КАЗАНСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

им. А.Н. Туполева - КАИ

Кафедра АСОИУ

Лабораторные работы №1-8

по дисциплине

«ТЕОРИЯ фОРМАЛЬНЫХ ГРАММАТИК И АВТОМАТОВ»

Выполнил:

Студент группы 4309

Хафизов Н. Р.

Проверила: Бикмуллина И.И.

Казань 2023

**Лабораторная работа № 1. Разработка транслитератора**

**Теоретическая часть**

Транслитерация – точная передача знаков одной письменности знаками другой письменности, при которой каждый знак (или последовательность знаков) одной системы письма передается одним и тем же знаком (или последовательностью знаков) другой системы письма. Например, романизация – передача нелатинской письменности средствами расширенного латинского алфавита с применением диакритических знаков и буквосочетании.

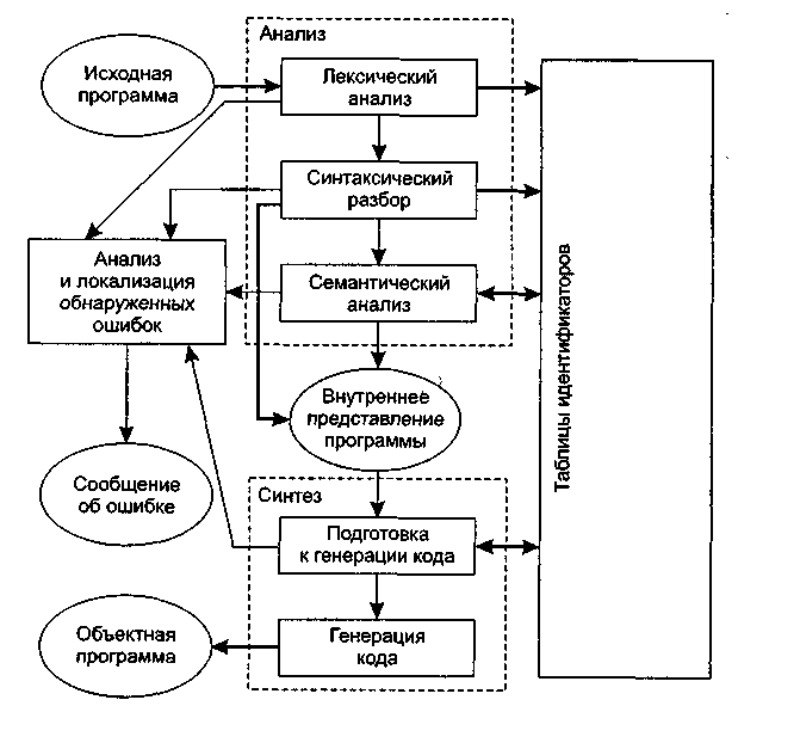


Рис. 1. Общая схема жизненного цикла транслитератоа

Грамматика, применяемая в лабораторной работе:

1. A,B,C,D – Letter
2. Пробел – Space
3. Точка с запятой – tochzap
4. Запятая – Zap
5. Левая и правая скобка – opbr, clbr
6. Восклицательный знак - Voskl

**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.Text;

using System.Windows.Forms;

using ТЯПЛР1;

namespace ТЯПЛР1

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

tbFSource.AppendText("(0;1,a!b)" + "\r\n");

tbFSource.AppendText("1 a");

int n = tbFSource.Lines.Length;

}

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

{

CLex Lex = new CLex();

Lex.strPSource = tbFSource.Lines;

Lex.strPMessage = tbFMessage.Lines;

int x = tbFSource.TextLength;

int y = tbFSource.Lines.Length;

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

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

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

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

case TCharType.EXP: { s1 = "voskl"; break; }

case TCharType.COM: { s1 = "zap"; break; }

case TCharType.TCHZP: { s1 = "tochzap"; 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;

}

}

}

}

**СLex.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace ТЯПЛР1

{

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

public enum TCharType { Letter, Digit, EndRow, EndText, Space, ReservedSymbol,OPbr,CLbr,EXP,COM,TCHZP }; // тип символа

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

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 == '(' ) enumFSelectionCharType = TCharType.OPbr;

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

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

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

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

//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();

}

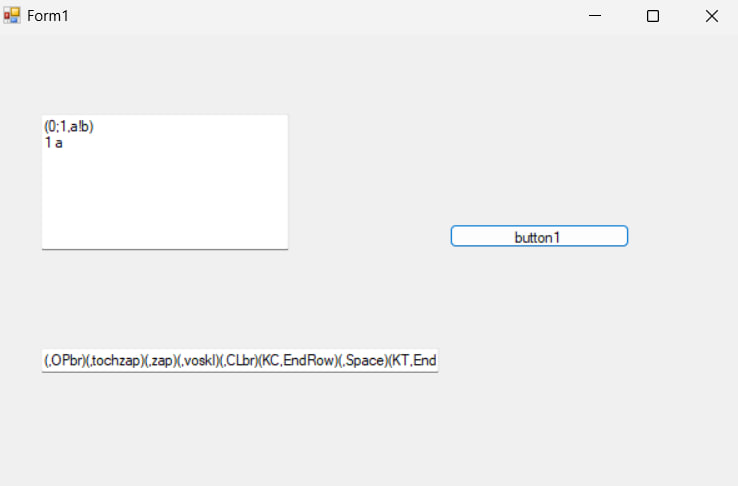
intFSourceColSelection++;

}

}

}

**Полученный результат:**



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

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

1. Основные функции лексического анализатора:
2. Чтение с помощью транслитератора очередного слова в исходном
3. тексте и его класси-фикация;
4. Пропуск пробелов и комментариев;
5. Выдача диагностических сообщений об обнаруженных
6. 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.

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. Давайте рассмотрим эти термины более подробно:

1. Терминальные (Terminal):

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.

2. Нетерминальные (Non-terminal):

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.

24 Вариант

|  |  |  |
| --- | --- | --- |
| (011)\*101(110)\* | (a|b|c|d)+ | Вторые два символа всегда ba |

**Первое слово:**

(000)\*101(110)\*

A → 0B | 1C

B → 1D

C → 0Е

D → 1А

E → 1 | 1F

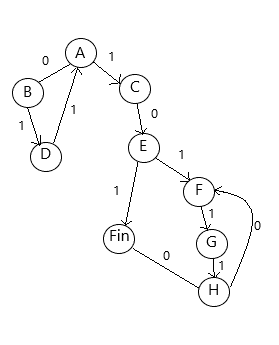
F → 1G

G → 1H

H → 0 | 0F

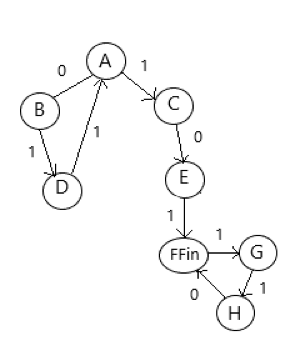
**Недетерминированная матрица:**

|  |  |  |
| --- | --- | --- |
|  | 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 | B | C |
| B |  | D |
| C | E |  |
| D |  | A |
| E |  | FFin |
| FFin |  | G |
| G |  | H |
| H | FFin |  |



**Второе слово:**

(a|b|c|d)+

Вторые два символа всегда ba

A → aB | bB | cB | dB

B → bС

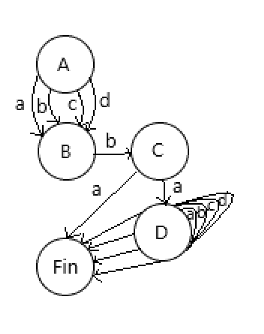
C → a | aD

D → a | b | c | d | aD | bD | cD | dD

**Недетерминированная матрица:**

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

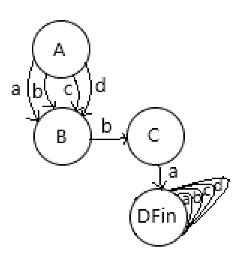
**Граф:**



**Детерминированная матрица:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | c | d |
| A | B | B | B | B |
| B |  | C |  |  |
| C | DFin |  |  |  |
| DFin | DFin | DFin | DFin | DFin |

**Граф:**



**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 KhayrullinKA\_4307

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

tbFSource.AppendText("000101110" + "\r\n");

tbFSource.AppendText("adbc");

int n = tbFMessage.Lines.Length;

}

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

{

}

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

{

CLex Lex = new CLex();

Lex.strPSource = tbFSource.Lines;

Lex.strPMessage = tbFMessage.Lines;

Lex.intPSourceColSelection = -1;

Lex.intPSourceRowSelection = 0;

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.LBracket: { s = "LBracket";break; }

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

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

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

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

case TCharType.Question: { s = "Question"; 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.Linq;

using System.Text;

using System.Threading.Tasks;

namespace KhayrullinKA\_4307

{

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

public enum TCharType { Letter, Digit, EndRow, EndText, Space, ReservedSymbol, LBracket, RBracket, Exclamation, Semicolon, Coma, Question }; // тип символа

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

public class CLex

{

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 == '(') enumFSelectionCharType = TCharType.LBracket;

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

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

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

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

else if (chrFSelection == '?') enumFSelectionCharType = TCharType.Question;

else if (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 24

switch (enumFSelectionCharType)

{

case TCharType.Letter:

{

// a b c d

// A | B | B | B | B |

// B | | C | | |

// C |DFin| | | |

// DFin |DFin|DFin|DFin|DFin|

A:

{

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

{

TakeSymbol();

goto B;

}

}

B:

{

if (chrFSelection == 'b')

{

TakeSymbol();

goto C;

}

else throw new Exception("Вторые два символа всегда ba");

}

C:

{

if (chrFSelection == 'a')

{

TakeSymbol();

goto DFin;

}

else throw new Exception("Вторые два символа всегда ba");

}

DFin:

{

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

{

TakeSymbol();

goto DFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

case TCharType.Digit:

{

// 0 1

// A | B | C |

// B | | D |

// C | E | |

// D | | A |

// E | |FFin |

// FFin | | G |

// G | | H |

// H |FFin | |

A:

if (chrFSelection == '0')

{

TakeSymbol();

goto B;

}

else if (chrFSelection == '1')

{

TakeSymbol();

goto C;

}

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

B:

if (chrFSelection == '1')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался 1");

C:

if (chrFSelection == '0')

{

TakeSymbol();

goto E;

}

else throw new Exception("Ожидался 0");

D:

if (chrFSelection == '1')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидался 1");

E:

if (chrFSelection == '1')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 1");

FFin:

if (chrFSelection == '1')

{

TakeSymbol();

goto G;

}

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

else throw new Exception("Ожидалась 1");

G:

if (chrFSelection == '1')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался 1");

H:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидался 0");

}

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;

}

break;

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

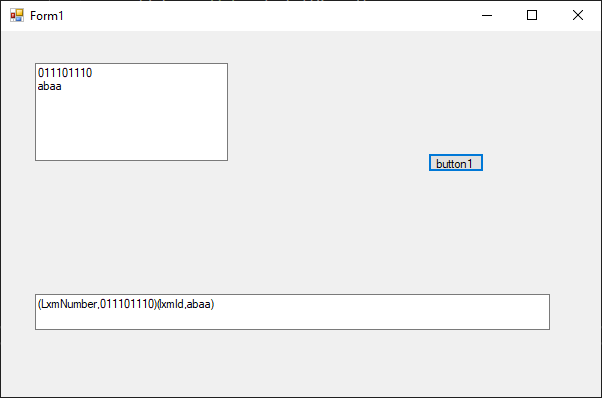
}

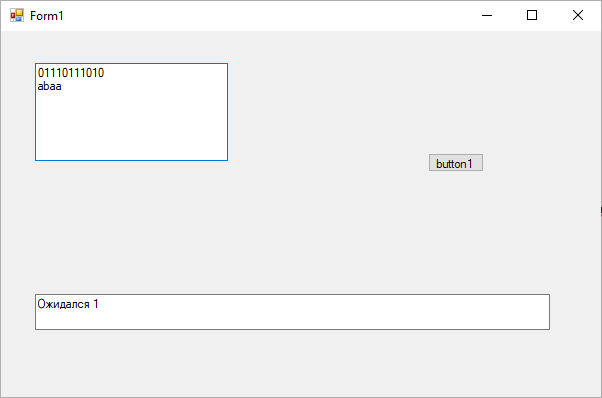
}

}

}

**Результаты тестирования:**





**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 устраняется путем преобразования грамматики в праворекурсивную грамматику.

1. Простая факторизация (Simple Left Factoring):

Этот метод заключается в разделении правил продукции с левой рекурсией на два вида. Предположим, у нас есть правило продукции A -> Aα | β, где α и β - последовательности символов.

1. Факторизация по правому контексту (Right Factoring):

Если левая рекурсии нетривиальная (не просто A -> Aα), то применение факторизации по правому контексту может быть полезным. Этот метод заключается в выделении общего правого контекста в правилах продукции.

В рамках лабораторной работы будет использован метод факторизации грамматики для избежания левой рекурсии. В связи со следующими преимуществами:

1. Избежание бесконечных рекурсивных вызовов: Левая рекурсия может привести к бесконечному циклу при разборе, поскольку при попытке развернуть леворекурсивное правило, можно оказаться снова на том же самом символе. Использование факторизации позволяет избежать этой проблемы, переписывая правила так, чтобы они не включали левой рекурсии.
2. Сохранение структуры грамматики: Факторизация позволяет избежать изменений в общей структуре грамматики. Путем введения новых нетерминалов и переписывания правил вы сохраняете смысловую структуру грамматики, что облегчает понимание и поддержание кода разбора.
3. Сохранение читаемости и поддерживаемости: Преобразование левой рекурсии в правила без нее (как в вашем примере) часто делает грамматику более читаемой и понятной. Это может быть особенно полезным при работе в команде, где другие разработчики могут легче воспринимать и вносить изменения в грамматику.
4. Улучшение производительности разбора: Некоторые генераторы синтаксических анализаторов более эффективно обрабатывают грамматики без левой рекурсии, что может привести к улучшению производительности в процессе разбора.

В целом, факторизация грамматики является распространенным методом в языках формальных грамматик для обеспечения их корректного разбора и улучшения читаемости.

**Текст задания:**

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

**Грамматика:**

O → A = A | (L)

A → <1> | <2>

L→ True | False | U | not L | O

U → <2>

**Код программы:**

**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, LBracket, RBracket, Exclamation, Semicolon, Coma, Question, Not, True, False }; // тип символа

public enum TToken { lxmIdentifier, lxmNumber, lxmUnknown, lxmEmpty, lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmComma, lxmText, lxmtz, lxmdt, lxmr, lxmrs, lxmls, lxmTrue, lxmFalse, lxmNot};

public class CLex

{

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 == '!') enumFSelectionCharType = TCharType.Exclamation;

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

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

else if (chrFSelection == '?') enumFSelectionCharType = TCharType.Question;

else if (chrFSelection == ';' || chrFSelection == ',' || chrFSelection == '[' || chrFSelection == ']' || chrFSelection == '=' || chrFSelection == ':' || chrFSelection == '(' || chrFSelection == ')' || chrFSelection == 'T' || chrFSelection == 'F' || chrFSelection == 'N') 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 24

switch (enumFSelectionCharType)

{

case TCharType.Letter:

{

// a b c d

// A | B | B | B | B |

// B | | C | | |

// C |DFin| | | |

// DFin |DFin|DFin|DFin|DFin|

A:

{

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

{

TakeSymbol();

goto B;

}

}

B:

{

if (chrFSelection == 'b')

{

TakeSymbol();

goto C;

}

else throw new Exception("Вторые два символа всегда ba");

}

C:

{

if (chrFSelection == 'a')

{

TakeSymbol();

goto DFin;

}

else throw new Exception("Вторые два символа всегда ba");

}

DFin:

{

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

{

TakeSymbol();

goto DFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

case TCharType.Digit:

{

// 0 1

// A | B | C |

// B | | D |

// C | E | |

// D | | A |

// E | |FFin |

// FFin | | G |

// G | | H |

// H |FFin | |

A:

if (chrFSelection == '0')

{

TakeSymbol();

goto B;

}

else if (chrFSelection == '1')

{

TakeSymbol();

goto C;

}

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

B:

if (chrFSelection == '1')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался 1");

C:

if (chrFSelection == '0')

{

TakeSymbol();

goto E;

}

else throw new Exception("Ожидался 0");

D:

if (chrFSelection == '1')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидался 1");

E:

if (chrFSelection == '1')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 1");

FFin:

if (chrFSelection == '1')

{

TakeSymbol();

goto G;

}

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

else throw new Exception("Ожидалась 1");

G:

if (chrFSelection == '1')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался 1");

H:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидался 0");

}

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.lxmComma;

GetSymbol();

return;

}

if (chrFSelection == ':')

{

enumFToken = TToken.lxmdt;

GetSymbol();

return;

}

if (chrFSelection == '=')

{

enumFToken = TToken.lxmr;

GetSymbol();

return;

}

if (chrFSelection == 'T')

{

enumFToken = TToken.lxmTrue;

GetSymbol();

return;

}

if (chrFSelection == 'F')

{

enumFToken = TToken.lxmFalse;

GetSymbol();

return;

}

if (chrFSelection == 'N')

{

enumFToken = TToken.lxmNot;

GetSymbol();

return;

}

break;

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

}

}

}

}

**uSyntAnalyzer.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

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 O()

{

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

{

A();

if (Lex.enumPToken == TToken.lxmr)

{

Lex.NextToken();

A();

}

else throw new Exception("Ожидал =");

}

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

{

Lex.NextToken();

L();

if (Lex.enumPToken == TToken.lxmRightParenth)

{

throw new Exception("Конец слова, текст верный. Для продолжения ожидается (");

}

else throw new Exception("Ожидалась )");

}

else throw new Exception("Ожидалась (");

}

public void A()

{

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

{

Lex.NextToken();

}

else throw new Exception("ozhidal word or numb");

}

public void L()

{

if (Lex.enumPToken == TToken.lxmTrue || Lex.enumPToken == TToken.lxmFalse)

{

Lex.NextToken();

}

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

{

U();

}

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

{

Lex.NextToken();

L();

}

else

{

O();

}

}

public void U()

{

if (Lex.enumPToken == TToken.lxmIdentifier)

{

Lex.NextToken();

}

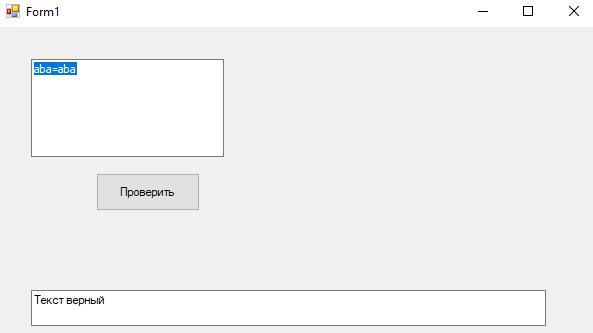
else throw new Exception("Ожидалось слово");

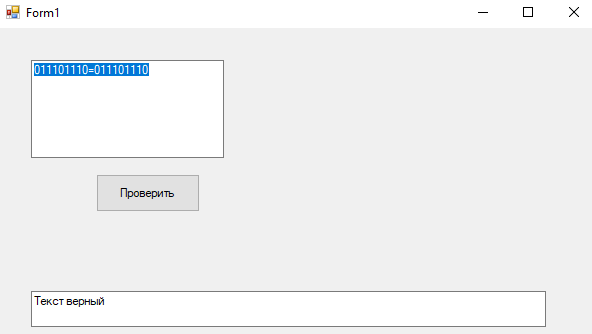
}

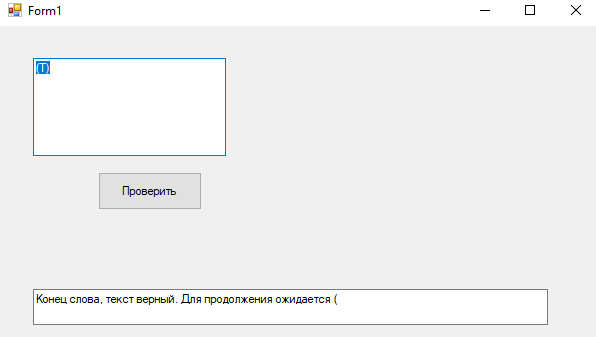
}

}

**Результаты тестирования:**







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

Основное назначение хеширования — проверка информации. Эта задача важна в огромном количестве случаев: от проверки паролей на сайте до сложных вычислений в блокчейне. Так как хеш — это уникальный код определенного набора данных, по нему можно понять, соответствует ли информация ожидаемой. Поэтому программа может хранить хеши вместо образца данных для сравнения. Это может быть нужно для защиты чувствительных сведений или экономии места.

Хеш-табли́ца — это структура данных, реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. Существуют два основных варианта хеш-таблиц: с цепочками и открытой адресацией.

Хеш-функция— функция, осуществляющая преобразование массива входных данных произвольной длины в выходную битовую строку установленной длины, выполняемое определённым алгоритмом. Преобразование, производимое хеш-функцией, называется хешированием. Исходные данные называются входным массивом, «ключом» или «сообщением».  
Общий жизненный цикл создания и использования хеш-таблицы в программировании включает следующие этапы:

1. **Инициализация:**
   * На этом этапе создаются основные структуры данных для хранения данных в хеш-таблице (например, массив, связанные списки, и т.д.).
   * Выбирается или разрабатывается хеш-функция, которая будет использоваться для преобразования ключей в индексы массива.
2. **Добавление данных:**
   * При поступлении новых данных (ключ-значение) вычисляется хеш от ключа с использованием выбранной хеш-функции.
   * Вычисленный хеш преобразуется в индекс массива, куда будет помещено значение.
   * Если в соответствующей ячейке массива уже есть данные, возможны различные стратегии разрешения коллизий (открытое адресное хеширование, цепочки и т.д.).
3. **Поиск данных:**
   * При поиске данных по ключу, хеш ключа снова вычисляется, и индекс массива определяется по вычисленному хешу.
   * Если в найденной ячейке есть данные, то производится сравнение ключей, и, при необходимости, выполняются дополнительные шаги в случае коллизий.
4. **Обновление и удаление данных:**
   * При обновлении или удалении данных сначала происходит поиск по ключу, чтобы найти соответствующую ячейку в массиве.
   * Если данные найдены, они обновляются или удаляются, в зависимости от операции.
5. **Обработка коллизий:**
   * Коллизии могут возникнуть, когда два разных ключа приводят к одному и тому же индексу массива.
   * Различные методы разрешения коллизий включают открытое адресное хеширование, метод цепочек и другие.
6. **Разрешение конфликтов:**
   * В случае возникновения конфликтов или неожиданных ситуаций может потребоваться разрешение проблем, таких как перехеширование, изменение хеш-функции или использование дополнительных структур данных.
7. **Оптимизация и управление памятью:**
   * В ходе работы с хеш-таблицей могут возникнуть потребности в оптимизации, особенно при изменении объема данных.
   * Оптимизация может включать в себя перехеширование для расширения или уменьшения размера таблицы, выбор более эффективных хеш-функций и другие меры.
8. **Завершение работы:**
   * При завершении работы программы или после завершения использования хеш-таблицы, если она больше не нужна, возможно освобождение памяти и другие завершающие действия.

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](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D0%BE%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE), можно задать [инъективную](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%8A%D0%B5%D0%BA%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D1%81%D1%82%D1%8C) 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.

Метод цепочек является наиболее простым методом разрешения коллизий. В ячейке массива мы будем хранить не элементы, а связанный список данных элементов. Потому как добавление в начало списка обладает асимптотикой , мы не испортим общую асимптотику, и она останется равной.

У данной реализации есть проблема: если списки будут очень сильно вырастать (в качестве крайнего случая можно рассмотреть хеш-функцию, которая возвращает константу для любого объекта), то мы получим асимптотику O(m), где m — число элементов во множестве, если размер массива фиксирован. Для избежания таких неприятностей вводится понятие коэффициент заполнения (он может быть равен, например, 1.5). Если при добавлении элемента оказывается, что доля числа элементов, находящихся в структуре данных по отношению к размеру массива, превосходит коэффициент заполнения, то происходит следующее: выделяется новый массив, размер которого превосходит размер старого массива (например, в 2 раза), и структура данных перестраивается на новом массиве.

Метод открытой адресации. В данном методе в ячейках хранятся сами элементы, а в случае коллизии происходит последовательность проб, то есть мы начинаем по некоторому алгоритму перебирать ячейки в надежде найти свободную. Это можно делать разными алгоритмами (линейная / квадратичная последовательности проб, двойное хеширование), каждый из которых обладает своими проблемами (например, возникновение первичных или вторичных кластеров).

Основные аспекты создания и использования этой хэш-таблицы:

1. **Структура классов:**
   * **CHashTableList**: Этот класс представляет собой список хэш-таблиц и содержит методы для работы с ними, такие как добавление, удаление и поиск лексических единиц.
   * **THashTable**: Этот класс реализует отдельную хэш-таблицу с методами для инициализации, добавления, удаления и поиска лексических единиц.
2. **Методы создания хэш-таблицы:**
   * **Resize**: Методы **Resize** используются для изменения размеров различных списков (хэш-таблиц, пользовательских таблиц, и т.д.) в классе **CHashTableList** в зависимости от заданного размера.
3. **Хэш-функции:**
   * **HashFunction\_Wainberger**: Реализует простую хэш-функцию для вычисления хэш-кода строки (**strALexicalUnit**).
4. **Методы работы с лексическими единицами:**
   * **SearchLexicalUnit**: Поиск лексической единицы в хэш-таблице.
   * **AddLexicalUnit**: Добавление лексической единицы в хэш-таблицу.
   * **DeleteLexicalUnit** и **DeleteLexicalCode**: Удаление лексической единицы или кода из хэш-таблицы.
5. **Работа с файлами:**
   * **Save** и **Load**: Сохранение и загрузка состояния хэш-таблицы из файла.
6. **Другие методы:**
   * **SetUserTable**, **SetUserData**, **GetUserData**: Методы для работы с пользовательскими данными, связанными с лексическими единицами.
7. **Работа с памятью:**
   * Используется механизм выделения и освобождения памяти (например, метод **Resize**), чтобы поддерживать оптимальное использование ресурсов.

В данном коде используется метод разрешения коллизий, называемый "линейное перехеширование" (linear probing). Этот метод предполагает поиск следующей доступной ячейки в таблице, если место, вычисленное хеш-функцией, уже занято. Реализация этого метода приведена в методе **ReHashFunction\_Line**.

Вот основные шаги линейного перехеширования в коде:

1. **Хеш-функция Wainberger:**

UInt32 HashFunction\_Wainberger(string strALexicalUnit)

{

UInt32 h = 0, g;

for (int i = 0, l = strALexicalUnit.Length; i < l; i++)

{

h = (h << 4) + strALexicalUnit[i];

g = h & 0xF0000000;

if (h != 0)

{

h ^= g >> 24 ^ g;

}

}

return h;

}

1. **Линейная функция перехеширования:**

int ReHashFunction\_Line(int h, string strALexicalUnit)

{

int i = 1, hi = h;

bool boolVFinish = false;

int hash2 = 7 - (int)(HashFunction\_Wainberger(strALexicalUnit) % 7); // Выбор второй хеш-функции

do

{

if (arrFHashTable[hi] == 0)

{

boolVFinish = true;

}

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

{

boolVFinish = true;

}

else

{

i++;

hi = (h + i \* hash2) % arrFHashTable.Count;

}

} while (!boolVFinish);

return hi;

}

Метод ReHashFunction\_Line применяется при возникновении коллизии, чтобы найти следующую доступную ячейку в таблице для вставки элемента. Это осуществляется путем последовательного прохода по ячейкам таблицы до тех пор, пока не будет найдено свободное место.

**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.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using nsLex;

using nsSynt;

using nsHashTables;

//011101110

namespace khafizovNR

{

public partial class Form1 : Form

{

public CHashTableList htl = new CHashTableList(2);

public Form1()

{

InitializeComponent();

tbFSource.AppendText("aba=aba" + "\r\n");

int n = tbFMessage.Lines.Length;

}

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

{

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

listBox1.Items.Clear();

listBox2.Items.Clear();

htl.TableToStringList(0, listTable);

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

listBox1.Items.Add(listTable[i]);

listTable.Clear();

htl.TableToStringList(1, listTable);

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

listBox2.Items.Add(listTable[i]);

listTable.Clear();

}

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

{

}

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.O();

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 btnFRecord\_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;

}

}

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 Search\_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))

{

tbFMessage2.AppendText("(Найден идентификатор)");

}

else tbFMessage2.AppendText("(Идентификатор не найден)");

break;

}

case TToken.lxmNumber:

{

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

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

{

tbFMessage2.AppendText("(Найден номер)");

}

else tbFMessage2.AppendText("(Номер 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 Ydalenie\_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);

TablesToMemo(this, e);

break;

}

case TToken.lxmNumber:

{

s1 = "num " + Lex.strPLexicalUnit;

htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 1);

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 Izmenenie\_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;

object p = listBox1.SelectedItem;

string index = Convert.ToString(p); //kotoroe menyaem

htl.DeleteLexicalUnit(index, 0);

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

TablesToMemo(this, e);

break;

}

case TToken.lxmNumber:

{

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

object p = listBox2.SelectedItem;

string index = Convert.ToString(p); //kotoroe menyaem

htl.DeleteLexicalUnit(index, 1);

htl.AddLexicalUnit(Lex.strPLexicalUnit, 1, 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;

}

}

}

}

**Usyntanalyzer.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

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 O()

{

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

{

A();

if (Lex.enumPToken == TToken.lxmr)

{

Lex.NextToken();

A();

}

else throw new Exception("Ожидал =");

}

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

{

Lex.NextToken();

L();

if (Lex.enumPToken == TToken.lxmRightParenth)

{

throw new Exception("Конец слова, текст верный. Для продолжения ожидается (");

}

else throw new Exception("Ожидалась )");

}

else throw new Exception("Ожидалась (");

}

public void A()

{

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

{

Lex.NextToken();

}

else throw new Exception("ozhidal word or numb");

}

public void L()

{

if (Lex.enumPToken == TToken.lxmTrue || Lex.enumPToken == TToken.lxmFalse)

{

Lex.NextToken();

}

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

{

U();

}

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

{

Lex.NextToken();

L();

}

else

{

O();

}

}

public void U()

{

if (Lex.enumPToken == TToken.lxmIdentifier)

{

Lex.NextToken();

}

else throw new Exception("Ожидалось слово");

}

}

**Class1.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, LBracket, RBracket, Exclamation, Semicolon, Coma, Question, Not, True, False }; // тип символа

public enum TToken { lxmIdentifier, lxmNumber, lxmUnknown, lxmEmpty, lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmComma, lxmText, lxmtz, lxmdt, lxmr, lxmrs, lxmls, lxmTrue, lxmFalse, lxmNot };

public class CLex

{

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 == '!') enumFSelectionCharType = TCharType.Exclamation;

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

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

else if (chrFSelection == '?') enumFSelectionCharType = TCharType.Question;

else if (chrFSelection == ';' || chrFSelection == ',' || chrFSelection == '[' || chrFSelection == ']' || chrFSelection == '=' || chrFSelection == ':' || chrFSelection == '(' || chrFSelection == ')' || chrFSelection == 'T' || chrFSelection == 'F' || chrFSelection == 'N') 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 24

switch (enumFSelectionCharType)

{

case TCharType.Letter:

{

// a b c d

// A | B | B | B | B |

// B | | C | | |

// C |DFin| | | |

// DFin |DFin|DFin|DFin|DFin|

A:

{

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

{

TakeSymbol();

goto B;

}

}

B:

{

if (chrFSelection == 'b')

{

TakeSymbol();

goto C;

}

else throw new Exception("Вторые два символа всегда ba");

}

C:

{

if (chrFSelection == 'a')

{

TakeSymbol();

goto DFin;

}

else throw new Exception("Вторые два символа всегда ba");

}

DFin:

{

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

{

TakeSymbol();

goto DFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

case TCharType.Digit:

{

// 0 1

// A | B | C |

// B | | D |

// C | E | |

// D | | A |

// E | |FFin |

// FFin | | G |

// G | | H |

// H |FFin | |

A:

if (chrFSelection == '0')

{

TakeSymbol();

goto B;

}

else if (chrFSelection == '1')

{

TakeSymbol();

goto C;

}

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

B:

if (chrFSelection == '1')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался 1");

C:

if (chrFSelection == '0')

{

TakeSymbol();

goto E;

}

else throw new Exception("Ожидался 0");

D:

if (chrFSelection == '1')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидался 1");

E:

if (chrFSelection == '1')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 1");

FFin:

if (chrFSelection == '1')

{

TakeSymbol();

goto G;

}

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

else throw new Exception("Ожидалась 1");

G:

if (chrFSelection == '1')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался 1");

H:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидался 0");

}

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.lxmComma;

GetSymbol();

return;

}

if (chrFSelection == ':')

{

enumFToken = TToken.lxmdt;

GetSymbol();

return;

}

if (chrFSelection == '=')

{

enumFToken = TToken.lxmr;

GetSymbol();

return;

}

if (chrFSelection == 'T')

{

enumFToken = TToken.lxmTrue;

GetSymbol();

return;

}

if (chrFSelection == 'F')

{

enumFToken = TToken.lxmFalse;

GetSymbol();

return;

}

if (chrFSelection == 'N')

{

enumFToken = TToken.lxmNot;

GetSymbol();

return;

}

break;

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

}

}

}

}

**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;

}

//HashFunc

UInt32 HashFunction\_Wainberger(string strALexicalUnit)

{

UInt32 h = 0;

for (int i = 0, l = strALexicalUnit.Length; i < l; i++)

{

h += strALexicalUnit[i]; //Функция хэширования

}

return h; }

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) % (Int32)(arrFHashTable.Count); //функция избавления от коллизий

}

} while (!boolVFinish);

return hi;

}

public void HashIndex(string strALexicalUnit)

{

int h;

h = (Int32)HashFunction\_Wainberger(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);

}

}

}

u

**HashTableList.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++)

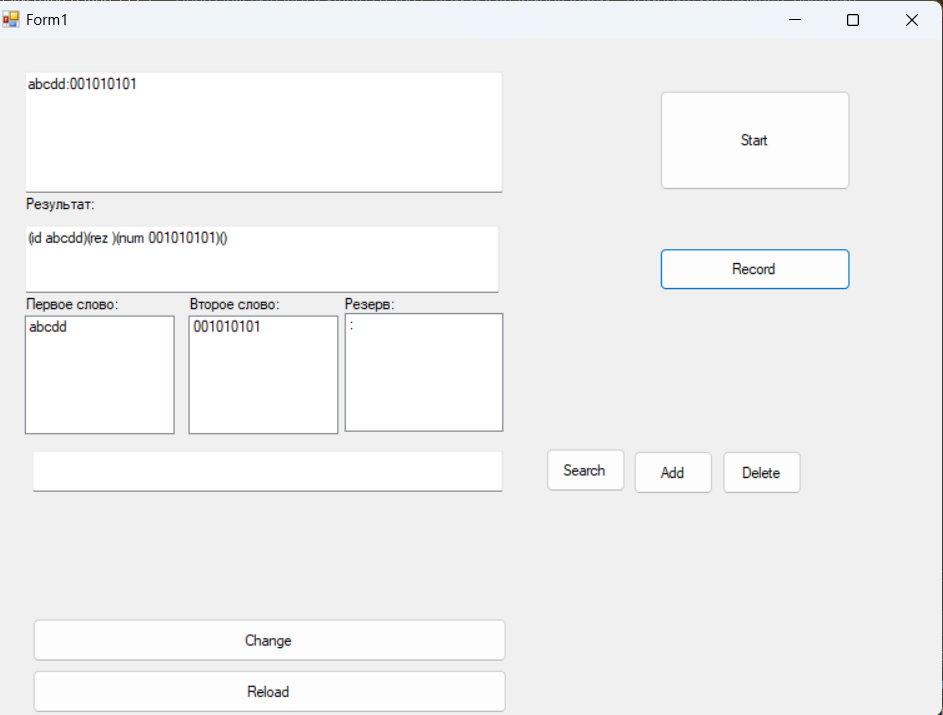
sList.Add(arrFHeapTable[i].strFLexicalUnit);

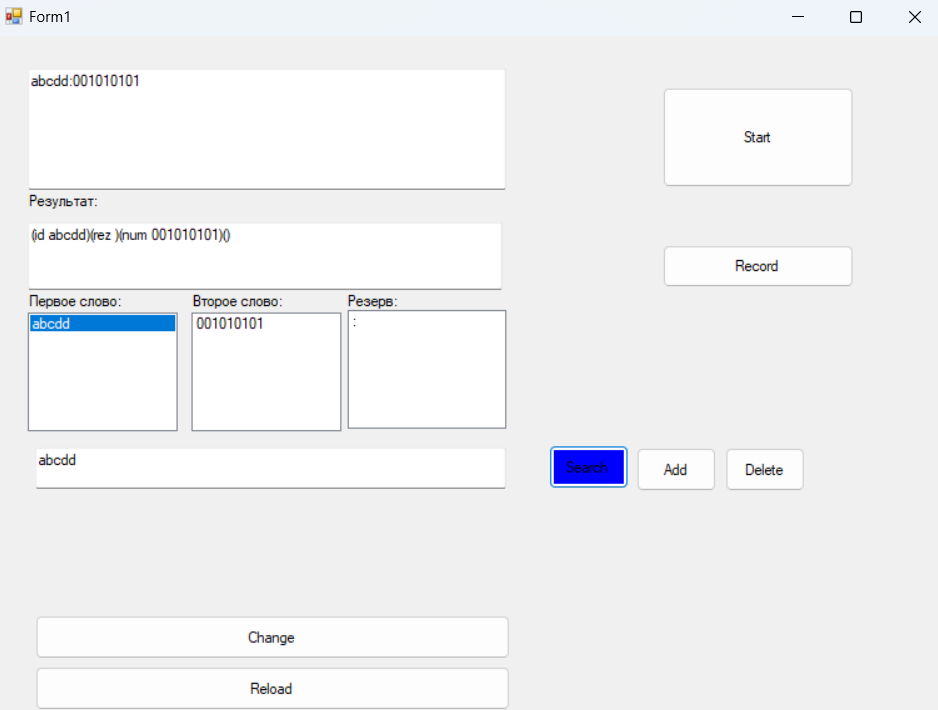
}

}

}

**Результат работы:**





**Лабораторная работа № 5. Построение синтаксического дерева (2 часа)**

**Краткое теоретическое обоснование:**

Построение синтаксического дерева (или абстрактного синтаксического дерева, AST) — это важная часть синтаксического анализа, которая преобразует текст программы или входные данные в структуру данных, которая представляет собой абстракцию синтаксической структуры этого текста. Синтаксическое дерево позволяет легко анализировать и обрабатывать синтаксическую структуру программы, а также выполнять различные виды статического анализа или генерации кода.

Вот основные шаги построения синтаксического дерева:

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.

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.

Построение синтаксического дерева является фундаментальным этапом при разработке компиляторов и интерпретаторов, а также при создании инструментов для анализа кода.

**Абстрактное синтаксическое дерево (AST):**

AST — это более абстрактное и упрощенное представление синтаксической структуры программы по сравнению с синтаксическим деревом. Оно убирает некоторые детали, такие как скобки и другие второстепенные элементы, и оставляет только важные узлы, представляющие смысл программы. AST обычно используется для дальнейшей обработки и анализа программы, такой как оптимизация и генерация кода.

**Аннотации и информация:**

Синтаксическое дерево может быть аннотировано дополнительной информацией, такой как типы данных, области видимости, местоположение в исходном коде и другие атрибуты. Эта информация полезна при выполнении семантического анализа и других видов анализа.

**Генерация кода:**

В контексте компиляторов, после построения синтаксического дерева можно использовать его для генерации промежуточного кода или целевого кода для целевой платформы.

**Оптимизация:**

Синтаксическое дерево может подвергаться различным видам оптимизаций, чтобы улучшить производительность программы. Например, можно проводить оптимизации вычислений, устранение избыточных вычислений и другие виды оптимизаций.

Рекурсивный спуск и генерация синтаксического анализатора:

Для построения синтаксического дерева можно использовать метод рекурсивного спуска, который часто используется при разработке синтаксических анализаторов. Рекурсивный спуск означает, что анализатор разбирает текст программы, вызывая себя рекурсивно для каждой синтаксической конструкции.

Многие инструменты для генерации синтаксических анализаторов (например, ANTLR, Bison, YACC) позволяют создавать код для синтаксического анализа на основе описания грамматики, что упрощает разработку.

Построение синтаксического дерева является фундаментальной частью компиляторов, интерпретаторов и других инструментов для анализа и обработки языка. Корректное и эффективное построение дерева позволяет проводить разнообразные виды анализа и манипуляций с программным кодом.

**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 WindowsFormsApp2

{

public partial class Form1 : Form

{

public CHashTableList htl = new CHashTableList(3);

public Form1()

{

InitializeComponent();

//tbFSource.AppendText("((abc)[101])" + "\r\n");

tbFSource.AppendText("((((abccc)[010101110]))[[[101](abca)]])\r\n" + "\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(listTable[i]);

listTable.Clear();

htl.TableToStringList(1, listTable);

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

listBox2.Items.Add(listTable[i]);

listTable.Clear();

htl.TableToStringList(2, listTable);

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

listBox3.Items.Add(listTable[i]);

listTable.Clear();

}

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

{

tbFMessage.Clear();

uSyntAnalyzer Synt = new uSyntAnalyzer(SyntTree);

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 Form1\_Load(object sender, EventArgs e)

{

}

private void btnFRecord\_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.lxmRightParenth:

case TToken.lxmLeftParenth:

case TToken.lxmrs:

case TToken.lxmls:

case TToken.lxmr:

case TToken.lxmdt:

case TToken.lxmComma:

case TToken.lxmDot:

{

s1 = "NS " + 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 button1\_Click(object sender, EventArgs e)

{

listBox3.Items.Remove(listBox3.SelectedItem);

string s = listBox3.Text.ToString();

htl.DeleteLexicalUnit(s, 2);

}

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

{

int b = 0;

listBox3.Items.Add(textBox1.Text);

string s = textBox1.Text;

htl.AddLexicalUnit(s, 2,ref b);

}

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

{

int b = 0;

string s = listBox3.Text.ToString();

int index = listBox3.Items.IndexOf(s);

listBox3.Items.Remove(listBox3.Items[index]);

htl.DeleteLexicalUnit(s, 2);

s = textBox1.Text;

listBox3.Items.Add(textBox1.Text);

htl.AddLexicalUnit(s, 2, ref b);

}

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

{

int b = 0;

bool flag = htl.SearchLexicalUnit(textBox1.Text, 2, ref b);

if (flag)

{

button4.BackColor = Color.Green;

}

else

{

button4.BackColor = Color.Red;

}

}

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

{

listBox1.Items.Clear();

listBox2.Items.Clear();

listBox3.Items.Clear();

}

}

}

**THashTable.cs**  
  
using System;

using System.Collections.Generic;

using System.IO;

using System.Windows.Forms;

using System.Linq;

using System.Threading;

namespace WindowsFormsApp2

{

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\_Wainberger(string strALexicalUnit)

{

UInt32 hash = 5381;

foreach (char c in strALexicalUnit)

{

// Учтем ASCII-код символа

if (c == '(' || c == ')' || c == '[' || c == ']' || c == '=')

{

hash = ((hash << 5) + hash) ^ (UInt32)c;

}

else

{

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\_Wainberger(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)

{

intALexicalCode = objFHeap.intPFreeItem;

objFHeap.AddLexicalUnit(strAlexicalUnit, (byte)0, intFHashIndex, ref intALexicalCode);

arrFHashTable[intFHashIndex] = intALexicalCode;

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;

using System.Threading;

using WindowsFormsApp2;

namespace WindowsFormsApp2

{

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;

}

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

}

}  
  
**Program.cs**  
using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp2

{

static class Program

{

/// <summary>

/// Главная точка входа для приложения.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

}

}

**THeap.cs**using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

using System.Windows.Forms;

namespace WindowsFormsApp2

{

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);

}

}

}

**uLex.cs**  
  
using System;

using System.Collections.Generic;

using System.Text;

namespace WindowsFormsApp2

{

class uLex

{

}

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

public enum TCharType { Letter, Digit, EndRow, EndText, Space, RevS1, RevS2, RevS3, RevS4, RevS5, RevS6, RevS7, RevS8, NonSymbol, ReservedSymbol }; // тип символа

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

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 = 0;

private int intFSourceColSelection = -1;

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 == '(') enumFSelectionCharType = TCharType.ReservedSymbol;

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

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

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

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

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

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

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

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

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

//else enumFSelectionCharType = TCharType.NonSymbol;

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();

}

switch (enumFSelectionCharType)

{

case TCharType.Letter:

{

// a b c d

// A | B | | | |

// B | | | |CFin|

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

A:

{

if (chrFSelection == 'b')

{

TakeSymbol();

goto CFin;

}

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

{

TakeSymbol();

goto BFin;

}

else throw new Exception("Этой буквы нет");

}

BFin:

{

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

{

TakeSymbol();

goto BFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

CFin:

{

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

{

TakeSymbol();

goto CFin;

}

if (chrFSelection == 'a')

{

throw new Exception("Если слово началось в буквы 'b', то оно не должно содержать букву 'a'.");

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

case TCharType.Digit:

{

// 0 1

// A | B | C |

// B | D | |

// C | E | |

// D | A | |

// E | |FFin |

// FFin | | G |

// G | H | |

A:

if (chrFSelection == '0')

{

TakeSymbol();

goto B;

}

else if (chrFSelection == '1')

{

TakeSymbol();

goto C;

}

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

B:

if (chrFSelection == '1')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался 1");

C:

if (chrFSelection == '0')

{

TakeSymbol();

goto E;

}

else throw new Exception("Ожидался 0");

D:

if (chrFSelection == '0')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидался 0");

E:

if (chrFSelection == '1')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 1");

FFin:

if (chrFSelection == '1')

{

TakeSymbol();

goto G;

}

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

else throw new Exception("Ожидалась 1");

G:

if (chrFSelection == '1')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался 0");

H:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидался 0");

}

case TCharType.ReservedSymbol:

{

{

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

{

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

}

GetSymbol();

}

if (chrFSelection == '(')

{

enumFToken = TToken.lxmLeftParenth;

strFLexicalUnit += '(';

GetSymbol();

return;

}

if (chrFSelection == '|')

{

enumFToken = TToken.lxStolb;

GetSymbol();

return;

}

if (chrFSelection == ')')

{

enumFToken = TToken.lxmRightParenth;

strFLexicalUnit += ')';

GetSymbol();

return;

}

if (chrFSelection == '[')

{

enumFToken = TToken.lxmls;

strFLexicalUnit += '[';

GetSymbol();

return;

}

if (chrFSelection == ']')

{

enumFToken = TToken.lxmrs;

strFLexicalUnit += ']';

GetSymbol();

return;

}

if (chrFSelection == ',')

{

enumFToken = TToken.lxmComma;

strFLexicalUnit += ',';

GetSymbol();

return;

}

if (chrFSelection == '.')

{

enumFToken = TToken.lxmDot;

strFLexicalUnit += '.';

GetSymbol();

return;

}

if (chrFSelection == ':')

{

enumFToken = TToken.lxmdt;

strFLexicalUnit += ':';

GetSymbol();

return;

}

if (chrFSelection == '=')

{

enumFToken = TToken.lxmr;

strFLexicalUnit += '=';

GetSymbol();

return;

}

break;

}

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

}

}

}

}

**uSyntAnalyzer.cs**  
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp2

{

internal 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();

TreeView tree;

public uSyntAnalyzer(TreeView treeView)

{

tree = treeView;

}

public void S()

{

TreeNode parent = new TreeNode("S");

tree.Nodes.Add(parent);

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

{

A(parent);

}

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

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

B(parent);

}

throw new Exception("Конец слова, текст верный. Для продолжения ожидается ;");

}

public void C(TreeNode highParent)

{

if (Lex.enumPToken == TToken.lxmtz)

{

Lex.NextToken();

A(highParent);

if (Lex.enumPToken == TToken.lxmtz)

C(highParent);

}

}

public void A(TreeNode highParent)

{

TreeNode parent = new TreeNode("A");

highParent.Nodes.Add(parent);

if (Lex.enumPToken == TToken.lxmIdentifier)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

}

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

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmLeftParenth)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

A(parent);

if (Lex.enumPToken == TToken.lxmRightParenth)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmls)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

B(parent);

if (Lex.enumPToken == TToken.lxmrs)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmRightParenth)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

}

else throw new Exception("Ожидалась )");

}

else throw new Exception("Ожидалась ]");

}

else throw new Exception("Ожидалась [");

}

else throw new Exception("Ожидалась )");

}

else throw new Exception("Ожидалась (");

}

else throw new Exception("Ожидалось (");

}

public void B(TreeNode highParent)

{

TreeNode parent = new TreeNode("B");

highParent.Nodes.Add(parent);

if (Lex.enumPToken == TToken.lxmNumber)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

}

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

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmls)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

B(parent);

if (Lex.enumPToken == TToken.lxmrs)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmLeftParenth)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

A(parent);

if (Lex.enumPToken == TToken.lxmRightParenth)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmrs)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

}

else throw new Exception("Ожидалась ]");

}

else throw new Exception("Ожидалась )");

}

else throw new Exception("Ожидалась (");

}

else throw new Exception("Ожидалась ]");

}

else throw new Exception("Ожидалась [");

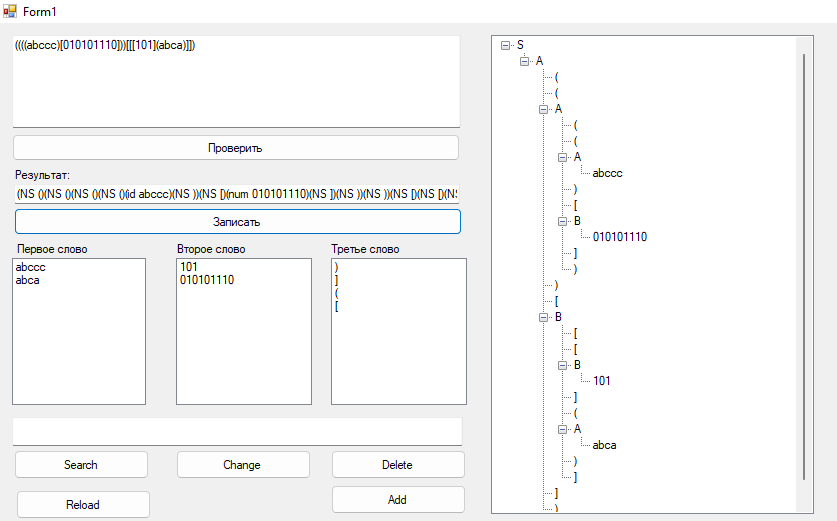
}

else throw new Exception("Ожидалось [");

}

}

}



**Лабораторная работа № 6.**

**Разработка генератора (2 часа)**

**Краткое теоретическое обоснование:**

**Структурированное представление в TreeView:**

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.

**Основные элементы структурированного представления TreeView включают в себя:**

Узлы: Каждый узел представляет собой элемент в дереве и может содержать данные и/или другие дочерние узлы.

**Дерево:** Совокупность всех узлов и их отношений между собой формирует дерево, которое отображается в виде иерархической структуры.

**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.

**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.

Примеры задач, для которых могут быть разработаны генераторы, включают в себя:

Генерация исходного кода на основе спецификации (например, генераторы кода для языков программирования).

Генерация документации на основе комментариев в исходном коде (например, генераторы документации).

Генерация структур данных или конфигурационных файлов из описаний.

Генерация отчетов и отображений из данных.

**Разработка генератора обычно включает в себя следующие шаги:**

1. Определение исходных данных или спецификаций, на основе которых будет производиться генерация.
2. Создание алгоритма или логики генерации.
3. Разработка генератора, который реализует этот алгоритм.
4. Тестирование и отладка генератора.
5. Интеграция генератора в рабочий процесс разработки или другой инструмент.

**Form1.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Diagnostics;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

namespace WindowsFormsApp2

{

public partial class Form1 : Form

{

public CHashTableList htl1 = new CHashTableList(1);

public CHashTableList htl2 = new CHashTableList(2);

public CHashTableList htl3 = new CHashTableList(3);

public Form1()

{

InitializeComponent();

//tbFSource.AppendText("((abc)[101])" + "\r\n");

tbFSource.AppendText("((((abccc)[010101110]))[[[101](abca)]])\r\n" + "\r\n");

//tbFSource.AppendText("((((abccc)[010101110]))[[[101](((abccacc)[101110110]))]])\r\n" + "\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();

htl1.TableToStringList(0, listTable);

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

listBox1.Items.Add(listTable[i]);

listTable.Clear();

htl2.TableToStringList(1, listTable);

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

listBox2.Items.Add(listTable[i]);

listTable.Clear();

htl3.TableToStringList(2, listTable);

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

listBox3.Items.Add(listTable[i]);

listTable.Clear();

}

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

{

tbFMessage.Clear();

uSyntAnalyzer Synt = new uSyntAnalyzer(SyntTree, SyntTree2);

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;

}

SyntTree.ExpandAll();

SyntTree2.ExpandAll();

}

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

{

}

private void btnFRecord\_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 (htl1.AddLexicalUnit(Lex.strPLexicalUnit, 0, ref b))

{

TablesToMemo(this, e);

}

break;

}

case TToken.lxmNumber:

{

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

if (htl2.AddLexicalUnit(Lex.strPLexicalUnit, 1, ref b))

{

TablesToMemo(this, e);

}

break;

}

case TToken.lxmRightParenth:

case TToken.lxmLeftParenth:

case TToken.lxmrs:

case TToken.lxmls:

case TToken.lxmr:

case TToken.lxmdt:

case TToken.lxmComma:

case TToken.lxmDot:

{

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

if (htl3.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;

}

SyntTree.ExpandAll();

SyntTree2.ExpandAll();

}

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

{

listBox2.Items.Remove(listBox2.SelectedItem);

string s2 = listBox2.Text.ToString();

htl2.DeleteLexicalUnit(s2, 1);

listBox3.Items.Remove(listBox3.SelectedItem);

string s3 = listBox3.Text.ToString();

htl3.DeleteLexicalUnit(s3, 2);

}

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

{

int b = 0;

listBox3.Items.Add(textBox1.Text);

string s = textBox1.Text;

htl3.AddLexicalUnit(s, 2, ref b);

}

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

{

int b = 0;

string s = listBox3.Text.ToString();

int index = listBox3.Items.IndexOf(s);

listBox3.Items.Remove(listBox3.Items[index]);

htl3.DeleteLexicalUnit(s, 2);

s = textBox1.Text;

listBox3.Items.Add(textBox1.Text);

htl3.AddLexicalUnit(s, 2, ref b);

}

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

{

int b = 0;

bool flag = htl2.SearchLexicalUnit(textBox1.Text, 1, ref b);

if (flag)

{

button4.BackColor = Color.Green;

}

else

{

button4.BackColor = Color.Red;

}

}

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

{

SyntTree.Nodes.Clear();

SyntTree2.Nodes.Clear();

foreach (var item in listBox1.Items)

{

//listBox2.Items.Remove(item);

string s1 = item.ToString();

htl1.DeleteLexicalUnit(s1, 0);

}

listBox1.Items.Clear();

foreach (var item in listBox2.Items)

{

//listBox2.Items.Remove(item);

string s2 = item.ToString();

htl2.DeleteLexicalUnit(s2, 1);

}

listBox2.Items.Clear();

foreach (var item in listBox3.Items)

{

//listBox2.Items.Remove(item);

string s3 = item.ToString();

htl3.DeleteLexicalUnit(s3, 2);

}

listBox3.Items.Clear();

}

private void button6\_Click(object sender, EventArgs e)

{

TreeConverterSwap treeConverter = new TreeConverterSwap();

treeConverter.ConvertBinaryToDecimal(SyntTree2);

}

private void button7\_Click(object sender, EventArgs e)

{

try

{

Duplicate duplicate = new Duplicate(SyntTree, htl2);

duplicate.ChekingForDuplicate(SyntTree);

throw new Exception("Текст верный");

}

catch (Exception exc)

{

tbFMessage.Text = "";

tbFMessage.Text += exc.Message;

}

}

private void button8\_Click(object sender, EventArgs e)

{

TreeConverter treeConverter = new TreeConverter();

treeConverter.ConvertBinaryToDecimal(SyntTree2);

}

}

}

**HashTables.cs**

using System;

using System.Collections.Generic;

using System.IO;

using System.Windows.Forms;

using System.Linq;

using System.Threading;

namespace WindowsFormsApp2

{

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\_Wainberger(string strALexicalUnit)

{

UInt32 hash = 5381;

foreach (char c in strALexicalUnit)

{

// Учтем ASCII-код символа

if (c == '(' || c == ')' || c == '[' || c == ']' || c == '=')

{

hash = ((hash << 5) + hash) ^ (UInt32)c;

}

else

{

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\_Wainberger(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)

{

intALexicalCode = objFHeap.intPFreeItem;

//objFHeap.AddLexicalUnit(strAlexicalUnit, (byte)0, intFHashIndex, ref intALexicalCode);

arrFHashTable[intFHashIndex] = intALexicalCode;

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;

using System.Threading;

using WindowsFormsApp2;

namespace WindowsFormsApp2

{

public class CHashTableList

{

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

private bool boolFIsSaved;

public bool boolFIsLoaded;

private byte byteFTablesSize;

private int Count = 0;

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 int CountLexicalUnit(string strALexicalUnit, byte byteATable, ref int intALexicalCode)

{

Count += arrFHashTableList[byteATable].SearchLexicalUnit(strALexicalUnit, ref intALexicalCode, Count);

return Count;

}\*/

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

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;

}

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

}

}

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp2

{

static class Program

{

/// <summary>

/// Главная точка входа для приложения.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Application.Run(new Form1());

}

}

}

**THeap.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.IO;

using System.Windows.Forms;

namespace WindowsFormsApp2

{

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);

}

}

}

**TreeConverter.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp2

{

internal class TreeConverter

{

public void ConvertBinaryToDecimal(TreeView tree)

{

foreach (TreeNode node in tree.Nodes)

{

ConvertBinaryToDecimal(node);

}

/\*foreach (TreeNode node in tree.Nodes)

{

SpecialConverter(node);

}\*/

}

private void ConvertBinaryToDecimal(TreeNode node)

{

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

if (IsBin(binaryValue))

{

int decimalValue = Convert.ToInt32(binaryValue, 2);

node2.Text = decimalValue.ToString();

}

}

/\*if (node.Text == "A")

{

if (node.Nodes.Count == 1)

{

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

if (IsBin(binaryValue))

{

int decimalValue = Convert.ToInt32(binaryValue, 2);

node2.Text = decimalValue.ToString();

}

node.Parent.Nodes.Insert(node.Index, node2.Text);

}

node.Remove();

}

}\*/

foreach (TreeNode childNode in node.Nodes)

{

ConvertBinaryToDecimal(childNode);

}

}

private bool IsBin(string value)

{

foreach (var c in value)

if (c != '0' && c != '1')

return false;

return true;

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Xml.Linq;

namespace WindowsFormsApp2

{

internal class TreeConverterSwap

{

public Dictionary<TreeNode, string> myDict = new Dictionary<TreeNode, string>();

public List<TreeNode> list = new List<TreeNode>();

public List<string> strings = new List<string>();

public string t;

public int k = 0;

public void ConvertBinaryToDecimal(TreeView tree)

{

foreach (TreeNode node in tree.Nodes)

{

ConvertBinaryToDecimal(node);

}

/\*foreach (TreeNode node in tree.Nodes)

{

SpecialConverter(node);

}\*/

foreach (var item in myDict)

{

list.Add(item.Key);

strings.Add(item.Value);

}

foreach (TreeNode node in tree.Nodes)

{

Perebor(node);

}

}

private void SpecialConverter(TreeNode node)

{

if(node.Text == "A")

{

if (node.Nodes.Count == 8 && node.Index == 2)

{

foreach (TreeNode node2 in node.Nodes)

{

node.Parent.Nodes.Insert(node.Index, node2);

}

node.Remove();

}

}

}

private void ConvertBinaryToDecimal(TreeNode node)

{

if (node.Text == "B")

{

if (node.Nodes.Count == 1)

{

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

if (IsBin(binaryValue))

{

//int decimalValue = Convert.ToInt32(binaryValue, 2);

if (!myDict.ContainsKey(node2))

{

myDict.Add(node2, binaryValue);

}

}

}

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

if (IsBin(binaryValue))

{

//int decimalValue = Convert.ToInt32(binaryValue, 2);

if (!myDict.ContainsKey(node2))

{

myDict.Add(node2, binaryValue);

}

}

node.Parent.Nodes.Insert(node.Index, node2.Text);

}

node.Remove();

}

}

if (node.Text == "A")

{

if (node.Nodes.Count == 1)

{

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

if (IsBin(binaryValue))

{

//int decimalValue = Convert.ToInt32(binaryValue, 2);

if (!myDict.ContainsKey(node2))

{

myDict.Add(node2, binaryValue);

}

}

node.Parent.Nodes.Insert(node.Index, node2.Text);

}

node.Remove();

}

}

foreach (TreeNode childNode in node.Nodes)

{

ConvertBinaryToDecimal(childNode);

}

}

private bool IsBin(string value)

{

foreach (var c in value)

if (c != '0' && c != '1')

return false;

return true;

}

private void Perebor(TreeNode node)

{

if (node.Nodes.Count > 0)

{

foreach (TreeNode Node in node.Nodes)

{

/\*if (Node.Text == strings[1])

{

Node.Text = strings[0];

}

else if (Node.Text == strings[0])

{

Node.Text = strings[1];

}\*/

for (int i = 0, j = strings.Count-1; i < strings.Count; i++, j--)

{

if(Node.Text == strings[i])

{

t = strings[i];

Node.Text = strings[j];

break;

}

}

}

}

foreach (TreeNode childNode in node.Nodes)

{

Perebor(childNode);

}

}

}

}

**uLex.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace WindowsFormsApp2

{

class uLex

{

}

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

public enum TCharType { Letter, Digit, EndRow, EndText, Space, RevS1, RevS2, RevS3, RevS4, RevS5, RevS6, RevS7, RevS8, NonSymbol, ReservedSymbol }; // тип символа

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

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 = 0;

private int intFSourceColSelection = -1;

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 == '(') enumFSelectionCharType = TCharType.ReservedSymbol;

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

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

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

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

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

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

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

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

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

//else enumFSelectionCharType = TCharType.NonSymbol;

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();

}

switch (enumFSelectionCharType)

{

case TCharType.Letter:

{

// a b c d

// A | B | | | |

// B | | | |CFin|

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

A:

{

if (chrFSelection == 'b')

{

TakeSymbol();

goto CFin;

}

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

{

TakeSymbol();

goto BFin;

}

else throw new Exception("Этой буквы нет");

}

BFin:

{

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

{

TakeSymbol();

goto BFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

CFin:

{

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

{

TakeSymbol();

goto CFin;

}

if (chrFSelection == 'a')

{

throw new Exception("Если слово началось в буквы 'b', то оно не должно содержать букву 'a'.");

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

case TCharType.Digit:

{

// 0 1

// A | B | C |

// B | D | |

// C | E | |

// D | A | |

// E | |FFin |

// FFin | | G |

// G | H | |

A:

if (chrFSelection == '0')

{

TakeSymbol();

goto B;

}

else if (chrFSelection == '1')

{

TakeSymbol();

goto C;

}

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

B:

if (chrFSelection == '1')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался 1");

C:

if (chrFSelection == '0')

{

TakeSymbol();

goto E;

}

else throw new Exception("Ожидался 0");

D:

if (chrFSelection == '0')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидался 0");

E:

if (chrFSelection == '1')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 1");

FFin:

if (chrFSelection == '1')

{

TakeSymbol();

goto G;

}

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

else throw new Exception("Ожидалась 1");

G:

if (chrFSelection == '1')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался 0");

H:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидался 0");

}

case TCharType.ReservedSymbol:

{

{

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

{

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

}

GetSymbol();

}

if (chrFSelection == '(')

{

enumFToken = TToken.lxmLeftParenth;

strFLexicalUnit += '(';

GetSymbol();

return;

}

if (chrFSelection == '|')

{

enumFToken = TToken.lxStolb;

GetSymbol();

return;

}

if (chrFSelection == ')')

{

enumFToken = TToken.lxmRightParenth;

strFLexicalUnit += ')';

GetSymbol();

return;

}

if (chrFSelection == '[')

{

enumFToken = TToken.lxmls;

strFLexicalUnit += '[';

GetSymbol();

return;

}

if (chrFSelection == ']')

{

enumFToken = TToken.lxmrs;

strFLexicalUnit += ']';

GetSymbol();

return;

}

if (chrFSelection == ',')

{

enumFToken = TToken.lxmComma;

strFLexicalUnit += ',';

GetSymbol();

return;

}

if (chrFSelection == '.')

{

enumFToken = TToken.lxmDot;

strFLexicalUnit += '.';

GetSymbol();

return;

}

if (chrFSelection == ':')

{

enumFToken = TToken.lxmdt;

strFLexicalUnit += ':';

GetSymbol();

return;

}

if (chrFSelection == '=')

{

enumFToken = TToken.lxmr;

strFLexicalUnit += '=';

GetSymbol();

return;

}

break;

}

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

}

}

}

}

**uSyntAnalyzer.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp2

{

internal 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();

TreeView tree;

TreeView tree2;

public uSyntAnalyzer(TreeView treeView, TreeView treeView2)

{

tree = treeView;

tree2 = treeView2;

}

public void S()

{

TreeNode parent = new TreeNode("S");

tree.Nodes.Add(parent);

TreeNode parent2 = new TreeNode("S");

tree2.Nodes.Add(parent2);

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

{

A(parent, parent2);

}

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

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

B(parent, parent2);

}

throw new Exception("Конец слова, текст верный. Для продолжения ожидается ;");

}

public void C(TreeNode highParent, TreeNode highParent2)

{

if (Lex.enumPToken == TToken.lxmtz)

{

Lex.NextToken();

A(highParent, highParent2);

if (Lex.enumPToken == TToken.lxmtz)

C(highParent, highParent2);

}

}

public void A(TreeNode highParent, TreeNode highParent2)

{

TreeNode parent = new TreeNode("A");

highParent.Nodes.Add(parent);

TreeNode parent2 = new TreeNode("A");

highParent2.Nodes.Add(parent2);

if (Lex.enumPToken == TToken.lxmIdentifier)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

}

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

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmLeftParenth)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

A(parent, parent2);

if (Lex.enumPToken == TToken.lxmRightParenth)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmls)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

B(parent, parent2);

if (Lex.enumPToken == TToken.lxmrs)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmRightParenth)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

uSemantAnalyzer uSemant = new uSemantAnalyzer(tree);

Lex.NextToken();

}

else throw new Exception("Ожидалась )");

}

else throw new Exception("Ожидалась ]");

}

else throw new Exception("Ожидалась [");

}

else throw new Exception("Ожидалась )");

}

else throw new Exception("Ожидалась (");

}

else throw new Exception("Ожидалось (");

}

public void B(TreeNode highParent, TreeNode highParent2)

{

TreeNode parent = new TreeNode("B");

highParent.Nodes.Add(parent);

TreeNode parent2 = new TreeNode("B");

highParent2.Nodes.Add(parent2);

if (Lex.enumPToken == TToken.lxmNumber)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

// сын (типо под parents) (у него 2 родителя)

Lex.NextToken();

}

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

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmls)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

B(parent, parent2);

if (Lex.enumPToken == TToken.lxmrs)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmLeftParenth)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

A(parent, parent2);

if (Lex.enumPToken == TToken.lxmRightParenth)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmrs)

{

parent.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

parent2.Nodes.Add(new TreeNode(Lex.strPLexicalUnit));

//uSemantAnalyzer2 uSemant2 = new uSemantAnalyzer2(tree);

Lex.NextToken();

}

else throw new Exception("Ожидалась ]");

}

else throw new Exception("Ожидалась )");

}

else throw new Exception("Ожидалась (");

}

else throw new Exception("Ожидалась ]");

}

else throw new Exception("Ожидалась [");

}

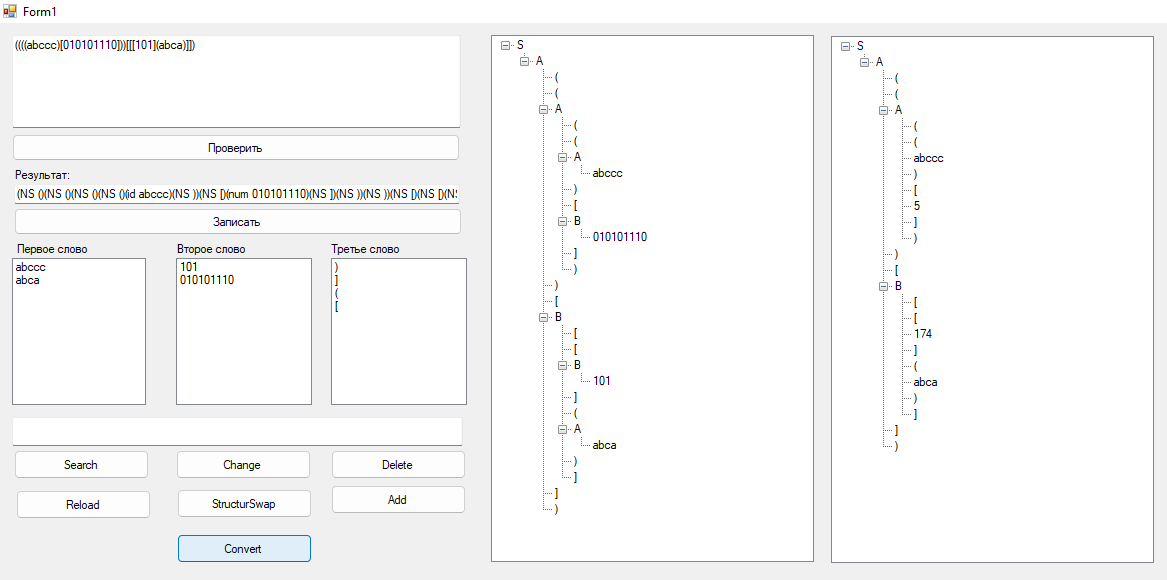
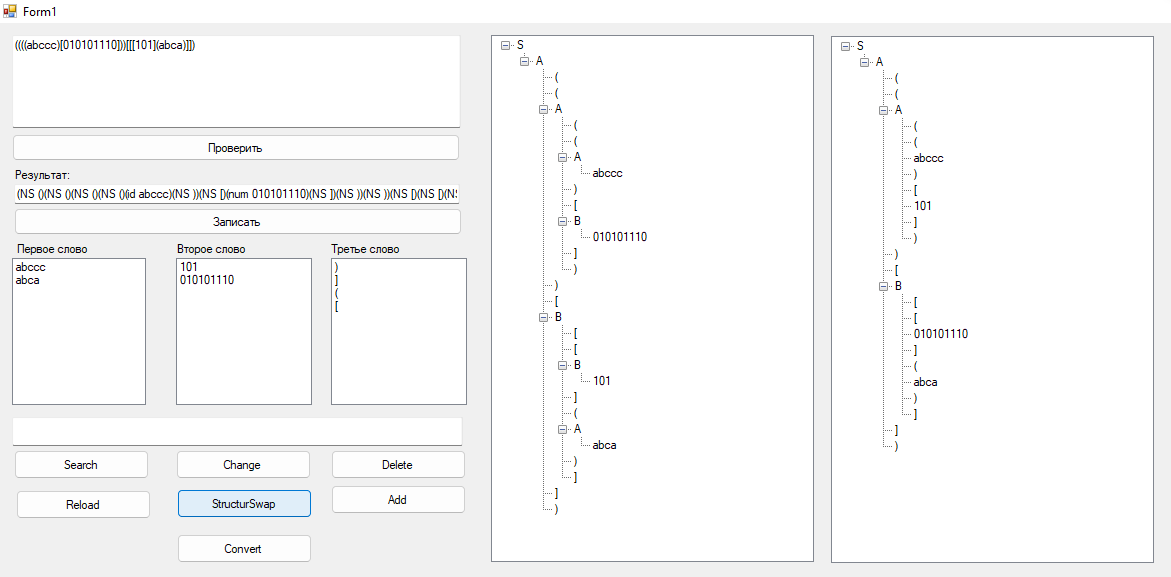
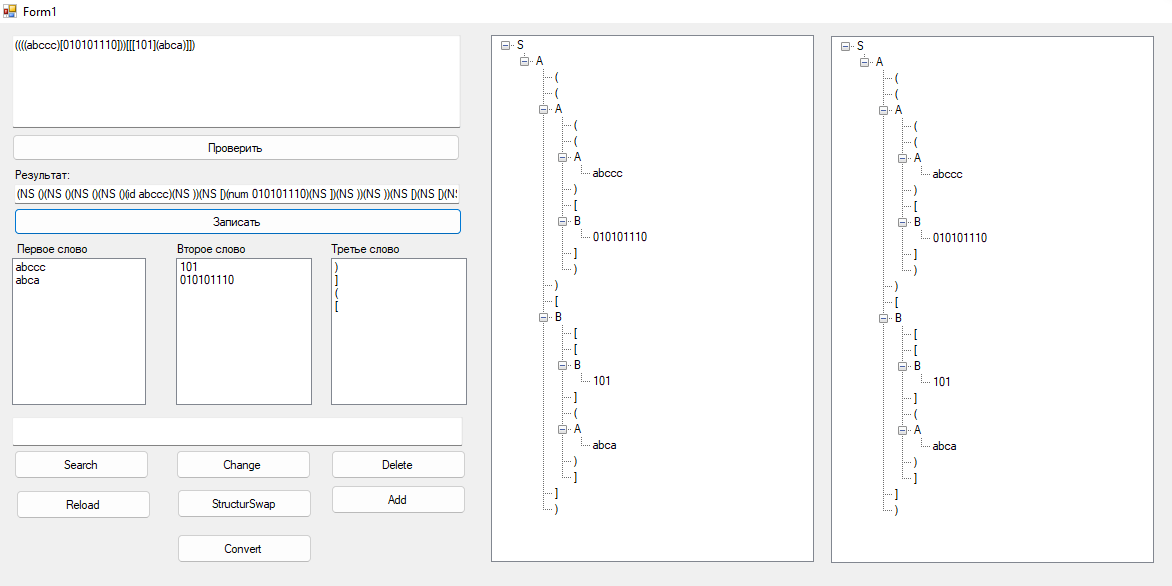
else throw new Exception("Ожидалось [");

}

}

}

Результат программы:



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

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

**Чтобы генератор считался полным, ему, возможно, не хватает следующих характеристик:**

1. **Итерабельность и Итератор**:

Полноценные генераторы в языках программирования, таких как Python, обычно предоставляют интерфейс итератора и могут быть использованы в циклах for. Это делает их более удобными для работы с последовательностями данных.

1. **Механизм передачи значений:**

В контексте генераторов данных, возможно, будет полезен механизм передачи значений от генератора к потребителю. В Python, это реализовано с использованием ключевого слова yield.

1. **Гибкость и Контроль:**

В случае генераторов кода, полноценный генератор может предоставлять гибкие опции конфигурации и контроля над создаваемым кодом.

1. **Поддержка различных операций:**

В некоторых случаях генератор может считаться полным, если он поддерживает различные операции, такие как фильтрация, преобразование или комбинирование данных в удобной форме.

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.

|  |  |
| --- | --- |
| 20 | Все числа должны быть разными |

Текст программы:

**Duplicate.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

using System.Xml.Linq;

namespace WindowsFormsApp2

{

public class Duplicate

{

//MyHashFunction myHashFunction = new MyHashFunction();

public CHashTableList Htl2;

Dictionary<string, int> TableIdentifier = new Dictionary<string, int>();

private TreeView tree;

CLex Lex = new CLex();

public Duplicate(TreeView treeView, CHashTableList htl2)

{

Htl2 = htl2;

tree = treeView;

TableIdentifier = new Dictionary<string, int>();

}

public void ChekingForDuplicate(TreeView tree)

{

foreach (TreeNode node in tree.Nodes) // Рассматриваем всевозможные ветви (node) нашего TreeView

{

ChekingForDuplicate(node);

}

foreach (var item in TableIdentifier)

{

if (item.Value > 1)

{

tree.Nodes.Clear();

throw new Exception(item.Key + " повторяется " + item.Value + ". Создайте новое дерево.");

}

}

}

public void ChekingForDuplicate(TreeNode node)

{

int temp;

if (node.Nodes.Count > 0) // Если эта ветка не пустая

{

foreach (TreeNode node2 in node.Nodes) // Перебираем все подветви нода с названием B, тем самым рассматривая каждый лексический юнит

{

if (int.TryParse(node2.Text, out temp))

{

int b = 0;

/\* if (Htl2.SearchLexicalUnit(node2.Text, 1, ref b))

{

tree.Nodes.Clear();

throw new Exception(node2.Text + " повторяется. Создайте новое дерево.");

//myHashFunction.AddWord(hashTableIdentifier, binaryValue);

}

else

{\*/

if (TableIdentifier.ContainsKey(node2.Text))

{

TableIdentifier[node2.Text] += 1;

}

else

{

TableIdentifier.Add(node2.Text, 1);

}

//Htl2.AddLexicalUnit(node2.Text, 1, ref b);

//}

}

}

}

foreach (TreeNode childNode in node.Nodes)

{

ChekingForDuplicate(childNode); // Тут мы разделяем опять ветвь (node) на ещё более мелкие, то есть углубляемся

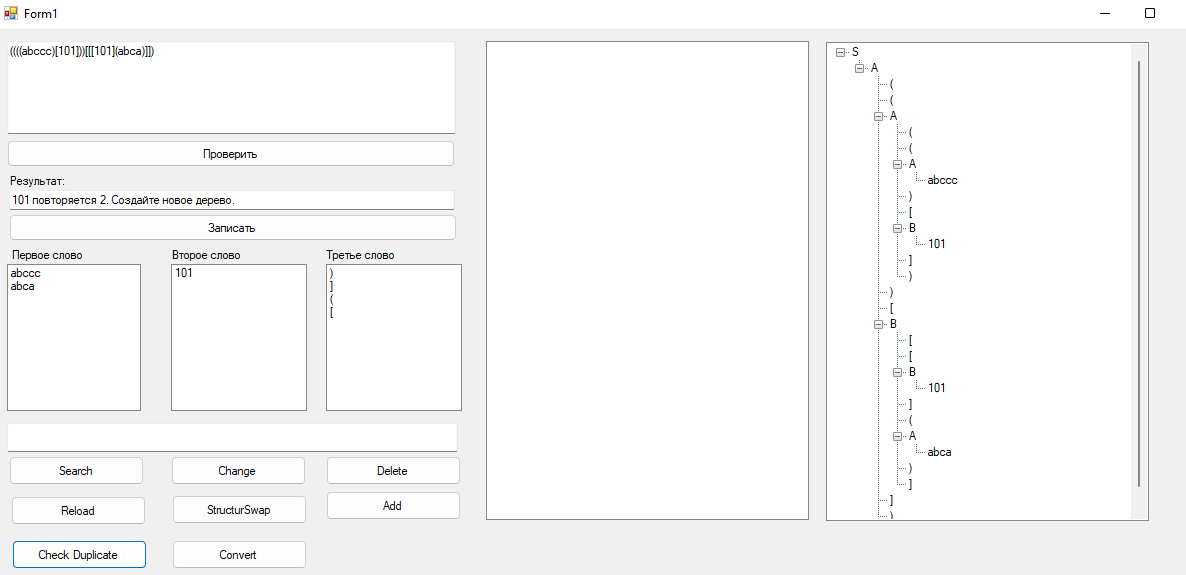
}

}

}

}

Результат программы:



**Лабораторная работа №8**

**Задание:**

Разработать семантический анализатор. Выполнить проверку внеконтекстной грамматики.

Вариант: Определения зависимостей между идентификаторами. Проверка того, что длинна идентификатора слова <1> будет меньше десятичного представления идентификатора числа <2>, а также наличие идентификаторе слова не менее 2 буква ‘a’.

**Краткое теоретическое обоснование:**

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.

Основные семантические ошибки:

* Несоответствие типов;
* Необъявленная переменная
* Использование зарезервированного идентификатора
* Многократное объявление переменной в области видимости
* Доступ к переменной вне области
* Фактическое и формальное несоответствие параметров

Чтобы помочь в своей работе, семантический анализатор обычно строит и поддерживает структуру данных *таблицы символов*, которая сопоставляет каждый идентификатор с информацией, известной о нем. Помимо прочего, эта информация включает в себя тип идентификатора, внутреннюю структуру (если таковая имеется) и область действия (часть программы, в которой он действителен).

Используя таблицу символов, семантический анализатор применяет большое разнообразие правил, которые не улавливаются иерархической структурой контекстно-свободной грамматики и дерева синтаксического анализа. Например, в C он проверяет, что

■ Каждый идентификатор объявляется перед его использованием.

■ Идентификатор не используется в неподходящем контексте (вызов целого числа в качестве подпрограммы, добавление строки к целому числу, ссылка на поле неправильного типа структуры и т. Д.).

■ Вызовы подпрограмм обеспечивают правильное количество и типы аргументов.

■ Метки на плечах оператора switch являются различными константами.

■ Любая функция с типом возврата non-void возвращает значение явно.

Во многих компиляторах работа семантического анализатора принимает форму *семантических подпрограмм действий*, вызываемых синтаксическим анализатором, когда он понимает, что достиг определенной точки в пределах грамматического правила.

Семантический анализаториспользует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице символов для последующего использования в процессе генерации промежуточного кода.

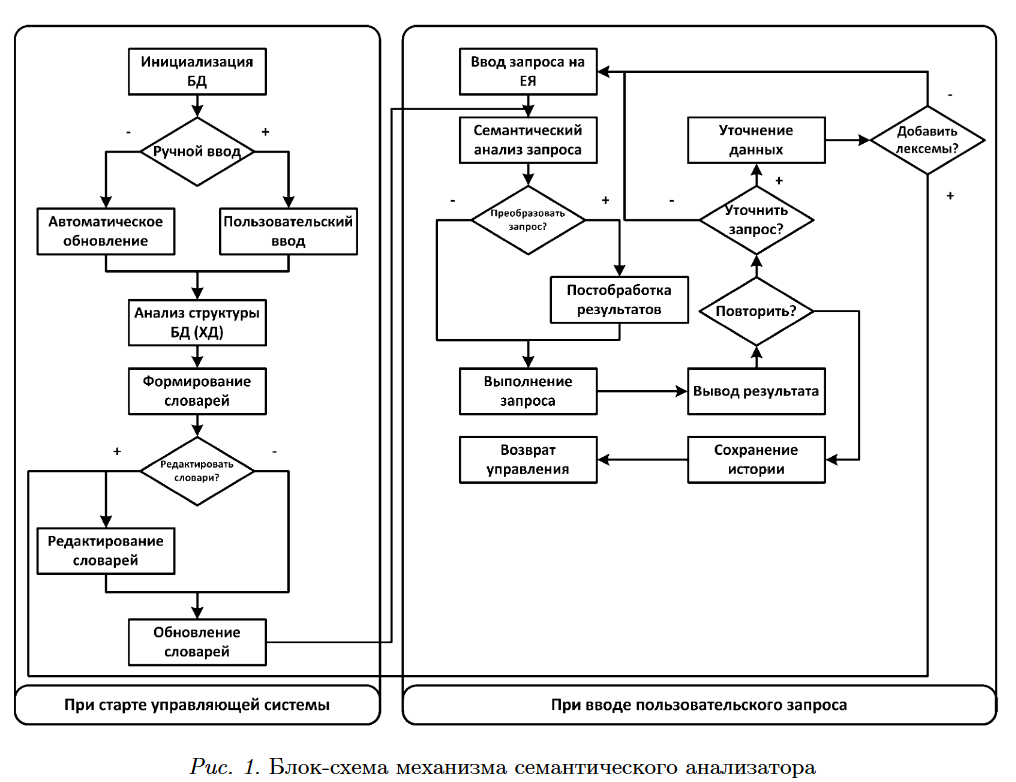
Важной частью семантического анализа является *проверка типов*, когда компилятор проверяет, имеет ли каждый оператор операнды соответствующего типа. Например, многие определения языков программирования требуют, чтобы индекс массива был целым числом; компилятор должен сообщить об ошибке, если в качестве индекса массива используется число с плавающей точкой.

Рассмотрим пример, описание типа переменной в Си вида int a,b=5,c[10] синтаксически реализуется группой правил, задающих цикл описания отдельных переменных, а общий тип и полученный список определяется отдельным правилом. Но между ними имеется связь, состоящая в том, что семантика типа int распространяется на все элементы списка. Отсюда имеем различные варианты реализации семантических процедур:

* если синтаксическая единица реализуется в одном правиле, то ее семантическая обработка., а также генерация кода или интерпретация могут быть выполнены независимой семантической процедурой. Взаимодействие с другими процедурами происходит по указанной выше схеме: процедура использует ссылки на семантику символов правой части (потомков поддерева для этого правила), формирует и возвращает ссылку на семантику левой части (корневой вершины поддерева). То же самое можно сказать и способе передачи сгенерированного кода или результата интерпретации;
* если синтаксическая единица реализуется несколькими правилами, то такая стройная картина нарушается. Например, может потребоваться более сложное взаимодействие семантических процедур (не только вверх-вниз по дереву, но и между смежными вершинами и т. п.). Для передачи результатов между процедурами могут использоваться не только ссылки на записи в семантических таблицах, но и более сложные структуры данных.

В приведенном выше примере правила, соединяющее общий тип описания (int) со списком переменных, имеет вид X::=TL и получает от первого нетерминала ссылку в семантической таблице типов, а от второго нетерминала множество ссылок (массив, список) на записи в семантической таблице переменных. Семантическая процедура правила должна дополнить все цепочки описаний типа данных в этом множестве записей указанным общим типов (причем по семантике определения типа данных в Си он должен быть дописан в конец цепочки).

Рассмотрим алгоритм функционирования предлагаемого механизма семантического анализатора в виде блок-схемы (рис. 1).



Прокомментируем работу алгоритма.

1. При инициализации система анализирует БД или ХД и на основе анализа описания данных, представленного при помощи XML-файла либо встроенного механизма платформы АИС, строит структуру используемых объектов.  
2. На основе построенной структуры объектов и их связей формируется проблемно-ориентированное подмножество ЕЯ.

3. Удаленный пользователь или оператор консоли (далее по тексту используется термин «оператор терминала» – ОТ) вводит запрос на ЕЯ на основе заполненных словарей.

4. Используя механизм семантического анализатора выполняется анализ введенного ОТ запроса на ЕЯ и строится соответствующий ему запрос на языке 1CQ.

5. Если АИС не поддерживает язык запросов 1CQ, то запрос конвертируется в стандарт языка SQL (переводится на английский язык, корректируется синтаксис и т. п.)

6. Имея в распоряжении код на языках запросов 1CQ или SQL – машинный запрос, платформа АИС выполняет его и выдает результат ОТ.

Пример: выберем оперативный учет успеваемости студентов  
Санкт-Петербургского государственного университета водных коммуникаций

**Код программы:**

**uSemantAnalyzer.cs**

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp2

{

public class uSemantAnalyzer

{

public int i = 0;

public string strIndentifier;

public string strDigital;

private TreeView tree;

public uSemantAnalyzer()

{

}

public uSemantAnalyzer(TreeView treeView)

{

tree = treeView;

TreeController(tree);

}

public void TreeController(TreeView tree)

{

foreach (TreeNode node in tree.Nodes)

{

TreeController(node);

}

}

public void TreeController(TreeNode node)

{

if (node.Text == "B")

{

if (node.Nodes.Count > 1)

{

strIndentifier = node.Nodes[5].Nodes[0].Text.ToString();

strDigital = node.Nodes[2].Nodes[0].Text.ToString();

Check(strIndentifier, strDigital, node);

}

}

foreach (TreeNode childNode in node.Nodes)

{

TreeController(childNode);

}

}

private void Check(string ident, string digit, TreeNode node)

{

int digitInt = Convert.ToInt32(digit,2);

int identLen = ident.Length;

if (ident.Count(x => x == 'a') < 2 && digitInt < identLen)

{

tree.SelectedNode = node;

tree.SelectedNode.BackColor = Color.Red;

throw new Exception("Errors semant");

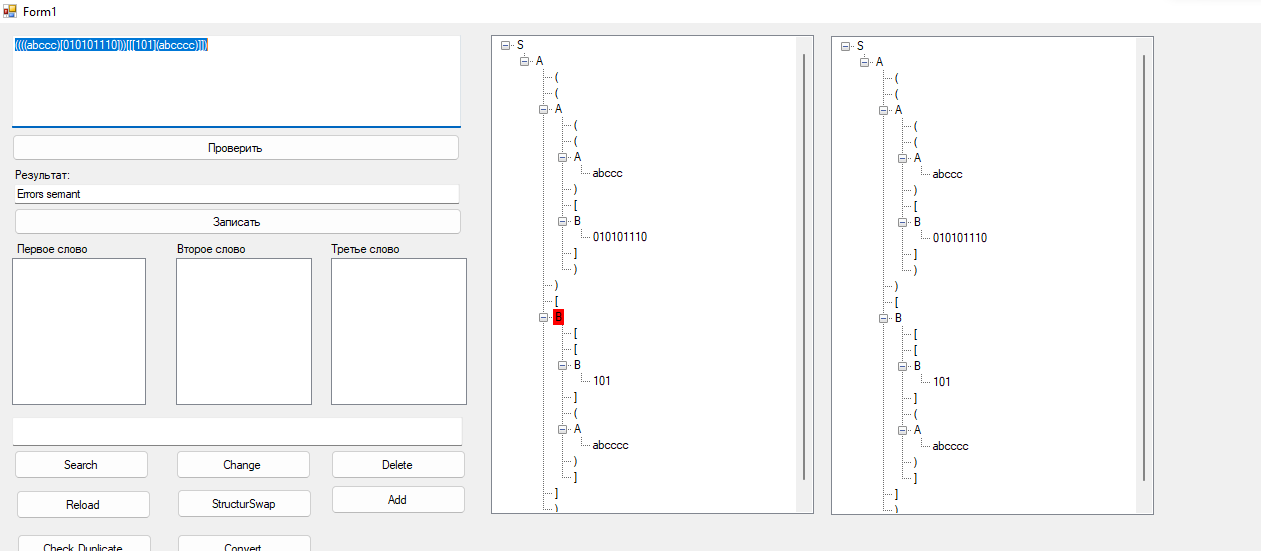
}

}

}

}

**Результат программы:**

****