Министерство образования Российской Федерации

Федеральное агентство по образованию

КАЗАНСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

им. А.Н. Туполева - КАИ

Кафедра АСОИУ

Отчет по лабораторным работам №1-8

по дисциплинам

«ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ» и

«МЕТОДЫ ТРАНСЛЯЦИИ»

Выполнил:

Студент группы 4309

Мазитов Р.Р.

Проверила: Бикмуллина И.И.

Казань 2023

**Лабораторная работа № 1. Разработка транслитератора**

**Текст задания:**

1. Спроектировать и отладить экранную форму для ввода исходных данных, вывода сообщений программы и управления программой.
2. Разработать и отладить транслитератор **void GetSymbol()**, пример имеется в модуле **uLexicalAnalizer** из папки «Программы».
3. Для отладки транслитератора временно включить в обработчик нажатия кнопки цикл чтения с помощью функции **GetSymbol()** символов исходного текста и вывода результатов анализа в поле диагностических сообщений.

**Код программы:**

**Form1.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace mazitov

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

textBox1.AppendText("01ab" + "\r\n");

textBox1.AppendText("1 a");

int n = textBox1.Lines.Length;

}

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

{

}

private void button1\_Click(object sender, EventArgs e)

{

uLex Lex = new uLex();

Lex.strPSource = textBox1.Lines;

Lex.strPMessage = textBox2.Lines;

int x = textBox1.TextLength;

int y = textBox1.Lines.Length;

textBox2.Text = "";

try

{

while (Lex.enumPState != TState.Finish)

{

Lex.GetSymbol(); // Выводятся литеры и классификация

Lex.NextToken();

String s = "";

String s1 = "";

switch (Lex.enumFSelectionCharType)

{

case TCharType.Letter: { s1 = "Letter"; break; }

case TCharType.Digit: { s1 = "Digit"; break; }

case TCharType.Space: { s1 = "Space"; break; }

case TCharType.clBracket: { s1 = "CloseBracket"; break; }

case TCharType.opBracket: { s1 = "OpenBracket"; break; }

case TCharType.expMark: { s1 = "ExclamationMark"; break; }

case TCharType.comma: { s1 = "Comma"; break; }

case TCharType.semicolon: { s1 = "Semicolon"; break; }

case TCharType.EndRow: { s = "KC"; s1 = "EndRow"; break; }

case TCharType.EndText: { s = "KT"; s1 = "EndText"; break; }

}

String m = "(" + s + "," + s1 + ")"; //литера и ее тип

textBox2.Text += m; //добавляется в строку сообщение

}

}

catch (Exception exc)

{

textBox2.Text += exc.Message;

textBox1.Select();

textBox1.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Lex.intPSourceRowSelection; i++) n += textBox1.Lines[i].Length + 2;

n += Lex.intPSourceColSelection;

textBox1.SelectionLength = n;

}

}

private void Form1\_Load\_1(object sender, EventArgs e)

{

}

}

}

**uLex.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace nsLex

{

public enum TState { Start, Continue, Finish }; //тип состояния

public enum TCharType { Letter, Digit, EndRow, EndText, Space, ReservedSymbol, opBracket, clBracket, expMark, comma, semicolon }; // тип символа

public enum TToken { lxmIdentifier, lxmNumber, lxmUnknown, lxmEmpty, lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmComma };

class uLex

{

private String[] strFSource; // указатель на массив строк

private String[] strFMessage; // указатель на массив строк

public TCharType enumFSelectionCharType;

public char chrFSelection;

private TState enumFState;

private int intFSourceRowSelection;

private int intFSourceColSelection;

private String strFLexicalUnit;

private TToken enumFToken;

public String[] strPSource { set { strFSource = value; } get { return strFSource; } }

public String[] strPMessage { set { strFMessage = value; } get { return strFMessage; } }

public TState enumPState { set { enumFState = value; } get { return enumFState; } }

public String strPLexicalUnit { set { strFLexicalUnit = value; } get { return strFLexicalUnit; } }

public TToken enumPToken { set { enumFToken = value; } get { return enumFToken; } }

public int intPSourceRowSelection { get { return intFSourceRowSelection; } set { intFSourceRowSelection = value; } }

public int intPSourceColSelection { get { return intFSourceColSelection; } set { intFSourceColSelection = value; } }

public uLex()

{

}

public void GetSymbol() //метод класса лексический анализатор

{

// продвигаем номер колонки

if (intFSourceColSelection > strFSource[intFSourceRowSelection].Length - 1)

{

intFSourceRowSelection++;

if (intFSourceRowSelection <= strFSource.Length - 1)

{

intFSourceColSelection = -1;

chrFSelection = '\0';

enumFSelectionCharType = TCharType.EndRow;

enumFState = TState.Continue;

}

else

{

chrFSelection = '\0';

enumFSelectionCharType = TCharType.EndText;

enumFState = TState.Finish;

}

}

else

{

chrFSelection = strFSource[intFSourceRowSelection][intFSourceColSelection]; //классификация прочитанной литеры

if (chrFSelection == ' ') enumFSelectionCharType = TCharType.Space;

else if (chrFSelection >= 'a' && chrFSelection <= 'd') enumFSelectionCharType = TCharType.Letter;

else if (chrFSelection == '0' || chrFSelection == '1') enumFSelectionCharType = TCharType.Digit;

else if (chrFSelection == '/') enumFSelectionCharType = TCharType.ReservedSymbol;

else if (chrFSelection == '\*') enumFSelectionCharType = TCharType.ReservedSymbol;

else if (chrFSelection == '(') enumFSelectionCharType = TCharType.opBracket;

else if (chrFSelection == ')') enumFSelectionCharType = TCharType.clBracket;

else if (chrFSelection == '!') enumFSelectionCharType = TCharType.expMark;

else if (chrFSelection == ',') enumFSelectionCharType = TCharType.comma;

else if (chrFSelection == ';') enumFSelectionCharType = TCharType.semicolon;

//else if (chrFSelection == '(' || chrFSelection == ')' || chrFSelection == ':' || chrFSelection == '-' || chrFSelection == ',' || chrFSelection == '.') enumFSelectionCharType = TCharType.ReservedSymbol;

else throw new System.Exception("Cимвол вне алфавита");

enumFState = TState.Continue;

}

intFSourceColSelection++;

}

private void TakeSymbol()

{

char[] c = { chrFSelection };

String s = new string(c);

strFLexicalUnit += s;

GetSymbol();

}

public void NextToken()

{

strFLexicalUnit = "";

if (enumFState == TState.Start)

{

intFSourceRowSelection = 0;

intFSourceColSelection = -1;

GetSymbol();

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

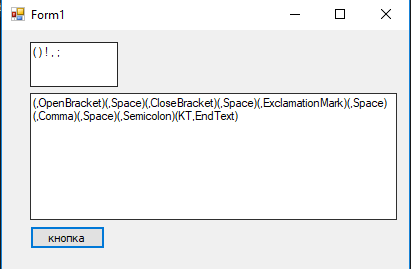
}

}

}

}

**Результат:**



# **Лабораторная работа № 2. Разработка лексического анализатора**

Лексический анализатор предназначен для чтения слов в исходном тексте и классификации прочитанных слов.

Основные функции лексического анализатора:

* Чтение с помощью транслитератора очередного слова в исходном тексте и его классификация;
* Пропуск пробелов и комментариев;
* Выдача диагностических сообщений об обнаруженных лексических ошибках.

Токен – некоторый класс слов (выделяется из потребностей синтаксического анализа в основном по принципу построения и назначения слов).

Лексический анализатор – программа, принимающая на вход текст (последовательность символов из алфавита) и разбивающая его на подстроки (лексемы) в соответствии с некоторым набором регулярных выражений. Последовательность лексем такова, что каждая лексема принадлежит хотя бы одному из языков, задаваемых регулярными выражениями.

1. Детерминированный граф:

В детерминированном графе для каждого состояния и входного события существует единственное определенное следующее состояние.

Переходы в детерминированном графе предсказуемы и однозначны.

Детерминированные графы широко используются в теории автоматов, в моделях вычислений, и в других областях, где важна однозначность переходов между состояниями.

1. Недетерминированный граф:

В недетерминированном графе для некоторых состояний и входных событий может существовать более одного возможного следующего состояния.

Переходы в недетерминированном графе могут быть неоднозначными, и система может выбирать один из нескольких возможных переходов в зависимости от условий выполнения.

Недетерминированные графы часто используются в моделях параллельных вычислений, теории конечных автоматов, а также в теории формальных языков.

Примером детерминированного графа может служить конечный автомат, в котором каждому состоянию соответствует определенный переход при входном символе. Напротив, недетерминированный конечный автомат может иметь несколько возможных переходов для одного и того же входа.

В детерминированном алгоритме для данного конкретного входного сигнала компьютер всегда будет выдавать один и тот же выходной сигнал, проходящий через одни и те же состояния, но в случае недетерминированного алгоритма для одного и того же входного сигнала компилятор может выдавать разные выходные данные в разных запусках. На самом деле, недетерминированные алгоритмы не могут решить проблему за полиномиальное время и не могут определить, каков следующий шаг. Недетерминированные алгоритмы могут демонстрировать разное поведение для одних и тех же входных данных при разном выполнении, и в этом есть определенная степень случайности.

Терминальные и нетерминальные понятия широко используются в контексте формальных языков и грамматик. И существуют терминальные и нетерминальные символы:

1. Терминальные символы:

* Это символы, которые присутствуют в конечном выводе строки языка.
* В контексте формальных грамматик они являются "конечными" элементами, которые не подлежат дальнейшему раскрытию или замене.
* В языках программирования терминальными символами могут быть, например, ключевые слова, знаки пунктуации, числа, строки и другие элементы конечного кода.

1. Нетерминальные символы:

* Это символы, которые могут быть заменены другими символами, включая последовательности терминальных и/или нетерминальных символов.
* В контексте формальных грамматик они представляют "неконечные" элементы, которые могут быть раскрыты в более длинные последовательности символов.
* В языках программирования нетерминальными символами могут быть, например, выражения, операторы, функции, идентификаторы и другие элементы, которые могут быть дополнительно развернуты или определены.

Элементарные операции над матрицами — это операции, выполняемые над строками и столбцами матрицы, которые не изменяют значение матрицы. Матрица — это способ представления чисел в виде массива, т.е. числа расположены в виде строк и столбцов. В матрице строки и столбцы содержат все значения в ячейке. Мы представляем матрицу в виде [A]m × n, где A - матрица, m - количество строк в матрице, n - количество столбцов матрицы.

**Текст задания:**

1. Спроектировать и отладить экранную форму для ввода исходных данных, вывода сообщений программы и управления программой.
2. Включить из лабораторной работы № 1 транслитератор **void GetSymbol().**
3. Составить регулярную грамматику для каждого вида слов.
4. Построить конечные автоматы для каждого вида слов, как правило, они будут недетерминированными.
5. Построить детерминированные конечные автоматы для каждого вида слов.
6. Составить объединенный конечный автомат.
7. Написать и отладить модуль лексического анализатора по алгоритму объединенного конечного автомата. Для чтения исходного текста использовать транслитератор. Предусмотреть обработчик лексических ошибок исходного текста, используется конструкция **try … catch**.
8. Для отладки лексического анализатора временно включить в обработчик нажатия кнопки цикл чтения слов исходного текста и вывода результатов лексического анализа.

|  |  |  |
| --- | --- | --- |
| (010)\*000(001)\* | (a|b|c|d)+ | Первые два символа всегда ca |

**Первое слово:**

(010)\*000(001)\*

A → 0B | 0C

B → 1D

C → 0Е

D → 0А

E → 0 | 0F

F → 0G

G → 0H

H → 1 | 1F

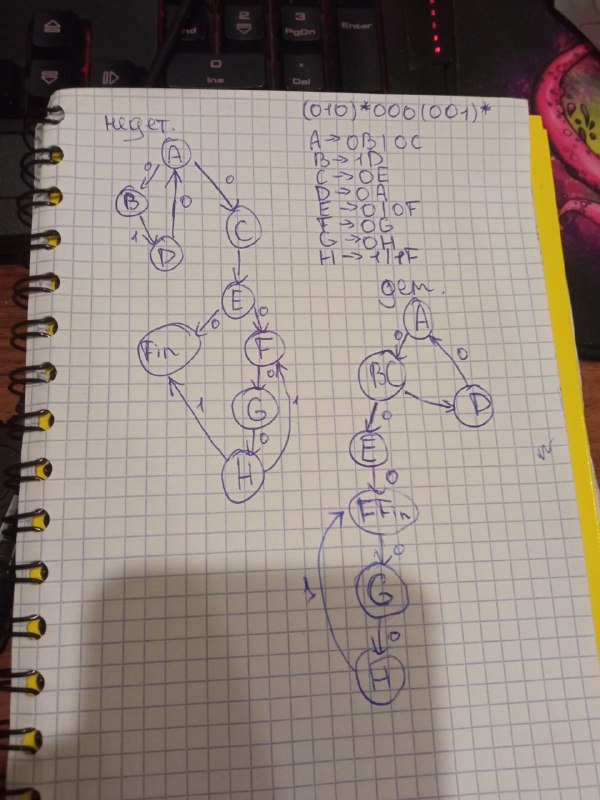
**Недетерминированная матрица:**

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| A | B, C |  |
| B |  | D |
| C | E |  |
| D | A |  |
| E | F,Fin |  |
| F | G |  |
| G | H |  |
| H |  | F,Fin |
| Fin |  |  |

**Детерминированная матрица:**

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| A | BC |  |
| BC | E | D |
| D | A |  |
| E | FFin |  |
| FFin | G |  |
| G | H |  |
| H |  | FFin |

**Графы:**



**Второе слово:**

(a|b|c|d)+

Первые два символа всегда ca

A → cB

B → a | aС

C → a | b | c | d | aC | bC | cC | dC

**Недетерминированная матрица:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | c | d |
| A |  |  | B |  |
| B | C,Fin |  |  |  |
| C | C,Fin | C,Fin | C,Fin | C,Fin |
| Fin |  |  |  |  |

**Детерминированная матрица:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | c | d |
| A |  |  | B |  |
| B | CFin |  |  |  |
| CFin | CFin | CFin | CFin | CFin |

**Код программы:**

**Form1.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace mazitov

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

// Инициализация текстового поля tbFSource

tbFSource.AppendText("010000001" + "\r\n");

tbFSource.AppendText("caab");

int n = tbFSource.Lines.Length;

}

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

{

}

private void btnFStart\_Click\_1(object sender, EventArgs e)

{

// Создание экземпляра лексического анализатора

CLex Lex = new CLex();

Lex.strPSource = tbFSource.Lines;

Lex.strPMessage = tbFMessage.Lines;

int x = tbFSource.TextLength;

int y = tbFSource.Lines.Length;

tbFMessage.Text = "";

try

{

Lex.GetSymbol(); // Выводятся литеры и классификация

while (Lex.enumPState != TState.Finish)

{

Lex.NextToken();

String s = "";

String s1 = "";

switch (Lex.enumFSelectionCharType)

{

case TCharType.Letter: { s1 = "Letter"; break; }

case TCharType.Digit: { s1 = "Digit"; break; }

case TCharType.Space: { s1 = "Space"; break; }

case TCharType.EndRow: { s = "KC"; s1 = "EndRow"; break; }

case TCharType.EndText: { s = "KT"; s1 = "EndText"; break; }

case TCharType.ReservedSymbol: {s = "ReservedSymbol"; break; }

case TCharType.LBracket: { s = "Left Bracket"; break; }

case TCharType.RBracket: { s = "Right Bracket"; break; }

case TCharType.Exclamation: { s = "Exclamation"; break; }

case TCharType.Question: { s = "Question"; break; }

case TCharType.Semicolon: { s = "Semicolon"; break; }

case TCharType.Comma: { s = "Comma"; break; }

case TCharType.Dot: { s = "Dot"; break; }

case TCharType.Star: { s = "Star"; break; }

}

switch(Lex.enumPToken)

{

case TToken.lxmNumber: { s = "LxmNumber"; s1 = Lex.strPLexicalUnit; break; }

case TToken.lxmIdentifier: { s = "lxmId"; s1 = Lex.strPLexicalUnit; break; }

}

String m = "(" + s + "," + s1 + ")"; //литера и ее тип

tbFMessage.Text += m; //добавляется в строку сообщение

}

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Lex.intPSourceRowSelection; i++) n += tbFSource.Lines[i].Length + 2;

n += Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

}

}

**uLex.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace nsLex

{

public enum TState { Start, Continue, Finish }; //тип состояния

public enum TCharType { Letter, Digit, EndRow, EndText, Space, ReservedSymbol }; // тип символа

public enum TToken { lxmIdentifier, lxmNumber, lxmUnknown, lxmEmpty, lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmSpace, lxmText, lxmtz, lxmdt, lxmr, lxmrs, lxmls, lxmUl };

public class CLex //класс лексический анализатор

{

private String[] strFSource; // указатель на массив строк

private String[] strFMessage; // указатель на массив строк

public TCharType enumFSelectionCharType;

public char chrFSelection;

private TState enumFState;

private int intFSourceRowSelection;

private int intFSourceColSelection;

private String strFLexicalUnit;

private TToken enumFToken;

public String[] strPSource { set { strFSource = value; } get { return strFSource; } }

public String[] strPMessage { set { strFMessage = value; } get { return strFMessage; } }

public TState enumPState { set { enumFState = value; } get { return enumFState; } }

public String strPLexicalUnit { set { strFLexicalUnit = value; } get { return strFLexicalUnit; } }

public TToken enumPToken { set { enumFToken = value; } get { return enumFToken; } }

public int intPSourceRowSelection { get { return intFSourceRowSelection; } set { intFSourceRowSelection = value; } }

public int intPSourceColSelection { get { return intFSourceColSelection; } set { intFSourceColSelection = value; } }

public CLex()

{

}

public void GetSymbol() //метод класса лексический анализатор

{

intFSourceColSelection++; // продвигаем номер колонки

if (intFSourceColSelection > strFSource[intFSourceRowSelection].Length - 1)

{

intFSourceRowSelection++;

if (intFSourceRowSelection <= strFSource.Length - 1)

{

intFSourceColSelection = -1;

chrFSelection = '\0';

enumFSelectionCharType = TCharType.EndRow;

enumFState = TState.Continue;

}

else

{

chrFSelection = '\0';

enumFSelectionCharType = TCharType.EndText;

enumFState = TState.Finish;

}

}

else

{

chrFSelection = strFSource[intFSourceRowSelection][intFSourceColSelection]; //классификация прочитанной литеры

if (chrFSelection == ' ') enumFSelectionCharType = TCharType.Space;

else if (chrFSelection >= 'a' && chrFSelection <= 'd') enumFSelectionCharType = TCharType.Letter;

else if (chrFSelection == '0' || chrFSelection == '1') enumFSelectionCharType = TCharType.Digit;

else if (chrFSelection == '/') enumFSelectionCharType = TCharType.ReservedSymbol;

else if (chrFSelection == '\*') enumFSelectionCharType = TCharType.ReservedSymbol;

else if (chrFSelection == ';' || chrFSelection == ',' || chrFSelection == '[' || chrFSelection == ']' || chrFSelection == '=' || chrFSelection == ':' || chrFSelection == '\_') enumFSelectionCharType = TCharType.ReservedSymbol;

else throw new System.Exception("Cимвол вне алфавита");

enumFState = TState.Continue;

}

}

private void TakeSymbol()

{

char[] c = { chrFSelection };

String s = new string(c);

strFLexicalUnit += s;

GetSymbol();

}

public void NextToken()

{

strFLexicalUnit = "";

if (enumFState == TState.Start)

{

intFSourceRowSelection = 0;

intFSourceColSelection = -1;

GetSymbol();

}

while (enumFSelectionCharType == TCharType.Space || enumFSelectionCharType == TCharType.EndRow)

{

GetSymbol();

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

// Вариант 14

switch (enumFSelectionCharType)

{

case TCharType.Letter:

{

// a b c d

// A | | | B | |

// B |CFin| | | |

// CFin |CFin|CFin|CFin|CFin|

A:

{

if (chrFSelection == 'c')

{

TakeSymbol();

goto B;

}

else throw new Exception("Слово должно начинаться с 'ca'");

}

B:

{

if (chrFSelection == 'a')

{

TakeSymbol();

goto CFin;

}

else throw new Exception("Слово должно начинаться с 'ca'");

}

CFin:

{

if (chrFSelection == 'a' || chrFSelection == 'b' || chrFSelection == 'c' || chrFSelection == 'd')

{

TakeSymbol();

goto CFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

case TCharType.Digit:

{

// 0 1

// A | BC | |

// BC | E | D |

// D | A | |

// E | FFin| |

// FFin | G | |

// G | H | |

// H | | FFin|

A:

if (chrFSelection == '0')

{

TakeSymbol();

goto BC;

}

else throw new Exception("Ожидался 0");

BC:

if (chrFSelection == '0')

{

TakeSymbol();

goto E;

}

else if (chrFSelection == '1')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался 0 или 1");

D:

if (chrFSelection == '0')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидался 0");

E:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидался 0");

FFin:

if (chrFSelection == '0')

{

TakeSymbol();

goto G;

}

else if (enumFSelectionCharType != TCharType.Digit) { enumFToken = TToken.lxmNumber; return; }

else throw new Exception("Ожидался 0");

G:

if (chrFSelection == '0')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался 0");

H:

if (chrFSelection == '1')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 1");

}

case TCharType.ReservedSymbol:

{

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

{

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

}

GetSymbol();

}

if (chrFSelection == '(')

{

enumFToken = TToken.lxmLeftParenth;

GetSymbol();

return;

}

if (chrFSelection == ')')

{

enumFToken = TToken.lxmRightParenth;

GetSymbol();

return;

}

if (chrFSelection == '[')

{

enumFToken = TToken.lxmls;

GetSymbol();

return;

}

if (chrFSelection == ']')

{

enumFToken = TToken.lxmrs;

GetSymbol();

return;

}

if (chrFSelection == ' ')

{

enumFToken = TToken.lxmSpace;

GetSymbol();

return;

}

if (chrFSelection == ':')

{

enumFToken = TToken.lxmdt;

GetSymbol();

return;

}

if (chrFSelection == '=')

{

enumFToken = TToken.lxmr;

GetSymbol();

return;

}

if (chrFSelection == '\_')

{

//strFLexicalUnit = "\_";

enumFToken = TToken.lxmUl;

TakeSymbol();

return;

}

break;

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

case TCharType.Space:

{

enumFToken = TToken.lxmSpace;

break;

}

}

}

}

}

# **Лабораторная работа № 3. Разработка контекстно-свободного (КС) синтаксического анализатора**

Синтаксический анализатор – это программное обеспечение, которое анализирует структуру текста и определяет его соответствие грамматике, что позволяет автоматически проверять и разбирать программный код, упрощая процесс разработки и отладки.

Синтаксический анализатор, также известный как парсер, является одной из ключевых компонент программного обеспечения, используемого в компиляторах и интерпретаторах. Он выполняет анализ входного текста, проверяет его на соответствие определенной грамматике и создает структуру данных, называемую синтаксическим деревом или абстрактным синтаксическим деревом (AST).

Синтаксический анализатор играет важную роль в процессе компиляции или интерпретации программного кода. Он принимает входной текст, который может быть написан на определенном языке программирования, и проверяет его на наличие синтаксических ошибок. Если текст соответствует грамматике языка, синтаксический анализатор создает структуру данных, которая представляет собой иерархическое представление программы.

Основная цель синтаксического анализатора – разбор входного текста и создание структуры данных, которая будет использоваться в дальнейшем для выполнения семантического анализа и генерации исполняемого кода. Он помогает программистам и разработчикам понять структуру программы и обнаружить возможные ошибки в синтаксисе.

Левая рекурсия - это ситуация в формальной грамматике, когда правило грамматики непосредственно или косвенно определяет символ через самого себя с левой рекурсией.

Левая рекурсия может привести к проблемам при построении анализаторов и интерпретаторов, так как они могут зациклиться при попытке разбора или выполнения правил с левой рекурсией. Поэтому часто требуется избавляться от левой рекурсии в грамматике.

В программировании левая рекурсия также может быть проблемой, особенно при реализации парсеров и компиляторов. Левая рекурсия в некоторых случаях может вызвать бесконечный цикл в рекурсивных вызовах функций, что приведет к переполнению стека и аварийному завершению программы.

**Текст задания:**

Для предложенного преподавателем варианта КС-грамматики разработать методом рекурсивного спуска синтаксический анализатор

Для предложенного преподавателем варианта КС-грамматики разработать методом рекурсивного спуска синтаксический анализатор.

Указания:

1. Лексический анализатор из лабораторной работы №1 должен быть расширен обработкой появившихся в КС-грамматике новых слов и включен в виде подпрограммы, поля класса или метода класса в синтаксический анализатор.

2. Оформить синтаксический анализатор в виде процедуры, функции или класса, которые при обращении обрабатывают весь исходный текст.

3. Если грамматика леворекурсивная, то устранить левую рекурсию.

4. При обнаружении лексической ошибки целесообразно возбуждать исключительную ситуацию, которая будет обрабатываться в главной форме программы.

**Грамматика:**

S → <2> | <1> A S

A → <1> | <2> S A

**Код программы:**

**Form1.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

namespace mazitov

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

tbFSource.AppendText("ca" + "\r\n");

int n = tbFSource.Lines.Length;

}

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

{

tbFMessage.Clear();

uSyntAnalyzer Synt = new uSyntAnalyzer();

Synt.Lex.strPSource = tbFSource.Lines;

Synt.Lex.strPMessage = tbFMessage.Lines;

Synt.Lex.enumPState = TState.Start;

try

{

Synt.Lex.NextToken();

Synt.S();

throw new Exception("Текст верный");

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Synt.Lex.intPSourceRowSelection; i++) n += tbFSource.Lines[i].Length + 2;

n += Synt.Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

}

}

**uLex.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace mazitov

{

public enum TState { Start, Continue, Finish }; //тип состояния

public enum TCharType { Letter, Digit, EndRow, EndText, Space, ReservedSymbol }; // тип символа

public enum TToken { lxmIdentifier, lxmNumber, lxmUnknown, lxmEmpty, lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmSpace, lxmText, lxmtz, lxmdt, lxmr, lxmrs, lxmls, lxmUl };

public class CLex //класс лексический анализатор

{

private String[] strFSource; // указатель на массив строк

private String[] strFMessage; // указатель на массив строк

public TCharType enumFSelectionCharType;

public char chrFSelection;

private TState enumFState;

private int intFSourceRowSelection;

private int intFSourceColSelection;

private String strFLexicalUnit;

private TToken enumFToken;

public String[] strPSource { set { strFSource = value; } get { return strFSource; } }

public String[] strPMessage { set { strFMessage = value; } get { return strFMessage; } }

public TState enumPState { set { enumFState = value; } get { return enumFState; } }

public String strPLexicalUnit { set { strFLexicalUnit = value; } get { return strFLexicalUnit; } }

public TToken enumPToken { set { enumFToken = value; } get { return enumFToken; } }

public int intPSourceRowSelection { get { return intFSourceRowSelection; } set { intFSourceRowSelection = value; } }

public int intPSourceColSelection { get { return intFSourceColSelection; } set { intFSourceColSelection = value; } }

public CLex()

{

}

public void GetSymbol() //метод класса лексический анализатор

{

intFSourceColSelection++; // продвигаем номер колонки

if (intFSourceColSelection > strFSource[intFSourceRowSelection].Length - 1)

{

intFSourceRowSelection++;

if (intFSourceRowSelection <= strFSource.Length - 1)

{

intFSourceColSelection = -1;

chrFSelection = '\0';

enumFSelectionCharType = TCharType.EndRow;

enumFState = TState.Continue;

}

else

{

chrFSelection = '\0';

enumFSelectionCharType = TCharType.EndText;

enumFState = TState.Finish;

}

}

else

{

chrFSelection = strFSource[intFSourceRowSelection][intFSourceColSelection]; //классификация прочитанной литеры

if (chrFSelection == ' ') enumFSelectionCharType = TCharType.Space;

else if (chrFSelection >= 'a' && chrFSelection <= 'd') enumFSelectionCharType = TCharType.Letter;

else if (chrFSelection == '0' || chrFSelection == '1') enumFSelectionCharType = TCharType.Digit;

else if (chrFSelection == '/') enumFSelectionCharType = TCharType.ReservedSymbol;

else if (chrFSelection == '\*') enumFSelectionCharType = TCharType.ReservedSymbol;

else if (chrFSelection == ';' || chrFSelection == ',' || chrFSelection == '[' || chrFSelection == ']' || chrFSelection == '=' || chrFSelection == ':' || chrFSelection == '\_') enumFSelectionCharType = TCharType.ReservedSymbol;

else throw new System.Exception("Cимвол вне алфавита");

enumFState = TState.Continue;

}

}

private void TakeSymbol()

{

char[] c = { chrFSelection };

String s = new string(c);

strFLexicalUnit += s;

GetSymbol();

}

public void NextToken()

{

strFLexicalUnit = "";

if (enumFState == TState.Start)

{

intFSourceRowSelection = 0;

intFSourceColSelection = -1;

GetSymbol();

}

while (enumFSelectionCharType == TCharType.Space || enumFSelectionCharType == TCharType.EndRow)

{

GetSymbol();

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

// Вариант 14

switch (enumFSelectionCharType)

{

case TCharType.Letter:

{

// a b c d

// A | | | B | |

// B |CFin| | | |

// CFin |CFin|CFin|CFin|CFin|

A:

{

if (chrFSelection == 'c')

{

TakeSymbol();

goto B;

}

else throw new Exception("Слово должно начинаться с 'ca'");

}

B:

{

if (chrFSelection == 'a')

{

TakeSymbol();

goto CFin;

}

else throw new Exception("Слово должно начинаться с 'ca'");

}

CFin:

{

if (chrFSelection == 'a' || chrFSelection == 'b' || chrFSelection == 'c' || chrFSelection == 'd')

{

TakeSymbol();

goto CFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

case TCharType.Digit:

{

// 0 1

// A | BC | |

// BC | E | D |

// D | A | |

// E | FFin| |

// FFin | G | |

// G | H | |

// H | | FFin|

A:

if (chrFSelection == '0')

{

TakeSymbol();

goto BC;

}

else throw new Exception("Ожидался 0");

BC:

if (chrFSelection == '0')

{

TakeSymbol();

goto E;

}

else if (chrFSelection == '1')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался 0 или 1");

D:

if (chrFSelection == '0')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидался 0");

E:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидался 0");

FFin:

if (chrFSelection == '0')

{

TakeSymbol();

goto G;

}

else if (enumFSelectionCharType != TCharType.Digit) { enumFToken = TToken.lxmNumber; return; }

else throw new Exception("Ожидался 0");

G:

if (chrFSelection == '0')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался 0");

H:

if (chrFSelection == '1')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 1");

}

case TCharType.ReservedSymbol:

{

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

{

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

}

GetSymbol();

}

if (chrFSelection == '(')

{

enumFToken = TToken.lxmLeftParenth;

GetSymbol();

return;

}

if (chrFSelection == ')')

{

enumFToken = TToken.lxmRightParenth;

GetSymbol();

return;

}

if (chrFSelection == '[')

{

enumFToken = TToken.lxmls;

GetSymbol();

return;

}

if (chrFSelection == ']')

{

enumFToken = TToken.lxmrs;

GetSymbol();

return;

}

if (chrFSelection == ' ')

{

enumFToken = TToken.lxmSpace;

GetSymbol();

return;

}

if (chrFSelection == ':')

{

enumFToken = TToken.lxmdt;

GetSymbol();

return;

}

if (chrFSelection == '=')

{

enumFToken = TToken.lxmr;

GetSymbol();

return;

}

if (chrFSelection == '\_')

{

enumFToken = TToken.lxmUl;

GetSymbol();

return;

}

break;

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

case TCharType.Space:

{

enumFToken = TToken.lxmSpace;

break;

}

}

}

}

}

**uSyntAnalyzer.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace mazitov

{

class uSyntAnalyzer

{

private String[] strFSource;

private String[] strFMessage;

public String[] strPSource { set { strFSource = value; } get { return strFSource; } }

public String[] strPMessage { set { strFMessage = value; } get { return strFMessage; } }

public CLex Lex = new CLex();

public void S()

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

A();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

S();

{

throw new Exception("Ожидалось cлово");

}

}

else throw new Exception("Ожидалось число или слово");

}

}

else throw new Exception("Ожидалось число или слово");

}

else throw new Exception("Ожидалось нижнее подчеркивание");

}

else throw new Exception("Ожидалось число или слово");

}

}

public void A()

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

S();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

A();

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber)

{

throw new Exception("Ожидалось число");

}

}

else throw new Exception("Ожидалось число или слово");

}

}

else throw new Exception("Ожидалось число или слово");

}

else throw new Exception("Ожидалось нижнее подчеркивание");

}

else throw new Exception("Ожидалось число или слово");

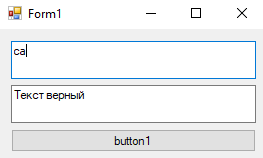
}

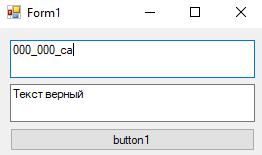
}

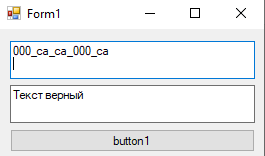
}

}

**Результат:**







# **Лабораторная работа № 4. Введение табличного способа хранения слов**

Табличный способ хранения слов — это метод организации данных, при котором слова или термины хранятся в виде таблицы, где каждое слово имеет свою строку или запись, а каждый атрибут или характеристика слова представляет собой столбец или поле в таблице. Этот метод часто используется в лингвистике, лексикографии и информационных системах для удобного хранения и поиска лексической информации.

В табличном способе хранения слов каждая запись может содержать различные атрибуты, такие как слово, его часть речи, определение, синонимы, антонимы, примеры использования и другие лингвистические данные. Это обеспечивает структурированное хранение информации о словах и облегчает поиск и анализ лексических данных.

Табличный способ хранения слов также может быть использован для создания словарей, лексиконов, тезаурусов и других лингвистических ресурсов. Он позволяет эффективно организовывать большие объемы лексической информации и обеспечивает удобный доступ к этим данным для исследования, обучения или других целей.

Хэш-таблица (или хеш-таблица) — это структура данных, которая используется для организации и управления данными. Основной идеей хэш-таблицы является использование хеш-функции для быстрого и эффективного поиска, вставки и удаления элементов. Хеш-функция преобразует ключи (например, строки или числа) в индексы (хеши), по которым элементы хранятся в массиве (бакете) с целью обеспечения быстрого доступа к ним. Хорошо спроектированная хеш-таблица может обеспечивать среднюю сложность O(1) для операций вставки, поиска и удаления.

Хеширование — это процесс применения хеш-функции к некоторым данным для получения уникального значения (хеш-кода или хеша), которое служит идентификатором для этих данных. Цель хеширования заключается в том, чтобы ускорить поиск элементов в коллекциях данных, таких как хеш-таблицы, используя хеш-код в качестве индекса для доступа к данным в массиве. Эффективное хеширование требует, чтобы хеш-функция была быстрой и обеспечивала равномерное распределение хешей для минимизации коллизий (когда разные ключи приводят к одному и тому же хешу).

Под коллизией понимается ситуация, когда при добавлении разных объектов мы попадаем в одну и ту же ячейку массива. Для разрешения коллизий придумано 2 метода: метод цепочек и метод открытой адресации.

Коллизии существуют для большинства хеш-функций, но для «хороших» хеш-функций частота их возникновения близка к теоретическому минимуму. В некоторых частных случаях, когда множество различных входных данных конечно, можно задать инъективную хеш-функцию, по определению не имеющую коллизий. Однако для хеш-функций, принимающих вход переменной длины и возвращающих хеш постоянной длины, коллизии обязаны существовать, поскольку хотя бы для одного значения хеш-функции соответствующее ему множество входных данных будет бесконечно — и любые два набора данных из этого множества образуют коллизию.

В программировании коллизии (или столкновения) могут возникать при использовании хэш-таблиц, методов хэширования, при работе с базами данных, а также в других контекстах. Коллизии возникают, когда два различных входа приводят к одному и тому же значению хэша. Существуют различные методы для управления и избавления от коллизий:

1. Метод цепочек (Separate Chaining):

Каждая ячейка (бакет) хэш-таблицы представляет собой связанный список. Если возникает коллизия, новый элемент добавляется в соответствующий связанный список. Этот метод эффективен при небольшой загрузке таблицы.

1. Открытая адресация (Open Addressing):

Этот метод позволяет сохранять элементы непосредственно в ячейках таблицы. Если происходит коллизия, новый элемент помещается в следующую свободную ячейку, используя определенное правило. Методы открытой адресации включают линейное пробирование, квадратичное пробирование и двойное хэширование.

1. Линейное пробирование (Linear Probing):

При коллизии новый элемент помещается в следующую ячейку в порядке следования. Если эта ячейка занята, происходит переход к следующей свободной ячейке. Процесс повторяется, пока не найдется свободная ячейка.

1. Квадратичное пробирование (Quadratic Probing):

Этот метод изменяет шаг пробирования квадратичным образом. Вместо линейного шага i, используется шаг i2.

Это может помочь избежать сгустков элементов, которые могут возникнуть при линейном пробировании.

1. Двойное хэширование (Double Hashing):

Вместо постоянного шага пробирования, используется вторая хэш-функция для определения шага. Это позволяет более равномерно распределить элементы по ячейкам и избежать группировки.

1. Универсальное хэширование (Universal Hashing):

Используется семейство хэш-функций, из которого выбирается конкретная функция случайным образом при создании таблицы. Это позволяет равномерно распределить элементы и уменьшить вероятность коллизий.

**Текст задания:**

1. Подключить класс «Массив хеш-таблиц» к программе.
2. Завести три таблицы для хранения слов первого типа, слов второго типа и служебных слов (многосимвольных)
3. Отладить программу до рабочего состояния.

**Код программы:**

**Form1.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

using nsLex;

using nsSynt;

using nsHashTables;

namespace mazitov

{

public partial class Form1 : Form

{

public CHashTableList htl = new CHashTableList(3);

public Form1()

{

InitializeComponent();

tbFSource.AppendText("000\_ca\_cad\_000\_cab" + "\r\n");

int n = tbFSource.Lines.Length;

}

public void TablesToMemo(object sender, System.EventArgs e)

{

List<string> listTable = new List<string>();

listBox1.Items.Clear();

listBox2.Items.Clear();

listBox3.Items.Clear();

htl.TableToStringList(0, listTable);

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

listBox1.Items.Add(i.ToString() + " " + listTable[i]);

listTable.Clear();

htl.TableToStringList(1, listTable);

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

listBox2.Items.Add(i.ToString() + " " + listTable[i]);

listTable.Clear();

htl.TableToStringList(2, listTable);

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

listBox3.Items.Add(i.ToString() + " " + listTable[i]);

listTable.Clear();

}

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

{

tbFMessage.Clear();

uSyntAnalyzer Synt = new uSyntAnalyzer();

Synt.Lex.strPSource = tbFSource.Lines;

Synt.Lex.strPMessage = tbFMessage.Lines;

Synt.Lex.enumPState = TState.Start;

try

{

Synt.Lex.NextToken();

Synt.S();

throw new Exception("Текст верный");

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Synt.Lex.intPSourceRowSelection; i++) n += tbFSource.Lines[i].Length + 2;

n += Synt.Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

private void button1\_Click(object sender, EventArgs e)

{

CLex Lex = new CLex();

Lex.strPSource = tbFSource.Lines;

Lex.strPMessage = tbFMessage.Lines;

Lex.intPSourceColSelection = 0;

Lex.intPSourceRowSelection = 0;

int x = tbFSource.TextLength;

int y = tbFSource.Lines.Length;

tbFMessage.Text = "";

try

{

while (Lex.enumPState != TState.Finish)

{

Lex.NextToken();

string s1 = "", s = "";

switch (Lex.enumPToken)

{

case TToken.lxmIdentifier:

{

s1 = "id " + Lex.strPLexicalUnit; int b = 0;

if (htl.AddLexicalUnit(Lex.strPLexicalUnit, 0, ref b))

{

TablesToMemo(this, e);

}

break;

}

case TToken.lxmNumber:

{

s1 = "num " + Lex.strPLexicalUnit; int b = 0;

if (htl.AddLexicalUnit(Lex.strPLexicalUnit, 1, ref b))

{

TablesToMemo(this, e);

}

break;

}

case TToken.lxmUl:

{

if (Lex.enumFSelectionCharType == TCharType.EndText)

{

s1 = "EndRow";

break;

}

s1 = "Probel " + Lex.strPLexicalUnit; int b = 0;

if (htl.AddLexicalUnit(Lex.strPLexicalUnit, 2, ref b))

{

TablesToMemo(this, e);

}

break;

}

}

String m = "(" + s + "" + s1 + ")";

tbFMessage.Text += m;

}

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Lex.intPSourceRowSelection; i++) n += tbFSource.Lines[i].Length + 2;

n += Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

private void button2\_Click(object sender, EventArgs e)

{

{

CLex Lex = new CLex();

Lex.strPSource = tbFSource.Lines;

Lex.strPMessage = tbFMessage.Lines;

Lex.intPSourceColSelection = 0;

Lex.intPSourceRowSelection = 0;

int x = tbFSource.TextLength;

int y = tbFSource.Lines.Length;

tbFMessage.Text = "";

try

{

while (Lex.enumPState != TState.Finish)

{

Lex.NextToken();

string s1 = "", s = "";

switch (Lex.enumPToken)

{

case TToken.lxmIdentifier:

{

s1 = "id " + Lex.strPLexicalUnit;

htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 0);

// htl.DeleteLexicalCode(0);

TablesToMemo(this, e);

break;

}

case TToken.lxmNumber:

{

s1 = "num " + Lex.strPLexicalUnit;

htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 1);

// htl.DeleteLexicalCode(0);

TablesToMemo(this, e);

break;

}

case TToken.lxmUl:

{

s1 = "Probel " + Lex.strPLexicalUnit; int b = 0;

htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 2);

TablesToMemo(this, e);

break;

}

}

String m = "(" + s + "" + s1 + ")";

tbFMessage.Text += m;

}

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Lex.intPSourceRowSelection; i++) n += tbFSource.Lines[i].Length + 2;

n += Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

}

private void button3\_Click(object sender, EventArgs e)

{

{

CLex Lex = new CLex();

Lex.strPSource = textBox1.Lines;

Lex.strPMessage = tbFMessage.Lines;

Lex.intPSourceColSelection = 0;

Lex.intPSourceRowSelection = 0;

CLex Lex2 = new CLex();

Lex2.strPSource = textBox2.Lines;

Lex2.strPMessage = tbFMessage.Lines;

Lex2.intPSourceColSelection = 0;

Lex2.intPSourceRowSelection = 0;

int x = textBox1.TextLength;

int y = textBox2.Lines.Length;

tbFMessage.Text = "";

try

{

while (Lex.enumPState != TState.Finish)

{

Lex.NextToken();

switch (Lex.enumPToken)

{

case TToken.lxmIdentifier:

{

int b = 0;

if (htl.SearchLexicalUnit(Lex.strPLexicalUnit, 0, ref b))

{

while (Lex2.enumPState != TState.Finish)

{

Lex2.NextToken();

switch (Lex2.enumPToken)

{

case TToken.lxmIdentifier:

{

int b1 = 0;

if (htl.AddLexicalUnit(Lex2.strPLexicalUnit, 0, ref b1))

{

htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 0);

TablesToMemo(this, e);

}

tbFMessage.AppendText("(Индефикатор изменен)");

break;

}

case TToken.lxmNumber:

{

tbFMessage.AppendText("(Нельзя заменить Индефикатор на число)");

break;

}

}

}

}

else tbFMessage.AppendText("(Индефикатор для изменения не найден)");

break;

}

case TToken.lxmNumber:

{

int b = 0;

if (htl.SearchLexicalUnit(Lex.strPLexicalUnit, 1, ref b))

{

while (Lex2.enumPState != TState.Finish)

{

Lex2.NextToken();

switch (Lex2.enumPToken)

{

case TToken.lxmNumber:

{

int b1 = 0;

if (htl.AddLexicalUnit(Lex2.strPLexicalUnit, 1, ref b1))

{

htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 1);

TablesToMemo(this, e);

}

tbFMessage.AppendText("(Число изменено)");

break;

}

case TToken.lxmIdentifier:

{

tbFMessage.AppendText("(Нельзя заменить число на Индефикатор)");

break;

}

}

}

}

else tbFMessage.AppendText("(Число для изменения не найдено)");

break;

}

}

}

}

catch (Exception exc)

{

textBox1.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Lex.intPSourceRowSelection; i++) n += tbFSource.Lines[i].Length + 2;

n += Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

}

private void button4\_Click(object sender, EventArgs e)

{

CLex Lex = new CLex();

Lex.strPSource = tbFSource.Lines;

Lex.strPMessage = tbFMessage.Lines;

Lex.intPSourceColSelection = 0;

Lex.intPSourceRowSelection = 0;

int x = tbFSource.TextLength;

int y = tbFSource.Lines.Length;

tbFMessage.Text = "";

try

{

while (Lex.enumPState != TState.Finish)

{

Lex.NextToken();

string s1 = "", s = "";

switch (Lex.enumPToken)

{

case TToken.lxmIdentifier:

{

s1 = "id " + Lex.strPLexicalUnit; int b = 0;

if (htl.SearchLexicalUnit(Lex.strPLexicalUnit, 0, ref b))

{

tbFMessage.AppendText("(Найден идентификатор)");

}

else tbFMessage.AppendText("(Идентификатор не найден)");

break;

}

case TToken.lxmNumber:

{

s1 = "num " + Lex.strPLexicalUnit; int b = 0;

if (htl.SearchLexicalUnit(Lex.strPLexicalUnit, 1, ref b))

{

tbFMessage.AppendText("(Найден номер)");

}

else tbFMessage.AppendText("(Номер не найден)");

break;

}

case TToken.lxmUl:

{

s1 = "Probel " + Lex.strPLexicalUnit; int b = 0;

if (htl.SearchLexicalUnit(Lex.strPLexicalUnit, 2, ref b))

{

tbFMessage.AppendText("(Найден символ)");

}

else tbFMessage.AppendText("(Символ не найден)");

break;

}

}

String m = "(" + s + "" + s1 + ")";

tbFMessage.Text += m;

}

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Lex.intPSourceRowSelection; i++) n += tbFSource.Lines[i].Length + 2;

n += Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

private void button5\_Click(object sender, EventArgs e)

{

List<string> listTable = new List<string>();

CLex Lex = new CLex();

Lex.strPSource = textBox1.Lines;

Lex.strPMessage = tbFMessage.Lines;

Lex.intPSourceColSelection = 0;

Lex.intPSourceRowSelection = 0;

int x = textBox1.TextLength;

int y = textBox1.Lines.Length;

tbFMessage.Text = "";

try

{

while (Lex.enumPState != TState.Finish)

{

Lex.NextToken();

string s1 = "", s = "";

switch (Lex.enumPToken)

{

case TToken.lxmIdentifier:

{

s1 = "id " + Lex.strPLexicalUnit; int b = 0;

if (htl.SearchLexicalUnit(Lex.strPLexicalUnit, 0, ref b))

{

tbFMessage.AppendText("(Найден идентификатор)");

}

else tbFMessage.AppendText("(Номер не идентификатор)");

break;

}

case TToken.lxmNumber:

{

s1 = "num " + Lex.strPLexicalUnit; int b = 0;

if (htl.SearchLexicalUnit(Lex.strPLexicalUnit, 1, ref b))

{

tbFMessage.AppendText("(Найден номер)");

}

else tbFMessage.AppendText("(Номер не найден)");

break;

}

case TToken.lxmUl:

{

s1 = "Probel " + Lex.strPLexicalUnit; int b = 0;

if (htl.SearchLexicalUnit(Lex.strPLexicalUnit, 2, ref b))

{

tbFMessage.AppendText("(Найден символ)");

}

else tbFMessage.AppendText("(Символ не найден)");

break;

}

}

String m = "(" + s + "" + s1 + ")";

tbFMessage.Text += m;

}

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

textBox1.Select();

textBox1.SelectionStart = 0;

int n = 0;

for (int i = 0; i < Lex.intPSourceRowSelection; i++) n += textBox1.Lines[i].Length + 2;

n += Lex.intPSourceColSelection;

textBox1.SelectionLength = n;

}

}

}

}

**uSyntAnalyzer.cs**

using System;

using System.Collections.Generic;

using System.Text;

using nsLex;

namespace nsSynt

{

class uSyntAnalyzer

{

private String[] strFSource;

private String[] strFMessage;

public String[] strPSource { set { strFSource = value; } get { return strFSource; } }

public String[] strPMessage { set { strFMessage = value; } get { return strFMessage; } }

public CLex Lex = new CLex();

public void S()

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

A();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

S();

{

throw new Exception("Ожидалось cлово");

}

}

else throw new Exception("Ожидалось число или слово");

}

}

else throw new Exception("Ожидалось число или слово");

}

else throw new Exception("Ожидалось пробел (нижнее подчеркивание)");

}

else throw new Exception("Ожидалось число или слово");

}

}

public void A()

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

S();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmIdentifier)

{

A();

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber)

{

throw new Exception("Ожидалось число");

}

}

else throw new Exception("Ожидалось число или слово");

}

}

else throw new Exception("Ожидалось число или слово");

}

else throw new Exception("Ожидалось пробел (нижнее подчеркивание)");

}

else throw new Exception("Ожидалось число или слово");

}

}

}

}

**Hash.Tables.cs**

using System;

using System.Collections.Generic;

using System.IO;

using System.Windows.Forms;

using System.Linq;

namespace nsHashTables

{

public class THashTable

{

public List<int> arrFHashTable = new List<int>();

private int intFCurrentPrimeNumber;

private int intFItemReserve;

private bool boolIsSaved;

public int intFHashIndex;

public int cardPTableSize { get { return arrFHashTable.Count; } }

public List<object> arrFUserTable = null;

static THeap objFHeap;

public THashTable(ref THeap objAHeap)

{

objFHeap = objAHeap;

Init(7);

intFItemReserve = 0;

}

public void Init(int count)

{

arrFHashTable.Clear();

Resize(arrFHashTable, count);

intFCurrentPrimeNumber = count;

}

static void Resize(List<object> list, int size)

{

if (size > list.Count)

while (size > list.Count)

list.Add(new object());

else if (size < list.Count)

while (list.Count - size > 0)

list.RemoveAt(list.Count - 1);

}

static void Resize(List<int> list, int size)

{

if (size > list.Count)

while (size > list.Count)

list.Add(new Int32());

else if (size < list.Count)

while (list.Count - size > 0)

list.RemoveAt(list.Count - 1);

}

int NextPrimeNumber(int cardAOldPrimeNumber)

{

int intVLowerBound, intVUpperBound, intVNextPrimeNumber;

bool boolVIsDivisor;

intVNextPrimeNumber = cardAOldPrimeNumber + cardAOldPrimeNumber / 10 + 1; // увеличиваем на 10 процентов

if ((intVNextPrimeNumber % 2) == 0) intVNextPrimeNumber++;

do

{

boolVIsDivisor = true; intVNextPrimeNumber = intVNextPrimeNumber + 2;

intVLowerBound = 3; intVUpperBound = intVNextPrimeNumber / 3 + 1; // диапазон делителей

while (boolVIsDivisor && (intVLowerBound < intVUpperBound))

{

if ((intVNextPrimeNumber % intVLowerBound) == 0) boolVIsDivisor = false;

else intVLowerBound = intVLowerBound + 2;

}

} while (!boolVIsDivisor);

return intVNextPrimeNumber;

}

UInt32 HashFunction(string strALexicalUnit)

{

UInt32 hash = 5381;

foreach (char c in strALexicalUnit)

{

hash = ((hash << 5) + hash) ^ c;

}

return hash;

}

int ReHashFunction\_Line(int h, string strALexicalUnit)

{

if (h == 0) h = arrFHashTable.Count / 3;

else if (h == 1) h = arrFHashTable.Count \* 3 / 4;

int i = 1, hi = h;

bool boolVFinish = false;

do

{

if (arrFHashTable[hi] == 0) boolVFinish = true;

else

if (objFHeap.arrFHeapTable[arrFHashTable[hi]].strFLexicalUnit == strALexicalUnit)

{

boolVFinish = true;

}

else

{

i++;

hi = h \* (i + 1) % (arrFHashTable.Count);

}

} while (!boolVFinish);

return hi;

}

public void HashIndex(string strALexicalUnit)

{

int h;

h = (Int32)HashFunction(strALexicalUnit) % (Int32)(arrFHashTable.Count);

intFHashIndex = ReHashFunction\_Line(h, strALexicalUnit);

}

void TableReHashing()

{

int i, j;

List<int> cardarrVHashTableImage = new List<int>();

List<object> arrVUserTableImage = new List<object>();

Resize(cardarrVHashTableImage, arrFHashTable.Count);

if (arrFUserTable != null)

Resize(arrVUserTableImage, arrFHashTable.Count);

for (i = 0; i < arrFHashTable.Count; i++)

{

cardarrVHashTableImage[i] = arrFHashTable[i];

if (arrFUserTable != null) arrVUserTableImage[i] = arrFUserTable[i];

}

arrFHashTable.Clear();

if (arrFUserTable != null) arrFUserTable.Clear();

Resize(arrFHashTable, intFCurrentPrimeNumber);

if (arrFUserTable != null) Resize(arrFUserTable, intFCurrentPrimeNumber);

for (i = 0; i < cardarrVHashTableImage.Count; i++)

{

if (cardarrVHashTableImage[i] != 0)

{

j = cardarrVHashTableImage[i];

HashIndex(objFHeap.arrFHeapTable[j].strFLexicalUnit);

arrFHashTable[intFHashIndex] = j;

if (arrFUserTable != null) arrFUserTable[intFHashIndex] = arrVUserTableImage[i];

THeapItem Th2 = objFHeap.arrFHeapTable[j];

Th2.intFHashIndex = intFHashIndex;

objFHeap.arrFHeapTable[j] = Th2;

}

}

cardarrVHashTableImage.Clear();

if (arrFUserTable != null) arrVUserTableImage.Clear();

}

void Expansion()

{

intFCurrentPrimeNumber = NextPrimeNumber(intFCurrentPrimeNumber);

TableReHashing();

}

object GetUserPointer(int cardILexicalCode)

{

THeapItem Item = objFHeap.arrFHeapTable[cardILexicalCode];

if (Item.intFHashIndex >= cardPTableSize)

{

MessageBox.Show("Индекс пользовательского массива вышел за диапазон!");

return null;

}

else

{

return arrFUserTable[objFHeap.arrFHeapTable[cardILexicalCode].intFHashIndex];

}

}

void SetUserPointer(int cardILexicalCode, object ptrANewPoint)

{

if (objFHeap.arrFHeapTable[cardILexicalCode].intFHashIndex >= cardPTableSize)

MessageBox.Show("Индекс пользовательского массива вышел за диапазон!");

else

arrFUserTable[objFHeap.arrFHeapTable[cardILexicalCode].intFHashIndex] = ptrANewPoint;

}

public void SetUserTable()

{

arrFUserTable = new List<object>();

Resize(arrFUserTable, arrFHashTable.Count);

}

public bool SearchLexicalUnit(string strAlexicalUnit, ref int intALexicalCode)

{

HashIndex(strAlexicalUnit);

if (arrFHashTable[intFHashIndex] == 0) return false;

else

{

intALexicalCode = arrFHashTable[intFHashIndex];

return true;

}

}

public bool AddLexicalUnit(string strALexicalUnit, byte byteAHashTable, ref int intALexicalCode)

{

HashIndex(strALexicalUnit);

if (arrFHashTable[intFHashIndex] != 0)

{

intALexicalCode = arrFHashTable[intFHashIndex];

return true;

}

else

{

if ((intFItemReserve + 2) > (cardPTableSize \* 0.9))

{

Expansion();

HashIndex(strALexicalUnit);

}

objFHeap.AddLexicalUnit(strALexicalUnit, byteAHashTable, intFHashIndex, ref intALexicalCode);

arrFHashTable[intFHashIndex] = intALexicalCode;

intFItemReserve++;

return false;

}

}

public void DeleteLexicalUnit(string strAlexicalUnit)

{

HashIndex(strAlexicalUnit);

if (arrFHashTable[intFHashIndex] != 0)

{

if (arrFUserTable != null)

{

if (arrFUserTable[intFHashIndex] != null)

MessageBox.Show("Удаление из таблицы связанного данного");

else

{

objFHeap.DeleteLexicalUnit(arrFHashTable[intFHashIndex]);

arrFHashTable[intFHashIndex] = 0;

intFItemReserve--;

TableReHashing();

}

}

else

{

objFHeap.DeleteLexicalUnit(arrFHashTable[intFHashIndex]);

arrFHashTable[intFHashIndex] = 0;

intFItemReserve--;

TableReHashing();

}

}

}

public void DeleteLexicalCode(int cardALexicalCode)

{

int VHashIndex;

VHashIndex = objFHeap.arrFHeapTable[cardALexicalCode].intFHashIndex;

if (arrFHashTable[VHashIndex] != 0)

if (arrFUserTable.Count != 0)

if (arrFUserTable[VHashIndex] != null)

MessageBox.Show("Удаление из таблицы связанного данного");

else

{

objFHeap.DeleteLexicalUnit(cardALexicalCode);

arrFHashTable[VHashIndex] = 0;

intFItemReserve--;

TableReHashing();

}

else

{

objFHeap.DeleteLexicalUnit(cardALexicalCode);

arrFHashTable[VHashIndex] = 0;

intFItemReserve--;

TableReHashing();

}

}

public void Save(ref StreamWriter fl)

{

try

{

fl.WriteLine(cardPTableSize.ToString());

fl.WriteLine(intFItemReserve.ToString());

for (int i = 1; i < cardPTableSize; i++)

fl.Write("\t" + arrFHashTable[i].ToString());

fl.Write("\n");

boolIsSaved = true;

}

catch (InvalidCastException)

{ boolIsSaved = false; }

}

public void GetLexicalUnitList(ref List<string> sList)

{

for (int i = 0; i < arrFHashTable.Count; i++) if (arrFHashTable[i] != 0) sList.Add(objFHeap.arrFHeapTable[arrFHashTable[i]].strFLexicalUnit);

}

}

}

**HashTablesList.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

using System.Windows.Forms;

namespace nsHashTables

{

public class CHashTableList

{

private List<THashTable> arrFHashTableList = new List<THashTable>();

private bool boolFIsSaved;

public bool boolFIsLoaded;

private byte byteFTablesSize;

static THeap objFHeap = new THeap();

//------------------------------------------------------------------------------

public CHashTableList(byte byteATableCount)

{

this.byteFTablesSize = byteATableCount;

objFHeap = new THeap();

Resize(arrFHashTableList, byteATableCount);

}

//------------------------------------------------------------------------------

public CHashTableList(string strAFileName)

{

try

{

boolFIsLoaded = Load(strAFileName);

}

catch (InvalidCastException)

{

MessageBox.Show("Ошибка при восстановлении хеш-таблиц из файла !");

boolFIsLoaded = false;

}

}

//------------------------------------------------------------------------------

public byte GetTableNumber(int intALexicalCode) { return objFHeap.arrFHeapTable[intALexicalCode].byteFHashTable; }

//------------------------------------------------------------------------------

public int GetTablesCount()

{

return arrFHashTableList.Count();

}

//------------------------------------------------------------------------------

static void Resize(List<THashTable> list, int size)

{

if (size > list.Count)

while (size > list.Count)

{

list.Add(new THashTable(ref objFHeap));

}

else if (size < list.Count)

while (list.Count - size > 0)

list.RemoveAt(list.Count - 1);

}

//------------------------------------------------------------------------------

static void Resize(List<object> list, int intANewSize)

{

if (intANewSize > list.Count)

while (intANewSize > list.Count)

list.Add(new object());

else if (intANewSize < list.Count)

while (list.Count - intANewSize > 0)

list.RemoveAt(list.Count - 1);

}

//------------------------------------------------------------------------------

static void Resize(List<int> list, int intANewSize)

{

if (intANewSize > list.Count)

while (intANewSize > list.Count)

list.Add(new Int32());

else if (intANewSize < list.Count)

while (list.Count - intANewSize > 0)

list.RemoveAt(list.Count - 1);

}

//------------------------------------------------------------------------------

public object GetUserData(int intALexicalCode)

{

if ((0 < intALexicalCode) && (intALexicalCode < objFHeap.intPFreeItem))

return arrFHashTableList[GetTableNumber(intALexicalCode)].arrFUserTable[intALexicalCode];

else

{

MessageBox.Show("Неверно задан лексический код при чтении пользовательских данных");

return null;

}

}

//------------------------------------------------------------------------------

public void SetUserData(int intALexicalCode, object objAUserData)

{

if ((0 < intALexicalCode) && (intALexicalCode < objFHeap.intPFreeItem))

{

if (arrFHashTableList[GetTableNumber(intALexicalCode)].arrFUserTable.Count > 0)

arrFHashTableList[GetTableNumber(intALexicalCode)].arrFUserTable[intALexicalCode] = objAUserData;

else

MessageBox.Show("Попытка записи адреса в несозданный массив пользовательских данных!");

}

else MessageBox.Show("Неверно задан лексический код при записи пользовательских данных!");

}

//------------------------------------------------------------------------------

public string GetLexicalUnit(int intALexicalCode)

{

if ((0 < intALexicalCode) && (intALexicalCode < objFHeap.intPFreeItem)) return objFHeap.arrFHeapTable[intALexicalCode].strFLexicalUnit;

else

{

MessageBox.Show("Неверно задан лексический код при чтении пользовательских данных!");

return "";

}

}

//------------------------------------------------------------------------------

public bool SearchLexicalUnit(string strALexicalUnit, byte byteATable, ref int intALexicalCode)

{

return arrFHashTableList[byteATable].SearchLexicalUnit(strALexicalUnit, ref intALexicalCode);

}

//------------------------------------------------------------------------------

public bool AddLexicalUnit(string strALexicalUnit, byte byteATable, ref int intALexicalCode)

{

if (byteATable >= arrFHashTableList.Count)

{

if (MessageBox.Show("Увеличить количество таблиц?", "Запрашиваемый индекс таблицы не существует.", MessageBoxButtons.YesNo) == DialogResult.Yes)

Resize(arrFHashTableList, byteATable + 1);

else

return false;

}

return arrFHashTableList[byteATable].AddLexicalUnit(strALexicalUnit, byteATable, ref intALexicalCode);

}

//------------------------------------------------------------------------------

public void DeleteLexicalUnit(string strALexicalUnit, byte byteATable)

{

arrFHashTableList[byteATable].DeleteLexicalUnit(strALexicalUnit);

}

//------------------------------------------------------------------------------

public void DeleteLexicalCode(int intALexicalCode)

{

short T = objFHeap.arrFHeapTable[intALexicalCode].byteFHashTable;

arrFHashTableList[T].DeleteLexicalCode(intALexicalCode);

}

//------------------------------------------------------------------------------

public void SetUserTable(byte byteATable)

{

arrFHashTableList[byteATable].SetUserTable();

}

//------------------------------------------------------------------------------

public void Expantion()

{

Resize(arrFHashTableList, ++byteFTablesSize);

}

//------------------------------------------------------------------------------

public void Save(string strAFileName)

{

try

{

StreamWriter fl = File.CreateText(strAFileName);

fl.WriteLine(byteFTablesSize.ToString());

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

fl.Write(arrFHashTableList[i].arrFHashTable.Count.ToString() + "\t");

fl.WriteLine("");

objFHeap.Save(ref fl);

boolFIsSaved = true;

fl.Close();

}

catch (InvalidDataException)

{ boolFIsSaved = false; }

}

//------------------------------------------------------------------------------

public bool Load(string strAFileName)

{

boolFIsLoaded = false;

try

{

StreamReader sr = new StreamReader(strAFileName);

byteFTablesSize = Convert.ToByte(sr.ReadLine());

if (byteFTablesSize < 1 || byteFTablesSize > 16)

{

MessageBox.Show("Unbelivable count of tables: " + byteFTablesSize.ToString());

return boolFIsLoaded;

}

arrFHashTableList.Clear();

Resize(arrFHashTableList, byteFTablesSize/\*+1\*/);

string line = sr.ReadLine();

char[] delim = { '\t'/\*,'\n'\*/ };

string[] counts = line.Split(delim);

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

{

arrFHashTableList[i].Init(Convert.ToInt32(counts[i]));

}

objFHeap.Load(ref sr);

sr.Close();

int n = objFHeap.arrFHeapTable.Count;

for (int i = 1; i < n; ++i)

{

THeapItem Item = objFHeap.arrFHeapTable[i];

if (Item.strFLexicalUnit.Length == 0)

break;

arrFHashTableList[Item.byteFHashTable].arrFHashTable[Item.intFHashIndex] = i;

}

boolFIsLoaded = true;

}

catch (InvalidDataException)

{

MessageBox.Show("Ошибка при восстановлении из файла хеш-таблиц!"); boolFIsLoaded = false;

}

return boolFIsLoaded;

}

//------------------------------------------------------------------------------

// отладка

public void HeapTableView(List<string> sList)

{

objFHeap.HeapTableView(sList);

}

//------------------------------------------------------------------------------

public void TableToStringList(byte byteATable, List<string> sList)

{

arrFHashTableList[byteATable].GetLexicalUnitList(ref sList);

}

//------------------------------------------------------------------------------

public int GetHashIndex(byte Table)

{

return arrFHashTableList[Table].intFHashIndex;

}

//------------------------------------------------------------------------------

}

}

**THeap.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

using System.Windows.Forms;

namespace nsHashTables

{

public struct THeapItem

{

public string strFLexicalUnit;

public byte byteFHashTable;

public int intFHashIndex;

public THeapItem(string strALexicalUnit, byte byteATable, int intAHashIndex)

{

strFLexicalUnit = strALexicalUnit;

byteFHashTable = byteATable;

intFHashIndex = intAHashIndex;

}

}

public class THeap

{

public List<THeapItem> arrFHeapTable = new List<THeapItem>();

private List<int> arrFDeleted = new List<int>();

private int intFFreeItem;

bool boolIsSaved;

bool boolIsLoaded;

public bool boolPIsSaved { get { return boolIsSaved; } }

public bool boolPIsLoaded { get { return boolIsLoaded; } }

public int intPFreeItem { get { return intFFreeItem; } }

public THeap()

{

Init();

intFFreeItem = 1;

}

protected void Init()

{

arrFDeleted.Clear();

arrFHeapTable.Clear();

int cnt = 4;

Resize(arrFHeapTable, cnt);

}

static void Resize(List<THeapItem> list, int size)

{

if (size > list.Count)

while (size > list.Count)

list.Add(new THeapItem("", 0, 0));

else if (size < list.Count)

while (list.Count - size > 0)

list.RemoveAt(list.Count - 1);

}

static void Resize(List<int> list, int size)

{

if (size > list.Count)

while (size > list.Count)

list.Add(new Int32());

else if (size < list.Count)

while (list.Count - size > 0)

list.RemoveAt(list.Count - 1);

}

static void Resize(List<char> list, int size)

{

if (size > list.Count)

while (size > list.Count)

list.Add('0');

else if (size < list.Count)

while (list.Count - size > 0)

list.RemoveAt(list.Count - 1);

}

public void Expansion()

{

int cardVSize = arrFHeapTable.Count;

cardVSize = cardVSize + cardVSize % 10 + 1;

Resize(arrFHeapTable, cardVSize);

Resize(arrFHeapTable, cardVSize);

}

public void AddLexicalUnit(string strALexicalUnit, byte byteAHashTable, int cardAHashIndex, ref int cardALexicalCode)

{

int intVIndex;

if (arrFDeleted.Count == 0)

{

intVIndex = intFFreeItem;

intFFreeItem++;

if (intFFreeItem >= (Int32)(arrFHeapTable.Count \* 0.9))

Expansion();

}

else

{

intVIndex = arrFDeleted[arrFDeleted.Count - 1];

Resize(arrFDeleted, arrFDeleted.Count - 1);

}

THeapItem Item = arrFHeapTable[intVIndex];

Item.strFLexicalUnit = strALexicalUnit;

Item.byteFHashTable = byteAHashTable;

Item.intFHashIndex = cardAHashIndex;

arrFHeapTable[intVIndex] = Item;

cardALexicalCode = intVIndex;

}

public void DeleteLexicalUnit(int cardALexicalCode)

{

int i;

if (arrFDeleted == null || !arrFDeleted.Any())

i = 0;

else i = arrFDeleted.Count();

Resize(arrFDeleted, i + 1);

arrFDeleted[i] = cardALexicalCode;

THeapItem Item = arrFHeapTable[cardALexicalCode];

Item.strFLexicalUnit = "";

Item.byteFHashTable = 0;

Item.intFHashIndex = 0;

}

public void Save(ref StreamWriter sw)

{

try

{

for (int i = 1; i < arrFHeapTable.Count; i++) //type?

{

if (arrFHeapTable[i].strFLexicalUnit == "")

break;

sw.Write(arrFHeapTable[i].strFLexicalUnit + "\t");

sw.Write(arrFHeapTable[i].byteFHashTable.ToString() + "\t");

sw.WriteLine(arrFHeapTable[i].intFHashIndex.ToString());

}

boolIsSaved = true;

}

catch (Exception) { boolIsSaved = false; }

}

public void Load(ref StreamReader sr)

{

try

{

Init();

int size = arrFHeapTable.Count;

int readSz = 0;

while (true)

{

string line = sr.ReadLine();

if (line == null)

break;

if (++readSz >= size)

{

size \*= 2;

Resize(arrFHeapTable, size);

}

char[] delim = { '\t'/\*,'\n'\*/ };

string[] lines = line.Split(delim);

THeapItem it = arrFHeapTable[readSz];

it.strFLexicalUnit = lines[0];

it.byteFHashTable = Convert.ToByte(lines[1]);

it.intFHashIndex = Convert.ToInt32(lines[2]);

arrFHeapTable[readSz] = it;

}

intFFreeItem = readSz + 1;

boolIsLoaded = true;

}

catch (InvalidCastException)

{ boolIsLoaded = false; }

}

THeapItem GetItem(int i)

{

if (i >= arrFHeapTable.Count)

{

MessageBox.Show("GetИндекс кучи вышел за диапазон!");

THeapItem Item = new THeapItem("", 0, 0);

return Item;

}

else return arrFHeapTable[i];

}

void SetItem(int i, THeapItem NewItem)

{

if (i >= arrFHeapTable.Count)

MessageBox.Show("SetИндекс кучи вышел за диапазон!");

else arrFHeapTable[i] = NewItem;

}

public void HeapTableView(List<string> sList)

{

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

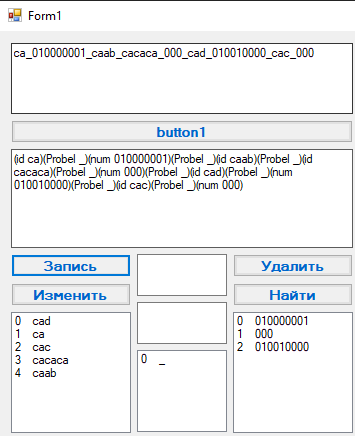
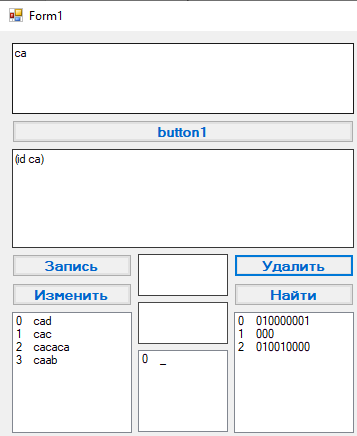
sList.Add(arrFHeapTable[i].strFLexicalUnit);

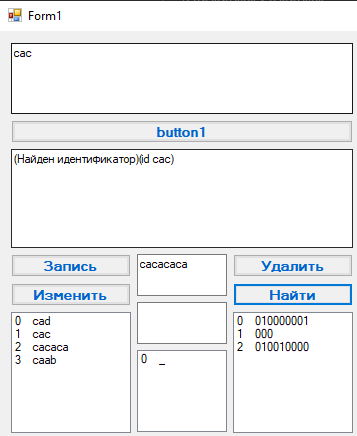
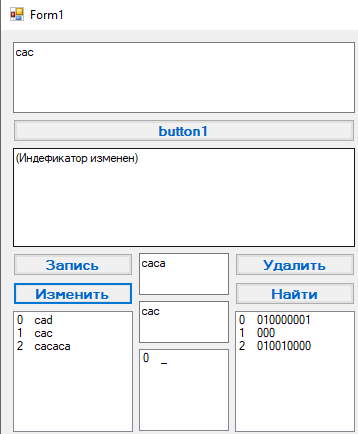
}

}

}

**Результаты:**

# **Лабораторная работа № 5. Построение синтаксического дерева**

Построение синтаксического дерева — это процесс анализа предложения или фразы с целью выявления структуры и связей между словами в соответствии с их синтаксическими ролями. Синтаксическое дерево представляет собой графическое представление синтаксической структуры предложения, где узлы представляют слова, а ребра - синтаксические связи между ними.

Процесс построения синтаксического дерева обычно включает в себя несколько этапов:

1. Лексический анализ (токенизация):

• Начните с лексического анализа входного текста или кода, который разбивает текст на лексемы (токены). Лексемы представляют собой минимальные логические единицы, такие как идентификаторы, ключевые слова, операторы и числа.

2. Синтаксический анализ:

• Затем применяется синтаксический анализатор для создания синтаксического дерева на основе лексем. Синтаксический анализатор использует контекстно-свободную грамматику, определенную для языка, чтобы определить структуру программы и создать соответствующее синтаксическое дерево.

• Синтаксический анализатор обрабатывает грамматические правила, которые определены в грамматике языка, и строит дерево, в котором узлы представляют синтаксические конструкции, а дуги (рёбра) указывают на связи между этими конструкциями.

3. Построение синтаксического дерева:

• В процессе синтаксического анализа начинается построение синтаксического дерева.

• Корень дерева обычно представляет программу в целом, а каждый узел дерева представляет собой определенную синтаксическую конструкцию, такую как выражение, оператор, условие и т. д.

• Для бинарных операторов, таких как арифметические операции или операции сравнения, левый и правый операнды представлены дочерними узлами этого оператора.

На основе результатов синтаксического анализа строится синтаксическое дерево, где каждый узел представляет слово, а ребра отображают синтаксические отношения между словами, такие как подчинение, определение и т.д.

Синтаксическое дерево позволяет визуально представить структуру предложения и отношения между его компонентами. Оно может быть использовано для анализа текста, машинного перевода, автоматического извлечения информации и других лингвистических задач

4. Упрощение дерева (по необходимости):

• В некоторых случаях синтаксическое дерево может быть упрощено или оптимизировано. Например, можно удалить узлы, представляющие ненужные промежуточные вычисления.

5. Использование синтаксического дерева:

• После построения синтаксического дерева, оно может использоваться для различных целей, таких как выполнение статического анализа, генерация промежуточного кода, компиляция, интерпретация и т. д.

• Синтаксическое дерево может быть также основой для построения абстрактных семантических деревьев (ASD), которые представляют собой более высокоуровневое представление смысла программы.

Элементы структурированного представления TreeView:

1. **Узлы:** Каждый узел представляет собой элемент в дереве и может содержать данные и/или другие дочерние узлы.
2. **Дерево:** Совокупность всех узлов и их отношений между собой формирует дерево, которое отображается в виде иерархической структуры.
3. **Разделители и иконки:** TreeView обычно использует разделители для отображения уровней вложенности и иконки для обозначения типа узлов (например, папки или файлы).

Структурированный TreeView может использоваться для представления различных типов данных, таких как файловая система, структуры документов, иерархии каталогов и многое другое. Пользователи могут разворачивать и сворачивать ветви дерева для просмотра и управления данными.

TreeView представляет визуальный элемент в виде дерева. Дерево содержит узлы, которые представляют объекты TreeNode. Узлы могут содержать другие подузлы и могут находиться как скрытом, так и в раскрытом состоянии. Все узлы содержатся в свойстве Nodes.

Внутреннее свойство Nodes, доступное в элементе TreeView возвращает коллекцию узлов дерева, и по умолчанию она пуста. Как видите, чтобы добавить новый элемент, достаточно использовать метод Add(string text) для данного свойства.

Метод Add(string text) возвращает добавленный узел в виде объекта класса TreeNode и к нему точно так же можно обратиться для добавления его дочерних узлов - через свойство Nodes и тот же метод Add(string text).

**Задание**. Включить в синтаксический анализатор из лабораторной работы №.3 построение синтаксического дерева. Использовать атрибутный метод Кнута, т.е. преобразовать КС – грамматику из лабораторной работы № 3 в атрибутную грамматику добавлением атрибутов и правил построения синтаксического дерева. Расширить программу синтаксического анализатора из лабораторной работы № 3 введением действий по построению синтаксического дерева.

**Код программы:**

uSyntAnalyzer.cs

using System;

using System.Collections.Generic;

using System.Text;

using nsLex;

using System.Windows.Forms;

using nsHashTables;

namespace nsSynt

{

class uSyntAnalyzer

{

const int word = 0;

const int number = 0;

const int reserved = 0;

public TreeNode[] treeNodes = new TreeNode[1024];

int i = 1;

int j = 1;

private String[] strFSource;

private String[] strFMessage;

public String[] strPSource { set { strFSource = value; } get { return strFSource; } }

public String[] strPMessage { set { strFMessage = value; } get { return strFMessage; } }

public CLex Lex = new CLex();

public CHashTableList htl = new CHashTableList(3);

int Count = 1;

int intVLexicalCode = 0;

public uSyntAnalyzer()

{

treeNodes[0] = new TreeNode("S");

}

public void S()

{

if (i > 0)

{

treeNodes[i] = new TreeNode("S");

treeNodes[i - 1].Nodes.Add(treeNodes[i]);

}

i++;

if (j == 1)

{

treeNodes[i] = new TreeNode("ca");

treeNodes[i - 1].Nodes.Add(treeNodes[i]);

j--;

}

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber)

{

treeNodes[i] = new TreeNode(Lex.strPLexicalUnit);

treeNodes[i - 1].Nodes.Add(treeNodes[i]);

A();

i--;

Lex.NextToken();

}

else if (Lex.enumPToken == TToken.lxmIdentifier)

{

treeNodes[i] = new TreeNode(Lex.strPLexicalUnit);

treeNodes[i - 1].Nodes.Add(treeNodes[i]);

i--;

S();

Lex.NextToken();

}

}

}

public void A()

{

if (i > 0)

{

treeNodes[i] = new TreeNode("A");

treeNodes[i - 1].Nodes.Add(treeNodes[i]);

}

i++;

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmUl)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmIdentifier)

{

treeNodes[i] = new TreeNode(Lex.strPLexicalUnit);

treeNodes[i - 1].Nodes.Add(treeNodes[i]);

S();

i--;

i--;

Lex.NextToken();

}

else if (Lex.enumPToken == TToken.lxmNumber)

{

treeNodes[i] = new TreeNode(Lex.strPLexicalUnit);

treeNodes[i - 1].Nodes.Add(treeNodes[i]);

i--;

A();

Lex.NextToken();

}

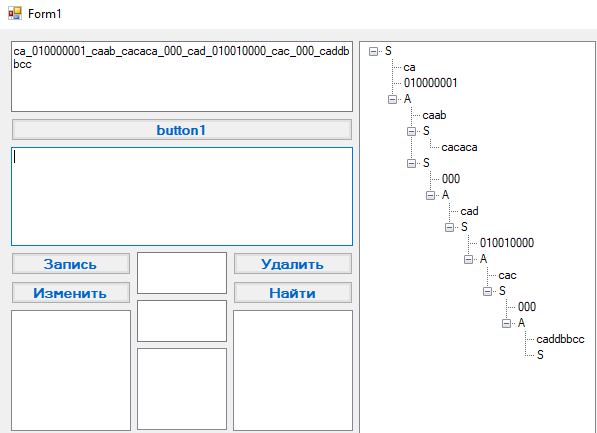
}

}

}

}

**Результат:**

****

# 

# **Лабораторная работа № 6. Разработка генератора**

Разработка генератора относится к созданию программного инструмента, который генерирует код или другой выходной результат на основе некоторых исходных данных или спецификаций. Генераторы могут использоваться для автоматизации и упрощения процессов разработки, а также для создания кода или документации.

На рис. 2 начальная стадия представлена блоками «Анализ» и «Генератор промежуточного кода», заключительная стадия – блоком «Генератор кода». В идеальном случае детали исходного языка ограничены начальной стадией (анализатор), а детали целевой машины (исполнителя) – заключительной стадией компилятора (генератор кода).

Генератор промежуточного кода выполняет перевод данных и операторов языка высокого уровня в некоторое абстрактное, независимое от архитектуры конкретного компьютера представление.

Генератор кода выполняет перевод с промежуточного кода в исполняемый код с учетом особенностей реализации компьютера.

Генерация кода состоит из ряда специфических, относительно независимых подзадач: распределение памяти (оперативной и регистровой), подбор команд компьютера, генерация объектного (или загрузочного) модуля. Конечно, независимость этих подзадач относительна: например, при выборе команд нельзя не учитывать схему распределения памяти, или, наоборот, схема распределения памяти необходимо ведет к генерации той или иной последовательности команд. Однако удобно и практично эти задачи все же разделять и при этом особо обращать внимание на их взаимодействие. В значительной мере схема генератора кода зависит от формы промежуточного представления исходной программы. Ясно, что генерация кода из вычислительной модели в виде дерева отличается от генерации кода из троек, а генерация кода из постфиксной записи отличается от генерации кода из модели ориентированного графа. В то же время, все генераторы кода имеют и много общего, и основные применяемые алгоритмы отличаются, как правило, только в деталях, связанных с используемым промежуточным представлением.

Атрибутивная грамматика — это тип формальной грамматики, который расширяет контекстно-свободные грамматики (КС-грамматики) добавлением атрибутов к нетерминалам и правилам грамматики. Атрибуты представляют собой значения, связанные с символами грамматики, и они могут быть вычислены с использованием семантических правил.

Основные компоненты атрибутивной грамматики:

1. Терминалы и нетерминалы:

Как и в контекстно-свободных грамматиках, атрибутивная грамматика определяет множество терминалов и нетерминалов.

1. Правила продукции:

Правила, определяющие, как нетерминалы могут быть заменены на последовательности терминалов и/или нетерминалов.

1. Семантические правила (атрибуты):

Семантические правила определяют, как вычисляются значения атрибутов. Эти правила могут быть связаны с правилами продукции и определяться для нетерминалов.

1. Интерпретатор:

Атрибутивная грамматика часто используется в компиляторах и интерпретаторах для определения того, как преобразовать входной код или программу.

Задание.

1. Перевести все числа в десятичное представление.

2. Выполнить вывод исходного текста в структурированном виде.

TreeView - это виджет (компонент пользовательского интерфейса) в графических пользовательских интерфейсах (GUI), который используется для отображения и управления иерархическими данными в виде древовидной структуры. В структурированном представлении TreeView узлы дерева могут быть организованы и иерархически связаны между собой.

Generator.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Text.RegularExpressions;

namespace mazitov

{

class Generator

{

List<string> word = new List<string>();

public List<string> ViewTree(TreeView tree, bool edit = true)

// Этот метод проходит через все узлы дерева и добавляет их содержимое в список.

// Если параметр edit равен true, он также преобразует текст каждого узла в десятичное число.

{

List<string> treeContent = new List<string>();

foreach (TreeNode node in tree.Nodes)

{

treeContent.AddRange(ViewNode(node, edit));

}

return treeContent;

}

public List<string> ViewNode(TreeNode node, bool edit)

{

List<string> nodeContent = new List<string>();

foreach (TreeNode child in node.Nodes)

{

nodeContent.AddRange(ViewNode(child, edit));

}

if (edit)

{

int value = -1;

if (int.TryParse(node.Text, out value))

{

int newValue = ConvertTo10(value);

node.Text = newValue.ToString();

}

}

nodeContent.Add(node.Text);

return nodeContent;

}

public int ConvertTo10(int num)

{

int l = 0;

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

{

int n = num.ToString().Length - i;

int s = num.ToString()[i] == '1' ? 1 : 0;

l += (int)Math.Pow(2, n - 1) \* s;

}

return l;

}

public TreeView currentTree;

public int d = 0;

public void Restruct(TreeView box, TreeView tree)

{

currentTree = tree;

tree.Nodes.Clear();

TreeNode sNode = null;

foreach (TreeNode node in box.Nodes)

{

if (node == null)

{

continue;

}

TreeNode newNode = new TreeNode(node.Text);

ReNode(null, node, ref newNode);

if (newNode.Text == "S")

{

sNode = newNode;

TreeNode nNode = new TreeNode("S");

sNode.Nodes.Add(nNode);

sNode = nNode;

}

tree.Nodes.Add(newNode);

}

if (sNode != null)

{

List<TreeNode> remove = new List<TreeNode>();

foreach (TreeNode node in tree.Nodes)

{

if (node != null && node.Text != "S" && sNode != null)

{

remove.Add(node);

}

}

TreeNode neNode = new TreeNode("cacaca");

if (sNode.Nodes != null)

{

sNode.Nodes.Add(neNode);

}

foreach (TreeNode node in remove)

{

if (node != null && node.Text != "A")

{

tree.Nodes.Remove(node);

if (sNode.Nodes != null)

{

sNode.Nodes.Add(node);

}

}

else

{

tree.Nodes.Remove(node);

if (neNode.Nodes != null)

{

node.Remove();

neNode.Nodes.Add(node);

}

}

}

}

tree.ExpandAll();

}

public void ReNode(TreeNode parent, TreeNode node, ref TreeNode newNode)

{

bool skipToRoot = false;

if (node.Nodes != null)

{

foreach (TreeNode child in node.Nodes)

{

TreeNode newn = new TreeNode(child.Text);

if (newn.Nodes != null)

{

int value = -1;

if (int.TryParse(newn.Text, out value))

{

int newValue = ConvertTo10(value);

newn.Text = newValue.ToString();

}

Console.WriteLine(child.Text);

if (Regex.IsMatch(child.Text, @"^[a-z]+$"))

{

if (word.Contains(Convert.ToString(child.Text)))

MessageBox.Show("В тексте не должно быть повторных индетификаторов!");

else

{

word.Add(child.Text);

}

}

ReNode(newNode, child, ref newn);

}

}

}

if (parent != null)

{

if (currentTree != null && skipToRoot)

{

currentTree.Nodes.Add(newNode);

}

else

{

parent.Nodes.Add(newNode);

}

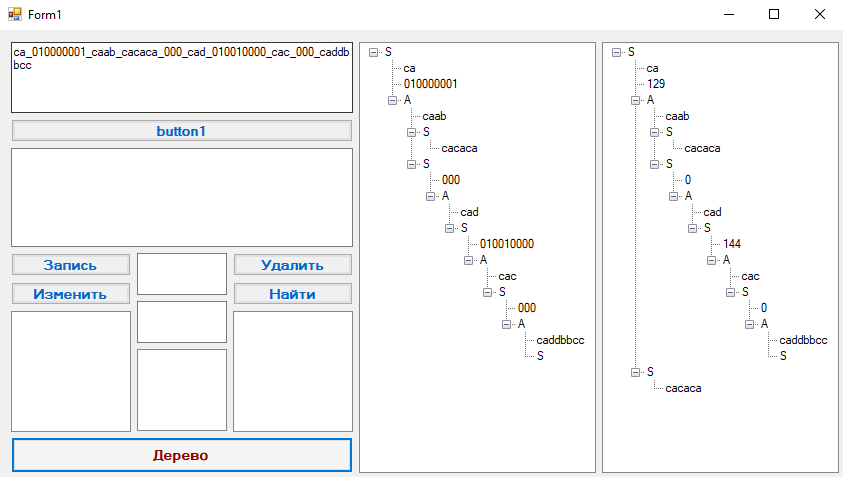
}

}

}

}

**Результат:**



# **Лабораторная работа № 7. Разработка контекстного анализатора**

Контекстный анализатор (иногда также называемый семантическим анализатором) является важным компонентом в процессе компиляции и анализа программных кодов. Его основная задача - проводить анализ программы после синтаксического анализа и проверять семантические правила языка.

Этапы разработки контекстного анализатора:

1. Определение целей и задач контекстного анализатора: определение области применения, целевой аудитории, основных задач и требований к функциональности.

2. Сбор и анализ данных: сбор необходимых данных для обучения анализатора, анализ существующих данных и их подготовка для обучения.

3. Выбор методов анализа: выбор подходящих методов машинного обучения, естественного языка и других технологий для реализации функциональности анализатора.

4. Разработка алгоритмов и моделей: разработка и обучение моделей машинного обучения, алгоритмов анализа текста и других компонентов анализатора.

5. Тестирование и отладка: тестирование разработанных моделей и алгоритмов на различных наборах данных, отладка и оптимизация функциональности.

6. Интеграция в приложение: интеграция разработанного контекстного анализатора в целевое приложение или систему.

7. Тестирование и оптимизация: проведение тестирования в реальных условиях, оптимизация работы анализатора и улучшение его точности и производительности.

8. Релиз и поддержка: выпуск готового продукта, поддержка и обновление контекстного анализатора в соответствии с потребностями пользователей и изменениями в окружающей среде.

Контекстуальность в языкознании представляет собой условие осмысленного употребления той или иной конкретной языковой единицы в речи (письменной или устной), с учётом её языкового окружения и ситуации речевого общения.

Задачей контекстного анализа языков программирования является установление правильности совместного использования синтаксических единиц в тексте исходной программы.

В языках программирования контекст – это текстовое окружение некоторой фразы языка программирования, некоторого фрагмента текста. Правила согласованного использования отдельных фрагментов текста программы называются контекстными условиями.

Для обнаружения контекстных ошибок предназначен анализатор контекстных зависимостей.

Генератор — итератор, реализованный в виде сопрограммы. Генератор документации — программа автоматического извлечения документации из исходных кодов или исполнимых модулей.

**Задание.** Для предложенного преподавателем варианта контекстного условия расширить атрибутную грамматику из лабораторной работы № 4 добавлением атрибутов, правил их вычисления, правил вычисления контекстных условий. Включить в программу синтаксического анализатора из лабораторной работы № 4 действия по вычислению атрибутов и проверки контекстных условий.

|  |
| --- |
| Все идентификаторы должны быть разными |

Generator.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Text.RegularExpressions;

namespace LabRab1

{

class Generator

{

List<string> word = new List<string>();

public List<string> ViewTree(TreeView tree, bool edit = true)

// Этот метод проходит через все узлы дерева и добавляет их содержимое в список.

// Если параметр edit равен true, он также преобразует текст каждого узла в десятичное число.

{

List<string> treeContent = new List<string>();

foreach (TreeNode node in tree.Nodes)

{

treeContent.AddRange(ViewNode(node, edit));

}

return treeContent;

}

public List<string> ViewNode(TreeNode node, bool edit)

// Этот метод рекурсивно проходит через все дочерние узлы данного узла и добавляет их содержимое в список.

// Если параметр edit равен true, он также преобразует текст каждого узла в десятичное число.

{

List<string> nodeContent = new List<string>();

foreach (TreeNode child in node.Nodes)

{

nodeContent.AddRange(ViewNode(child, edit));

}

if (edit)

{

int value = -1;

if (int.TryParse(node.Text, out value))

{

int newValue = ConvertTo10(value);

node.Text = newValue.ToString();

}

}

nodeContent.Add(node.Text);

return nodeContent;

}

public int ConvertTo10(int num) // Этот метод преобразует двоичное число в десятичное.

{

int l = 0;

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

{

int n = num.ToString().Length - i;

int s = num.ToString()[i] == '1' ? 1 : 0;

l += (int)Math.Pow(2, n - 1) \* s;

}

return l;

}

public TreeView currentTree;

public int d = 0;

public void Restruct(TreeView box, TreeView tree)

{

currentTree = tree;

tree.Nodes.Clear();

TreeNode sNode = null;

foreach (TreeNode node in box.Nodes)

{

if (node == null)

{

continue;

}

TreeNode newNode = new TreeNode(node.Text);

// MessageBox.Show(node.Text);

ReNode(null, node, ref newNode);

if (newNode.Text == "S")

{

sNode = newNode;

TreeNode nNode = new TreeNode("");

sNode.Nodes.Add(nNode);

sNode = nNode;

}

tree.Nodes.Add(newNode);

}

if (sNode != null)

{

List<TreeNode> remove = new List<TreeNode>();

foreach (TreeNode node in tree.Nodes)

{

if (node != null && node.Text != "S" && sNode != null)

{

remove.Add(node);

}

}

TreeNode neNode = new TreeNode("");

if (sNode.Nodes != null)

{

sNode.Nodes.Add(neNode);

}

foreach (TreeNode node in remove)

{

if (node != null && node.Text != "A")

{

tree.Nodes.Remove(node);

if (sNode.Nodes != null)

{

sNode.Nodes.Add(node);

}

}

else

{

tree.Nodes.Remove(node);

if (neNode.Nodes != null)

{

neNode.Nodes.Add(node);

}

}

}

}

tree.ExpandAll();

}

public void ReNode(TreeNode parent, TreeNode node, ref TreeNode newNode)

{

bool skipToRoot = false;

// Проверка на null, чтобы избежать NullReferenceException

if (node.Nodes != null)

{

foreach (TreeNode child in node.Nodes)

{

TreeNode newn = new TreeNode(child.Text);

// Проверка на null, чтобы избежать NullReferenceException

if (newn.Nodes != null)

{

int value = -1;

if (int.TryParse(newn.Text, out value))

{

int newValue = ConvertTo10(value);

newn.Text = newValue.ToString();

}

Console.WriteLine(child.Text);

if (Regex.IsMatch(child.Text, @"^[a-z]+$"))

{

if (word.Contains(Convert.ToString(child.Text)))

MessageBox.Show("В тексте не должно быть повторных индетификаторов");

else

{

word.Add(child.Text);

}

}

ReNode(newNode, child, ref newn);

}

}

}

if (parent != null)

{

if (currentTree != null && skipToRoot)

{

currentTree.Nodes.Add(newNode);

}

else

{

parent.Nodes.Add(newNode);

}

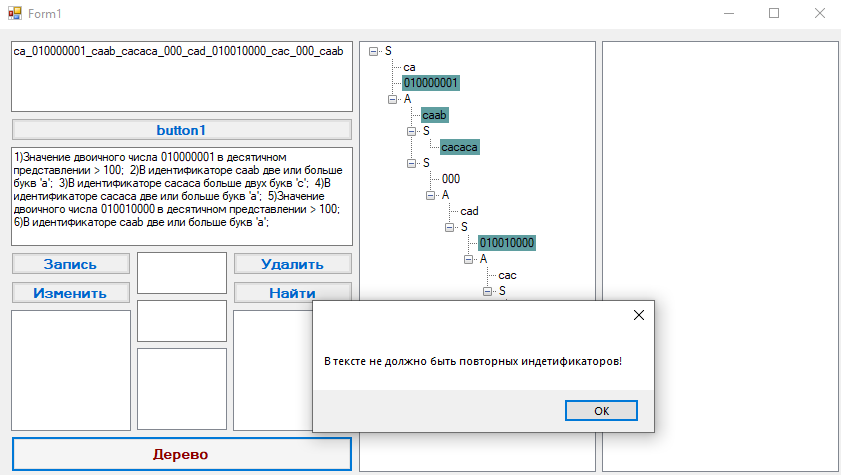
}

}

}

}

**Результат (caab):**



# **Лабораторная работа № 8. Разработка семантического анализатора**

Семантика – раздел лингвистики, изучающий смысловое значение единиц языка. Процесс человеческого мышления, как и язык, который представляет собой инструмент выражения мыслей, является очень гибким и трудно поддается формализации. Поэтому семантический анализ по праву считается самым сложным этапом автоматической обработки текстов.

Создание новых методов семантического анализа текстов откроет новые возможности и позволит существенно продвинуться в решении многих задач компьютерной лингвистики, таких как машинный перевод, автореферирование, классификация текстов и других. Не менее актуальна разработка новых инструментов, позволяющих автоматизировать семантический анализ.

Важным аспектом семантического анализа является проверка типов, когда компилятор проверяет, что каждый оператор имеет операнды допустимого спецификациями языка типа. Например, определение многих языков программирования требует, чтобы при использовании действительного числа в качестве индекса массива генерировалось сообщение об ошибке. В то же время спецификация языка может позволить определенное преобразование типов, например, когда бинарный арифметический оператор применяется к операндам целого и действительного типов. В этом случае компилятору может потребоваться преобразование целого числа в действительное.

Все типы известны во время компиляции, и это относится к типам выражений, идентификаторам и литералам. При этом неважно, насколько сложным является выражение: его тип может определяться во время компиляции за определенное количество шагов исходя из типов его составляющих. Фактически, это позволяет производить контроль типов во время компиляции и находить в процессе компиляции многие программные ошибки.

В процессе компиляции анализатору требуются таблицы:

– таблица символов;

– таблица типов;

– таблица функций;

– таблица меток.

Основная задача таблицы символов – установление соответствия между переменной и ее типом. Операции, связанные с таблицей символов:

– операция, соответствующая определяющему вхождению переменной. В таблицу символов помещаются имя переменной и ее тип;

– операция, соответствующая применимому вхождению переменной. Исследуется таблица символов для нахождения типа переменной.

**Задание:** разработать семантический анализатор. Выполнить проверку внеконтекстной грамматики.

**Вариант:** проверка идентификатора слова <1> на наличие 2х или более букв ‘a’ и проверка идентификатора слова <1> на наличие более 2х букв ‘c’, и проверка того, что значение двоичного числа в десятичном представлении больше 100.

**Код:**

uSemantAnalyzer

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Drawing;

using System.Threading.Tasks;

using nsLex;

using nsSynt;

using nsHashTables;

using System.Windows.Forms;

namespace mazitov

{

internal class uSemantAnalyzer

{

private TreeView tree;

List<string> errors = new List<string>();

public uSemantAnalyzer()

{

}

public uSemantAnalyzer(TreeView treeView)

{

tree = treeView;

TreeController(tree);

}

public void TreeController(TreeView tree)

{

string error = "";

foreach (TreeNode node in tree.Nodes)

{

FindError(node, error);

}

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

{

error += $"{i + 1}){errors[i]}; ";

}

if (error != "")

{

throw new Exception(error);

}

}

public void FindError(TreeNode node, string error)

{

int numToInt;

foreach (TreeNode nd in node.Nodes)

{

if (int.TryParse(nd.Text, out numToInt))

{

numToInt = Convert.ToInt32(nd.Text, 2);

}

if (nd.Text.Count(i => i == 'c') > 2)

{

tree.SelectedNode = nd;

tree.SelectedNode.BackColor = Color.CadetBlue;

tree.SelectedNode.ForeColor = Color.Black;

errors.Add($"В идентификаторе {nd.Text} больше двух букв 'c'");

}

if (nd.Text.Count(i => i == 'a') >= 2)

{

tree.SelectedNode = nd;

tree.SelectedNode.BackColor = Color.CadetBlue;

tree.SelectedNode.ForeColor = Color.Black;

errors.Add($"В идентификаторе {nd.Text} две или больше букв 'a'");

}

else if (numToInt > 100)

{

tree.SelectedNode = nd;

tree.SelectedNode.BackColor = Color.CadetBlue;

tree.SelectedNode.ForeColor = Color.Black;

errors.Add($"Значение двоичного числа {nd.Text} в десятичном представлении > 100");

}

FindError(nd, error);

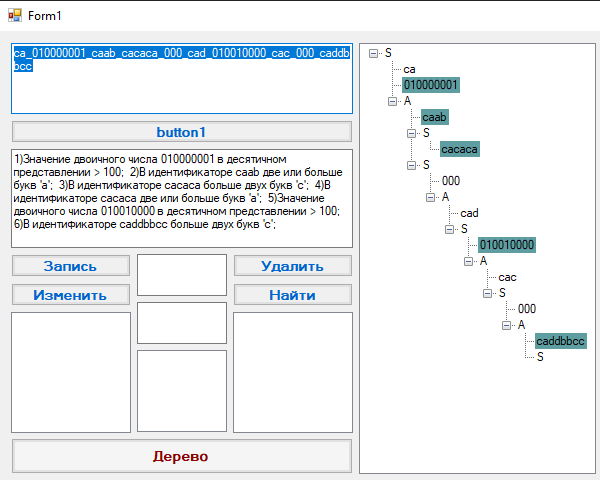
}

}

}

}

**Результат:**

****