ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 3

Проектирование лексического анализатора

Выполнил: Тимощенко Т.С.

Вариант - 13

1. ЗАДАНИЕ ПО ЛАБОРАТОРНОЙ РАБОТЕ

Разработать программу на WPF, которая выполняет следующие задачи:

* Считывает входной текст исходной программы из файла.
* Выполняет лексический анализ входного текста, формируя таблицу лексем, где вместо символа a подставляются реальные лексемы (идентификаторы и символьные константы).
* Производит синтаксический разбор по заданной грамматике, строя цепочку вывода и дерево синтаксического разбора.
* Выводит результаты работы: исходный текст, таблицу лексем и дерево разбора.

Вариант грамматики входного языка:

S -> a := F;

F -> F + T | T

T -> T \* E | T / E | E

E -> (F) | -(F) | a

Допустимые лексемы:

* Идентификаторы
* Символьные константы (в одинарных кавычках)
* Дополнительно: римские числа (последовательности букв X, V, I) и шестнадцатеричные числа (последовательность цифр и символов a–f, начинающаяся с цифры).

1. КРАТКОЕ ИЗЛОЖЕНИЕ ЦЕЛИ РАБОТЫ

Цель работы – изучение основ теории грамматик и синтаксического анализа, а также практическое освоение алгоритмов построения дерева синтаксического разбора. В рамках лабораторной работы реализуется программа, выполняющая:

* Лексический анализ входного текста (выделение лексем).
* Синтаксический анализ с использованием преобразованной грамматики (без левой рекурсии) методом рекурсивного спуска.
* Построение и визуальное отображение дерева вывода, которое используется для последующих этапов семантического анализа и генерации кода.

1. ГРАММАТИКА В ФОРМЕ БЭКУСА-НАУРА

Запись заданной грамматики входного языка в нотации Бэкуса-Наура:

<S> ::= <a> ":=" <F> ";"  
<F> ::= <F> "+" <T> | <T>  
<T> ::= <T> "\*" <E> | <T> "/" <E> | <E>  
<E> ::= "(" <F> ")" | "-" "(" <F> ")" | <a>

Здесь символ <a> соответствует терминальным лексемам (идентификаторам или символьным константам).

1. МНОЖЕСТВА КРАЙНИХ ЛЕВЫХ И КРАЙНИХ ПРАВЫХ СИМВОЛОВ

**Крайние левые символы**

S начинается с a (идентификатор или символьная константа), так как в правиле S → a := F; первым идёт терминал a.

F может начинаться с F + T или напрямую с T, но в любом случае вывод сводится к T → E, а E может начинаться с (, - или a. Поэтому крайние левые символы для F — это те же (, -, a.

T аналогично определяется правилами T → T \* E | T / E | E и тоже сводится к E, значит его крайние левые символы такие же: (, -, a.

E сразу задаётся правилами E → (F) | -(F) | a, откуда видно, что вывод начинается либо с (, либо с -, либо с a.

Таким образом, крайние левые символы:

* S: a
* F: (, -, a
* T: (, -, a
* E: (, -, a

**Крайние правые символы**

S заканчивается на F, после которого идёт ; (по правилу S → a := F;). Значит, в самом конце S всегда будет точка с запятой ;.

F в конце переходит к T (F → F + T | T), а T → E, следовательно, итоговый вывод F оканчивается тем же, чем может заканчиваться E.

T тоже завершается E (T → T \* E | T / E | E), значит последний терминал у T тот же, что и у E.

E может быть (F), -(F) или a. В первых двух случаях крайним символом оказывается ), а в последнем случае — a.

Таким образом, крайние правые символы:

* S: ;
* F: ), a
* T: ), a
* E: ), a

1. МАТРИЦА ПРЕДШЕСТВОВАНИЯ

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **a** | **:=** | **+** | **\*** | **/** | **(** | **)** | **;** | **конец** |
| **a** | > | > | > | > | > |  | > | > | > |
| **:=** | < |  | < | < | < | < |  | > | > |
| **+** | < | > | < | < | < | < | > | > | > |
| \* | < | > | > | < | < | < | > | > | > |
| **/** | < | > | > | < | < | < | > | > | > |
| **(** | < | < | < | < | < | = | > |  |  |
| **)** |  | > | > | > | > |  | > | > | > |
| **;** | < | < | < | < | < | < |  |  |  |
| **начало** | < | < | < | < | < | < |  |  |  |

1. ПРИМЕР ВЫПОЛНЕНИЯ РАЗБОРА

Входной файл:

x := (y + 'a') \* -(z);

**Пошаговый разбор:**

1. Лексический анализ.  
   Программа преобразует текст в следующую таблицу лексем:

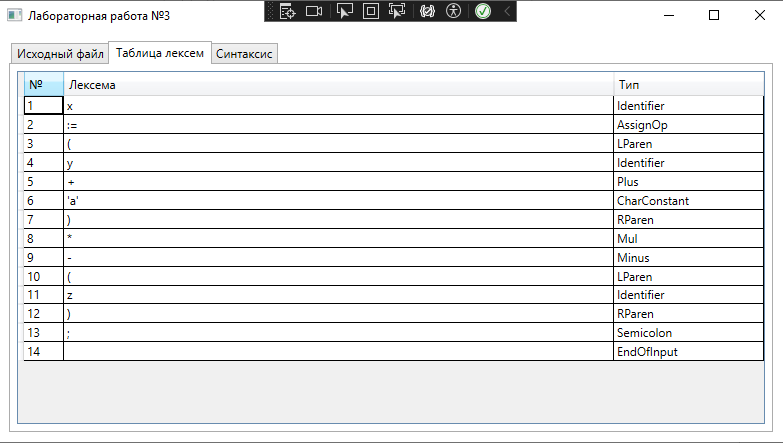


Рисунок 1 – Таблица лексем.

1. Синтаксический разбор.  
   Программа на основе правил грамматики строит дерево разбора. Ожидаемая структура дерева:

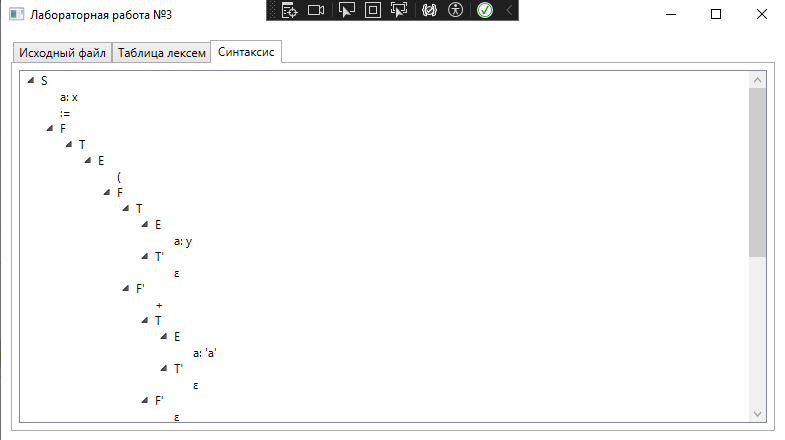


Рисунок 2 – Синтаксис.

1. Вывод программы.  
   После успешного разбора в приложении отображаются:

* Исходный текст из файла.
* Таблица лексем.
* Дерево синтаксического разбора (развёрнутое дерево).

1. ТЕКСТ ПРОГРАММЫ

MainWindow.xaml

<Window x:Class="lab3.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="Лабораторная работа №3" Height="450" Width="800">

<Grid>

<TabControl x:Name="tabControl" Margin="10">

<!-- Вкладка 1: Исходный файл -->

<TabItem Header="Исходный файл">

<Grid>

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition/>

</Grid.RowDefinitions>

<StackPanel Orientation="Horizontal" Margin="5">

<Button x:Name="btnOpenFile" Content="Выбрать файл"

Click="btnOpenFile\_Click" Margin="0,0,10,0"/>

<Button x:Name="btnParse" Content="Запустить разбор"

Click="btnParse\_Click"/>

</StackPanel>

<TextBox x:Name="txtSource"

Margin="5"

Grid.Row="1"

AcceptsReturn="True"

VerticalScrollBarVisibility="Auto"

HorizontalScrollBarVisibility="Auto"/>

</Grid>

</TabItem>

<!-- Вкладка 2: Таблица лексем -->

<TabItem Header="Таблица лексем">

<DataGrid x:Name="dgTokens" AutoGenerateColumns="False" Margin="5">

<DataGrid.Columns>

<DataGridTextColumn Header="№" Binding="{Binding Index}" Width="40"/>

<DataGridTextColumn Header="Лексема" Binding="{Binding Lexeme}" Width="\*"/>

<DataGridTextColumn Header="Тип" Binding="{Binding TokenType}" Width="150"/>

</DataGrid.Columns>

</DataGrid>

</TabItem>

<!-- Вкладка 3: Синтаксис (дерево разбора) -->

<TabItem Header="Синтаксис">

<TreeView x:Name="treeSyntax" Margin="5"/>

</TabItem>

</TabControl>

</Grid>

</Window>

MainWindow.xaml.cs

using lab3.Models;

using Microsoft.Win32;

using System;

using System.Collections.Generic;

using System.Windows;

using System.Windows.Controls;

namespace lab3

{

public partial class MainWindow : Window

{

private List<Token> \_tokens = new List<Token>();

private Parser \_parser;

public MainWindow()

{

InitializeComponent();

}

private void btnOpenFile\_Click(object sender, RoutedEventArgs e)

{

var ofd = new OpenFileDialog();

ofd.Filter = "Текстовые файлы|\*.txt;\*.cs;\*.cpp;\*.pas|Все файлы|\*.\*";

if (ofd.ShowDialog() == true)

{

txtSource.Text = System.IO.File.ReadAllText(ofd.FileName);

}

}

private void btnParse\_Click(object sender, RoutedEventArgs e)

{

var scanner = new Scanner();

\_tokens = scanner.Scan(txtSource.Text);

for (int i = 0; i < \_tokens.Count; i++)

\_tokens[i].Index = i + 1;

dgTokens.ItemsSource = \_tokens;

dgTokens.Items.Refresh();

\_parser = new Parser(\_tokens);

try

{

var root = \_parser.ParseS();

treeSyntax.Items.Clear();

var treeItem = CreateTreeItem(root);

treeSyntax.Items.Add(treeItem);

treeItem.ExpandSubtree();

}

catch (Exception ex)

{

MessageBox.Show("Синтаксическая ошибка: " + ex.Message);

}

}

private TreeViewItem CreateTreeItem(Node node)

{

var item = new TreeViewItem { Header = node.Name };

foreach (var child in node.Children)

{

item.Items.Add(CreateTreeItem(child));

}

return item;

}

}

}

Token.cs

using System.Collections.Generic;

namespace lab3.Models

{

public enum TokenType

{

Identifier,

CharConstant,

Plus, Minus, Mul, Div,

AssignOp,

Semicolon,

LParen, RParen,

Unknown,

EndOfInput

}

public class Token

{

public int Index { get; set; }

public string Lexeme { get; set; }

public TokenType TokenType { get; set; }

public Token(string lexeme, TokenType type)

{

Lexeme = lexeme;

TokenType = type;

}

}

}

Scanner.cs

using System.Collections.Generic;

namespace lab3.Models

{

public class Scanner

{

public List<Token> Scan(string text)

{

var tokens = new List<Token>();

int i = 0;

while (i < text.Length)

{

char c = text[i];

if (char.IsWhiteSpace(c))

{

i++;

continue;

}

if (c == ':' && i + 1 < text.Length && text[i + 1] == '=')

{

tokens.Add(new Token(":=", TokenType.AssignOp));

i += 2;

continue;

}

switch (c)

{

case ';':

tokens.Add(new Token(";", TokenType.Semicolon));

i++;

continue;

case '+':

tokens.Add(new Token("+", TokenType.Plus));

i++;

continue;

case '-':

tokens.Add(new Token("-", TokenType.Minus));

i++;

continue;

case '\*':

tokens.Add(new Token("\*", TokenType.Mul));

i++;

continue;

case '/':

tokens.Add(new Token("/", TokenType.Div));

i++;

continue;

case '(':

tokens.Add(new Token("(", TokenType.LParen));

i++;

continue;

case ')':

tokens.Add(new Token(")", TokenType.RParen));

i++;

continue;

}

if (c == '\'')

{

int start = i;

i++;

while (i < text.Length && text[i] != '\'')

i++;

if (i < text.Length) i++;

string lexeme = text.Substring(start, i - start);

tokens.Add(new Token(lexeme, TokenType.CharConstant));

continue;

}

if (char.IsLetter(c))

{

int start = i;

i++;

while (i < text.Length && (char.IsLetterOrDigit(text[i]) || text[i] == '\_'))

{

i++;

}

string lexeme = text.Substring(start, i - start);

tokens.Add(new Token(lexeme, TokenType.Identifier));

continue;

}

tokens.Add(new Token(c.ToString(), TokenType.Unknown));

i++;

}

tokens.Add(new Token("", TokenType.EndOfInput));

return tokens;

}

}

}

Parser.cs

using System;

using System.Collections.Generic;

namespace lab3.Models

{

public class Parser

{

private List<Token> \_tokens;

private int \_pos;

public Parser(List<Token> tokens)

{

\_tokens = tokens;

\_pos = 0;

}

private Token CurrentToken => \_tokens[\_pos];

private void NextToken() => \_pos++;

private void Match(TokenType type)

{

if (CurrentToken.TokenType == type)

{

NextToken();

}

else

{

throw new Exception($"Ожидалась лексема типа {type}, получено {CurrentToken.TokenType}");

}

}

private bool IsCurrentToken(TokenType type)

{

return CurrentToken.TokenType == type;

}

public Node ParseS()

{

var sNode = new Node("S");

if (!IsCurrentToken(TokenType.Identifier) && !IsCurrentToken(TokenType.CharConstant))

throw new Exception("Ожидался идентификатор или символьная константа в начале выражения (S)");

var aNode = new Node($"a: {CurrentToken.Lexeme}");

sNode.AddChild(aNode);

NextToken();

if (!IsCurrentToken(TokenType.AssignOp))

throw new Exception("Ожидался оператор ':='");

var assignNode = new Node($":=");

sNode.AddChild(assignNode);

NextToken();

var fNode = ParseF();

sNode.AddChild(fNode);

Match(TokenType.Semicolon);

return sNode;

}

private Node ParseF()

{

var fNode = new Node("F");

var tNode = ParseT();

fNode.AddChild(tNode);

var fPrime = ParseFPrime();

fNode.AddChild(fPrime);

return fNode;

}

private Node ParseFPrime()

{

var fPrime = new Node("F'");

if (IsCurrentToken(TokenType.Plus))

{

var plusNode = new Node("+");

fPrime.AddChild(plusNode);

NextToken();

var tNode = ParseT();

fPrime.AddChild(tNode);

var fPrime2 = ParseFPrime();

fPrime.AddChild(fPrime2);

}

else

{

fPrime.AddChild(new Node("ε"));

}

return fPrime;

}

private Node ParseT()

{

var tNode = new Node("T");

var eNode = ParseE();

tNode.AddChild(eNode);

var tPrime = ParseTPrime();

tNode.AddChild(tPrime);

return tNode;

}

private Node ParseTPrime()

{

var tPrime = new Node("T'");

if (IsCurrentToken(TokenType.Mul) || IsCurrentToken(TokenType.Div))

{

var op = CurrentToken.TokenType == TokenType.Mul ? "\*" : "/";

var opNode = new Node(op);

tPrime.AddChild(opNode);

NextToken();

var eNode = ParseE();

tPrime.AddChild(eNode);

var tPrime2 = ParseTPrime();

tPrime.AddChild(tPrime2);

}

else

{

tPrime.AddChild(new Node("ε"));

}

return tPrime;

}

private Node ParseE()

{

var eNode = new Node("E");

if (IsCurrentToken(TokenType.LParen))

{

var lparen = new Node("(");

eNode.AddChild(lparen);

NextToken();

var fNode = ParseF();

eNode.AddChild(fNode);

if (!IsCurrentToken(TokenType.RParen))

throw new Exception("Ожидалась закрывающая скобка ')'");

var rparen = new Node(")");

eNode.AddChild(rparen);

NextToken();

}

else if (IsCurrentToken(TokenType.Minus))

{

var minusNode = new Node("-");

eNode.AddChild(minusNode);

NextToken();

if (!IsCurrentToken(TokenType.LParen))

throw new Exception("После '-' ожидалась '('");

var lparen = new Node("(");

eNode.AddChild(lparen);

NextToken();

var fNode = ParseF();

eNode.AddChild(fNode);

if (!IsCurrentToken(TokenType.RParen))

throw new Exception("Ожидалась ')' после (F");

var rparen = new Node(")");

eNode.AddChild(rparen);

NextToken();

}

else

{

if (!IsCurrentToken(TokenType.Identifier) && !IsCurrentToken(TokenType.CharConstant))

throw new Exception("Ожидался идентификатор/символ (E)");

var aNode = new Node($"a: {CurrentToken.Lexeme}");

eNode.AddChild(aNode);

NextToken();

}

return eNode;

}

}

}

Node.cs

using System.Collections.Generic;

namespace lab3.Models

{

public class Node

{

public string Name { get; set; }

public List<Node> Children { get; set; } = new List<Node>();

public Node(string name)

{

Name = name;

}

public void AddChild(Node child)

{

Children.Add(child);

}

}

}