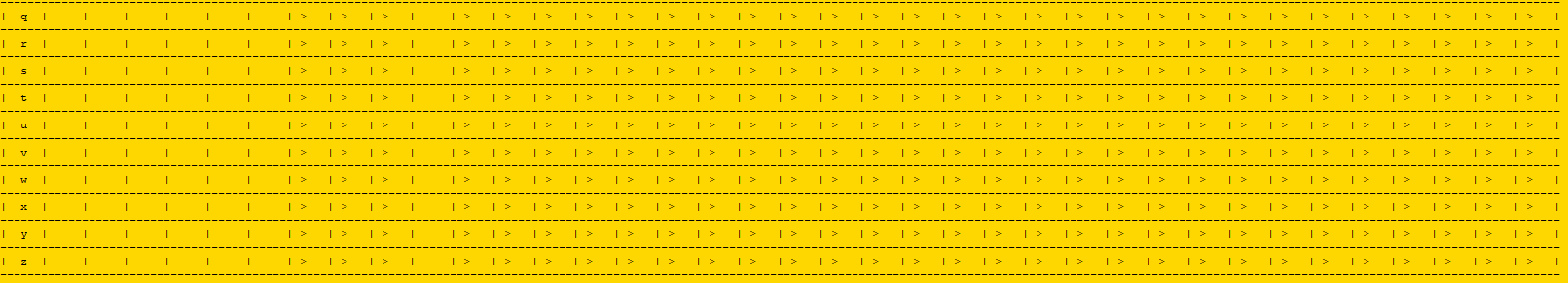


Рисунок 2



Рисунко 3

Вывод: в ходе выполнения лабораторной работы был разработан алгоритм и программа построения матрицы предшествования для заданной КС-грамматики.