Министерство образования Российской Федерации

Федеральное агентство по образованию

КАЗАНСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

им. А.Н. Туполева - КАИ

Кафедра АСОИУ

Лабораторные работы №1-8

по дисциплине

«ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ» и «Методы трансляции»

Выполнил:

Студент группы 4309

Мифтахутдинов Р. Р.

Проверила: Бикмуллина И. И.

Казань 2023

**Лабораторная работа № 1. Разработка транслитератора**

**Текст задания:**

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

**Краткое теоретическое обоснование:**

Транслитератор демонстрационного языка программирования используется для выделения следующих классов отдельных символов:

* Класс букв: содержит прописные и строчные буквы латинского алфавита, используемые при создании разнообразных конструкций языка. Русские буквы в этот класс не включаются, так как используются только внутри строк и комментариев, допускающих почти все символы.
* Класс десятичных цифр: объединяет арабские цифры от 0 до 9. Используется при формировании описаний действительных, а также некоторых из целых чисел.
* Класс двоичных цифр: объединяет цифры 0 и 1. Используется при анализе целых двоичных чисел.
* Класс восьмеричных цифр: объединяет цифры от 0 до 7. Используется при анализе целых восьмеричных чисел.
* Класс шестнадцатеричных цифр: включает цифры от 0 до 9, а также прописные и строчные буквы: A, B, C, D, E, F, a, b, c, d, e f.
* Класс пропусков: состоит из пробела, перевода строки, табуляции, перевода формата (разделяющего текст на отдельные страницы). Символы этого класса используются для разделения различных элементарных конструкций, слитное написание которых привело бы к неправильному восприятию (например, следующие друг за другом число и идентификатор "123E4 asdf" без пробела были бы восприняты как "123E4asdf", что является ошибкой).
* Класс игнорируемых символов: включает все символы, которые, как предполагается, не отображаются на экране текстового редактора. В используемых кодовых таблицах к ним относятся символы, коды которых меньше кода пробела. Исключение составляют перевод строки, табуляция, перевод формата, уже отнесенные к предыдущему классу. В некоторых текстовых редакторов данные символы отображаются в виде специальных значков. Поэтому, выделение данного класса может являться спорным и зависит от различных факторов.
* Класс прочих символов: включает все оставшиеся символы. Несмотря на то, что их тоже можно группировать в различные классы, в рассматриваемом языке нам, в большинстве ситуаций, достаточно использовать их непосредственные значения

Следует отметить, что классы символов пресекаются. Однако, вопрос принадлежности нужному классу можно решать, основываясь на текущем контексте. Класс символов можно специально не хранить, а проверять тогда, когда потребуется.

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

Основные функции лексического анализатора:

1. Чтение с помощью транслитератора очередного слова в исходном тексте и его классификация;

2. Пропуск пробелов и комментариев;

3. Выдача диагностических сообщений об обнаруженных лексических ошибках.

**Код программы:**

**Form1.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp79

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

tbFSource.AppendText("01ab" + "\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.BracketOpen: { s = "KT"; s1 = "BracketOpen"; break; }

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

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

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

case TCharType.Semicolon: { s = "KT"; s1 = "Semicolon"; 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 WindowsFormsApp79

{

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

public enum TCharType { Letter, Digit, EndRow, EndText, Space, ReservedSymbol, BracketOpen, BracketClosed, ExclamationMark, Comma, Semicolon }; // 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

{

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.BracketOpen;

else if (chrFSelection == ')')

enumFSelectionCharType = TCharType.BracketClosed;

else if (chrFSelection == ',')

enumFSelectionCharType = TCharType.Comma;

else if (chrFSelection == '!')

enumFSelectionCharType = TCharType.ExclamationMark;

else if (chrFSelection == ';')

enumFSelectionCharType = TCharType.Semicolon;

else if (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();

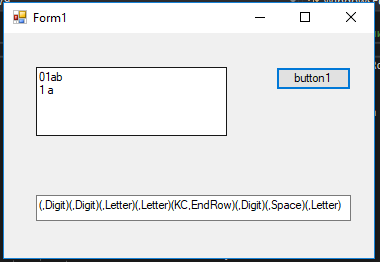
}

}

}

}

**Результаты тестирования:**



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

**Текст задания:**

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

|  |  |  |
| --- | --- | --- |
| (010)\*100(00)\* | (a|b|c|d)+ | Вторые два символа всегда cd |

**Недетерминированная матрица:**

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

**Граф:**

1

A

0

C

0

0

B

0

E

D

0

F

0

0

0

G

0

0

Fin

0

H

**Детерминированная матрица:**

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

**Детерминированная матрица:**

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

**Второе слово:**

(a|b|c|d)+

Вторые 2 символа всегда cd

**Недетерминированная матрица:**

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

**Граф:**

1

A

0

C

0

0

B

0

E

D

0

FFin

0

0

G

0

0

H

**Граф:**

A

B

d

c

d

b

a

C

b

a

c

Fin

**Краткое теоретическое обоснование:**

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

Основные функции лексического анализатора:

1. Чтение с помощью транслитератора очередного слова в исходном тексте и его классификация;

2. Пропуск пробелов и комментариев;

3. Выдача диагностических сообщений об обнаруженных лексических ошибках.

Грамматикой Хомского типа 3, или регулярной грамматикой, называется контекстно-свободная-грамматика, все правила которой односторонне линейны. Это означает, что либо они все праволинейны ,т.е имеют вид A-> Ba или A -> а; либо леволинейны , т.е имеют вид А→aВ или A -> а. Здесь А и В и обозначают нетерминальные символы, а через а обозначен терминальный символ.

Способы описания конечного автомата:

1. Множество команд. Нетерминальным символам грамматики сопоставим состояния автомата, терминальным символам грамматики сопоставим символы входной ленты. Правилу вида A → aB сопоставим команду автомата (A,a) →B; правилу вида A → a (соответствует последнему символу в слове) сопоставим команду автомата (A,a) →Fin, где Fin - обозначение финального состояния. Для рассмотренного примера команды будут выглядеть следующим образом (Z,0) →U, (Z,1) →V, (U,1) →Z, (U,1) →Fin, (V,0) →Z, (V,0) →Fin.
2. Графический способ. Состояния автомата будут представлять собой узлы диаграммы состояний и переходов, а символы входной ленты будут помечать стрелку, соединяющую состояния. Каждому правилу вида A → aB сопоставим следующий фрагмент диаграммы состояний

**а**

Каждому правилу вида A → a сопоставим фрагмент диаграммы состояний вида:

**а**

1. Матричный способ. Строки матрицы помечаются состояниями (нетерминальным алфавитом), а столбцы – терминальным алфавитом. В клетки матрицы заносятся состояния переходов.

**Код программы:**

**Form1.cs:**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp65

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

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

tbFSource.AppendText("");

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

{

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

while (Lex.enumPState != TState.Finish)

{

Lex.NextToken();

String s = "";

String s1 = "";

switch (Lex.enumPToken)

{

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

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

}

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

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

}

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

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

n += Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

}

}

**uLex.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace Class1

{

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

public enum TCharType { Letter, Digit, EndRow, EndText, Space, ReservedSymbol, ReservedSymbol2, ReservedSymbol3, UnknownNumbers }; // тип символа

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

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 = -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 == ':' || chrFSelection == '-' || chrFSelection == ',' || chrFSelection == '.' || chrFSelection == '=' || chrFSelection == '[' || chrFSelection == ']' || chrFSelection == ' ') enumFSelectionCharType = TCharType.ReservedSymbol;

else {

enumFSelectionCharType = TCharType.UnknownNumbers;

//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();

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

//Вариант 17

switch (enumFSelectionCharType)

{

case TCharType.Letter:

{

// a b c d

// A | | | B | |

// B | | | |CFin|

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

A:

{

if (chrFSelection == 'c')

{

TakeSymbol();

goto B;

}

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

}

B:

{

if (chrFSelection == 'd')

{

TakeSymbol();

goto CFin;

}

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

}

CFin:

{

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

{

TakeSymbol();

goto CFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

case TCharType.Digit:

{

// 0 1

// A | 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 == '0')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидался 0");

E:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 0");

FFin:

if (chrFSelection == '0')

{

TakeSymbol();

goto G;

}

else if (enumFSelectionCharType != TCharType.Digit)

{

enumFToken = TToken.lxmNumber;

return;

}

else throw new Exception("Ожидалась 0");

G:

if (chrFSelection == '0')

{

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;

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 == ' ')

{

enumFToken = TToken.IsEndOfText;

GetSymbol();

return;

}

break;

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

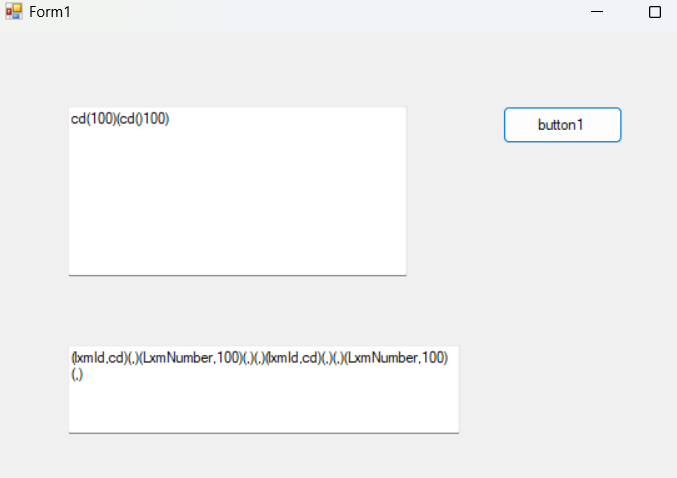
}

}

}

}

**Результаты тестирования:**



**Лабораторная работа № 3. Разработка транслитератора**

**Текст задания:**

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

Примечание: здесь и далее через <1> и <2> обозначены слова из лабораторной работы №1.

Указания:

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

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

3. Если грамматика леворекурсивная, то устранить левую рекурсию.

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

**Вариант 17-p:**

S 🡪 <2> SA

S 🡪 A

A 🡪 (S)

A 🡪 ()

A 🡪 <1>

Избавление от левой рекурсии не будет, так как левой рекурсии нет.

1: A 🡪 (S)()<1>

2: S 🡪<2>

3: S 🡪 <2> 🡪 SA🡪 <2> 🡪 SA 🡪 <2>AA 🡪 <2><1><1>

**Краткое теоретическое обоснование:**

Разработка контекстно-свободного (КС) синтаксического анализатора — это процесс создания программного инструмента, способного анализировать структуру текстового или программного кода на основе контекстно-свободной грамматики. Контекстно-свободная грамматика (КС-грамматика) — это формальное описание языка, которое определяет его синтаксическую структуру в виде набора правил.

Важные компоненты разработки КС-синтаксического анализатора включают в себя следующие этапы и концепции:

* Грамматика: Разработка КС-синтаксического анализатора начинается с определения КС-грамматики для языка, который вы хотите анализировать. Грамматика включает в себя терминальные и нетерминальные символы, правила продукции, стартовый символ и другие сведения о структуре языка.
* Терминальные и нетерминальные символы: Терминальные символы представляют лексемы (токены) языка, такие как ключевые слова, операторы и идентификаторы. Нетерминальные символы представляют собой абстрактные конструкции языка, которые могут быть разложены на более простые элементы с помощью правил продукции.
* Правила продукции: Правила продукции определяют, как нетерминальные символы могут быть заменены другими символами (терминальными или нетерминальными) в контексте грамматики. Эти правила определяют синтаксическую структуру языка.
* Стартовый символ: Стартовый символ — это нетерминальный символ, с которого начинается синтаксический анализ текста. Обычно это самый верхний уровень абстракции в грамматике.
* Токенизация: прежде чем начать синтаксический анализ, текст обычно проходит процесс токенизации, в ходе которого он разбивается на отдельные лексемы (токены), например, разделяя ключевые слова и операторы.
* Алгоритмы синтаксического анализа: существует несколько методов синтаксического анализа для обработки текста на основе КС-грамматики. Наиболее распространенными методами являются LL(к) и LR(к) анализ. Каждый из них имеет свои преимущества и недостатки, и выбор зависит от конкретных требований проекта.
* Генерация синтаксического анализатора: Синтаксический анализатор может быть создан вручную, но также существуют инструменты, способные автоматически генерировать анализаторы на основе заданной КС-грамматики, такие как YACC, Bison, ANTLR и другие.
* Обработка синтаксического дерева: после успешного синтаксического анализа текста обычно строится синтаксическое дерево, которое представляет собой структуру данных, отражающую иерархическую структуру текста. Это дерево затем может быть использовано для выполнения дополнительных анализов или для генерации кода.

Разработка контекстно-свободного (КС) синтаксического анализатора — это сложный процесс, который включает в себя несколько этапов и требует тщательного планирования и проектирования. Давайте более подробно рассмотрим некоторые из ключевых аспектов этого процесса:

* Определение целей и требований: прежде чем начать разработку КС-синтаксического анализатора, необходимо четко определить его цели и требования. Это включает в себя понимание языка, который анализатор будет обрабатывать, и конечных целей, таких как компиляция кода, интерпретация или структурный анализ.
* Выбор языка программирования: Выбор языка программирования для реализации синтаксического анализатора имеет значение. Часто используются языки с богатой поддержкой работы со строками и регулярными выражениями, такие как Python, Java, C++ или другие.
* Определение КС-грамматики: Создайте КС-грамматику, которая описывает синтаксис целевого языка. Грамматика должна быть формально определена и документирована. Можно использовать формальные спецификации, такие как Backus-Naur Form (BNF) или Extended Backus-Naur Form (EBNF), для представления грамматики.
* Разработка лексического анализатора: Часто КС-синтаксический анализатор включает в себя и лексический анализатор, который разбивает входной текст на лексемы (токены). Лексический анализатор может быть разработан отдельно или встроен в синтаксический анализатор.
* Выбор метода синтаксического анализа: Выберите подходящий метод синтаксического анализа, такой как LL(к), LR(к), или другие, в зависимости от сложности грамматики и требований к анализу. Этот выбор может потребовать тщательного анализа грамматики и оценки производительности анализатора.
* Разработка и отладка анализатора: на этом этапе происходит фактическая реализация синтаксического анализатора с использованием выбранного метода анализа. Здесь также проводятся тесты и отладка для обеспечения корректной работы.
* Обработка ошибок: Разработайте механизм обработки синтаксических ошибок, чтобы предоставить информативные сообщения об ошибках во входном тексте.
* Интеграция с другими компонентами: если синтаксический анализатор является частью компилятора или интерпретатора, необходимо интегрировать его с другими компонентами, такими как лексический анализатор, семантический анализ и генератор кода.
* Оптимизация и улучшение производительности: Проведите профилирование и оптимизацию синтаксического анализатора, чтобы улучшить его производительность и эффективность.
* Документация и тестирование: Документируйте код синтаксического анализатора и проведите обширное тестирование, включая функциональное тестирование и тестирование производительности.

**Код программы:**

**Form1.cs**

using Class1;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

namespace WindowsFormsApp5

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

tbFSource.AppendText("cd(100)(cd()100)");

int n = tbFSource.Lines.Length;

}

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

{

tbFMessage.Clear();

uSyntAnalyzer Synt = new uSyntAnalyzer();

Synt.Lex.strPSource = tbFSource.Lines;

Synt.Lex.strPMessage = tbFMessage.Lines;

Synt.Lex.enumPState = TState.Start;

try

{

Synt.Lex.NextToken();

Synt.S();

throw new Exception("Текст верный");

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

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

n += Synt.Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

}

}

**uLex.cs**

**using System;**

**using System.Collections.Generic;**

**using System.Text;**

**namespace Class1**

**{**

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

**public enum TCharType { Letter, Digit, EndRow, EndText, Space, ReservedSymbol, ReservedSymbol2, ReservedSymbol3, UnknownNumbers }; // тип символа**

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

**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 = -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 == ':' || chrFSelection == '-' || chrFSelection == ',' || chrFSelection == '.' || chrFSelection == '=' || chrFSelection == '[' || chrFSelection == ']' || chrFSelection == ' ') enumFSelectionCharType = TCharType.ReservedSymbol;**

**else {**

**enumFSelectionCharType = TCharType.UnknownNumbers;**

**//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();**

**}**

**if (chrFSelection == '/')**

**{**

**GetSymbol();**

**if (chrFSelection == '/')**

**while (enumFSelectionCharType != TCharType.EndRow)**

**{**

**GetSymbol();**

**}**

**GetSymbol();**

**}**

**//Вариант 17**

**switch (enumFSelectionCharType)**

**{**

**case TCharType.Letter:**

**{**

**// a b c d**

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

**// B | | | |CFin|**

**// CFin |CFin|CFin|CFin|CFin|**

**A:**

**{**

**if (chrFSelection == 'c')**

**{**

**TakeSymbol();**

**goto B;**

**}**

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

**}**

**B:**

**{**

**if (chrFSelection == 'd')**

**{**

**TakeSymbol();**

**goto CFin;**

**}**

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

**}**

**CFin:**

**{**

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

**{**

**TakeSymbol();**

**goto CFin;**

**}**

**else**

**{**

**enumFToken = TToken.lxmIdentifier;**

**return;**

**}**

**}**

**}**

**if (chrFSelection == '/')**

**{**

**GetSymbol();**

**if (chrFSelection == '/')**

**while (enumFSelectionCharType != TCharType.EndRow)**

**{**

**GetSymbol();**

**}**

**GetSymbol();**

**}**

**case TCharType.Digit:**

**{**

**// 0 1**

**// A | 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 == '0')**

**{**

**TakeSymbol();**

**goto A;**

**}**

**else throw new Exception("Ожидался 0");**

**E:**

**if (chrFSelection == '0')**

**{**

**TakeSymbol();**

**goto FFin;**

**}**

**else throw new Exception("Ожидалась 0");**

**FFin:**

**if (chrFSelection == '0')**

**{**

**TakeSymbol();**

**goto G;**

**}**

**else if (enumFSelectionCharType != TCharType.Digit)**

**{**

**enumFToken = TToken.lxmNumber;**

**return;**

**}**

**else throw new Exception("Ожидалась 0");**

**G:**

**if (chrFSelection == '0')**

**{**

**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;**

**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 == ' ')**

**{**

**enumFToken = TToken.IsEndOfText;**

**GetSymbol();**

**return;**

**}**

**break;**

**}**

**case TCharType.EndText:**

**{**

**enumFToken = TToken.lxmEmpty;**

**break;**

**}**

**}**

**}**

**}**

**}**

**uSyntAnalyzer.cs:**

using Class1;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApp5

{

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 int indexLxmLeftParenth = 0;

public int indexLxmRightParenth = 0;

public void S()

{

Console.WriteLine("S " + Lex.enumPToken);

if (Lex.enumPToken == TToken.lxmIdentifier)

{

Lex.NextToken();

S();

}

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

{

Lex.NextToken();

A();

}

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

{

throw new Exception("Текст верный. Конец текста.");

}

else throw new Exception("Текст не верный");

}

public void A()

{

Console.WriteLine("A");

if (Lex.enumPToken == TToken.lxmNumber)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmRightParenth)

{

Lex.NextToken();

S();

}

else throw new Exception("Ожидалась )");

}

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

{

Lex.NextToken();

S();

if (Lex.enumPToken == TToken.lxmRightParenth) Lex.NextToken();

}

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

{

Lex.NextToken();

S();

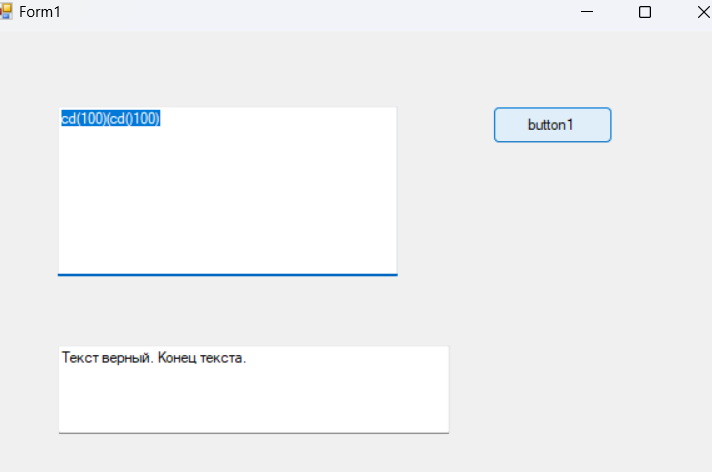
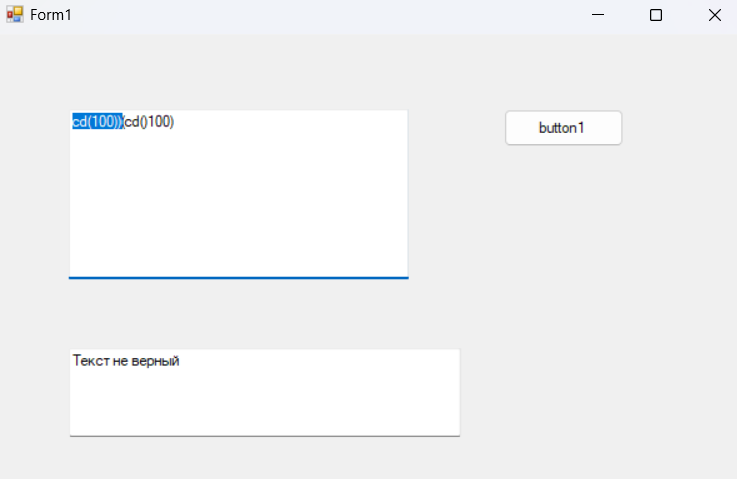
}

else throw new Exception("Ожидался <1> или <2>" + Lex.enumPToken);

}

}

}

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

**Задание:**

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.

**Краткое теоретическое обоснование:**

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

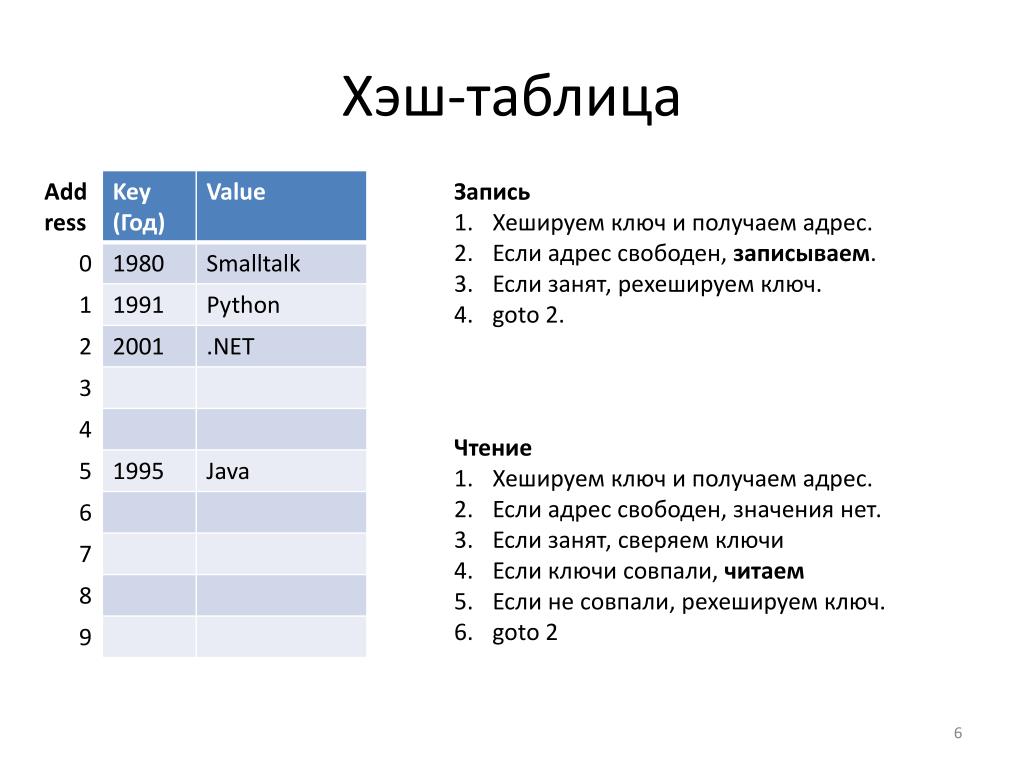
Хеширование — это процесс применения хеш-функции к некоторым данным для получения уникального значения (хеш-кода или хеша), которое служит идентификатором для этих данных. Цель хеширования заключается в том, чтобы ускорить поиск элементов в коллекциях данных, таких как хеш-таблицы, используя хеш-код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

Синтаксический анализатор (или парсер) — это компонент компилятора или интерпретатора, который выполняет анализ входного текста или программного кода с целью определения его синтаксической структуры в соответствии с заданной грамматикой. Он разбирает текст на более абстрактные структуры данных, такие как синтаксические деревья или абстрактные синтаксические деревья (AST), которые представляют собой иерархический способ представления структуры кода. Синтаксический анализатор проверяет, соответствует ли входной код грамматике языка и генерирует ошибки в случае нарушения синтаксических правил. После успешного анализа синтаксический анализатор передает абстрактное представление кода другим компонентам, таким как семантический анализатор и генератор кода, для дальнейшей обработки.

Разработка синтаксического анализатора — это важный этап создания компиляторов, интерпретаторов и других инструментов обработки языка. Ниже приведены шаги, которые обычно включаются в процесс разработки синтаксического анализатора:

1. Определение грамматики;
2. Выбор метода синтаксического анализа;
3. Разработка синтаксического анализатора;
4. Тестирование;
5. Интеграция;
6. Оптимизация;
7. Документация;
8. Обслуживание;

Пример хэш-таблицы:



**Код программы:**

**Form1.cs:**

using Class1;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

namespace WindowsFormsApp5

{

public partial class Form1 : Form

{

Dictionary<int, List<string>> hashTableIdentifier = new Dictionary<int, List<string>>();

Dictionary<int, List<string>> hashTableDigital = new Dictionary<int, List<string>>();

Dictionary<int, List<string>> hashTableRezerv = new Dictionary<int, List<string>>();

public MyHashFunction hashFunction = new MyHashFunction();

public Form1()

{

InitializeComponent();

tbFSource.AppendText("cd(100)(cd()100)");

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();

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

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

foreach (var entry in hashTableIdentifier)

{

listBox1.Items.Add(string.Join(", ", entry.Value));

}

listTable.Clear();

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

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

foreach (var entry in hashTableDigital)

{

listBox2.Items.Add(string.Join(", ", entry.Value));

}

listTable.Clear();

foreach (var entry in hashTableRezerv)

{

listBox3.Items.Add(string.Join(", ", entry.Value));

}

listTable.Clear();

}

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

{

tbFMessage.Clear();

uSyntAnalyzer Synt = new uSyntAnalyzer();

Synt.Lex.strPSource = tbFSource.Lines;

Synt.Lex.strPMessage = tbFMessage.Lines;

Synt.Lex.enumPState = TState.Start;

try

{

Synt.Lex.NextToken();

Synt.S();

throw new Exception("Текст верный");

}

catch (Exception exc)

{

tbFMessage.Text += exc.Message;

tbFSource.Select();

tbFSource.SelectionStart = 0;

int n = 0;

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

n += Synt.Lex.intPSourceColSelection;

tbFSource.SelectionLength = n;

}

}

private void label1\_Click(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:

{

hashFunction.AddWord(hashTableIdentifier, Lex.strPLexicalUnit);

s1 = "id " + Lex.strPLexicalUnit;

TablesToMemo(this, e);

break;

}

case TToken.lxmNumber:

{

hashFunction.AddWord(hashTableDigital, Lex.strPLexicalUnit);

s1 = "num " + Lex.strPLexicalUnit;

TablesToMemo(this, e);

break;

}

case (TToken.lxmLeftParenth):

{

hashFunction.AddWord(hashTableRezerv, "(");

s1 = "rez " + Lex.strPLexicalUnit;

TablesToMemo(this, e);

break;

}

case (TToken.lxmRightParenth):

{

hashFunction.AddWord(hashTableRezerv, ")");

s1 = "rez " + Lex.strPLexicalUnit;

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 searchBtn\_Click(object sender, EventArgs e)

{

if (hashFunction.SearchWord(hashTableIdentifier, listBox1.SelectedItem.ToString()) == 1)

{

searchBtn.BackColor = Color.Green;

}

else

{

searchBtn.BackColor = Color.Red;

}

}

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

{

hashFunction.AddWord(hashTableIdentifier, textBox1.Text.ToString());

}

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

{

if (hashFunction.RemoveWord(hashTableIdentifier, listBox1.SelectedItem.ToString()))

{

deleteBtn.BackColor = Color.Green;

}

else

{

deleteBtn.BackColor = Color.Red;

}

}

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

{

if (hashFunction.RemoveWord(hashTableIdentifier, listBox1.SelectedItem.ToString()))

{

hashFunction.AddWord(hashTableIdentifier, textBox1.Text.ToString());

changeBtn.BackColor = Color.Green;

}

else

{

changeBtn.BackColor = Color.Red;

}

}

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

{

listBox1.Items.Clear();

listBox2.Items.Clear();

listBox3.Items.Clear();

foreach (var entry in hashTableIdentifier)

{

listBox1.Items.Add(string.Join(", ", entry.Value));

}

foreach (var entry in hashTableDigital)

{

listBox2.Items.Add(string.Join(", ", entry.Value));

}

foreach (var entry in hashTableRezerv)

{

listBox3.Items.Add(string.Join(", ", entry.Value));

}

}

}

}

**uLex.cs:**using System;

using System.Collections.Generic;

using System.Text;

namespace Class1

{

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

public enum TCharType { Letter, Digit, EndRow, EndText, Space, ReservedSymbol, ReservedSymbol2, ReservedSymbol3, UnknownNumbers }; // тип символа

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

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 = -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 == ':' || chrFSelection == '-' || chrFSelection == ',' || chrFSelection == '.' || chrFSelection == '=' || chrFSelection == '[' || chrFSelection == ']') enumFSelectionCharType = TCharType.ReservedSymbol;

else {

enumFSelectionCharType = TCharType.UnknownNumbers;

//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();

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

//Вариант 17

switch (enumFSelectionCharType)

{

case TCharType.Letter:

{

// a b c d

// A | | | B | |

// B | | | |CFin|

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

A:

{

if (chrFSelection == 'c')

{

TakeSymbol();

goto B;

}

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

}

B:

{

if (chrFSelection == 'd')

{

TakeSymbol();

goto CFin;

}

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

}

CFin:

{

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

{

TakeSymbol();

goto CFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

}

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

while (enumFSelectionCharType != TCharType.EndRow)

{

GetSymbol();

}

GetSymbol();

}

case TCharType.Digit:

{

// 0 1

// A | 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 == '0')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидался 0");

E:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 0");

FFin:

if (chrFSelection == '0')

{

TakeSymbol();

goto G;

}

else if (enumFSelectionCharType != TCharType.Digit)

{

enumFToken = TToken.lxmNumber;

return;

}

else throw new Exception("Ожидалась 0");

G:

if (chrFSelection == '0')

{

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;

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;

}

break;

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

}

}

}

}

uSyntAnalyzer.cs:

using Class1;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApp5

{

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 int indexLxmLeftParenth = 0;

public int indexLxmRightParenth = 0;

public void S()

{

Console.WriteLine("S " + Lex.enumPToken);

if (Lex.enumPToken == TToken.lxmIdentifier)

{

Lex.NextToken();

S();

}

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

{

Lex.NextToken();

A();

}

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

{

throw new Exception("Текст верный. Конец текста.");

}

else throw new Exception("Я ебал");

}

public void A()

{

Console.WriteLine("A");

if (Lex.enumPToken == TToken.lxmNumber)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmRightParenth)

{

Lex.NextToken();

S();

}

else throw new Exception("Ожидалась )");

}

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

{

Lex.NextToken();

S();

if (Lex.enumPToken == TToken.lxmRightParenth) Lex.NextToken();

}

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

{

Lex.NextToken();

S();

}

else throw new Exception("Ожидался <1> или <2>" + Lex.enumPToken);

}

}

}

**MyHashFunction.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApp5

{

public class MyHashFunction

{

public int HashFunction(string word)

{

int hashValue = 0;

foreach (char c in word)

{

hashValue += (int)c;

}

return hashValue;

}

public void AddWord(Dictionary<int, List<string>> hashTable, string word)

{

int hashValue = HashFunction(word);

if (!hashTable.ContainsKey(hashValue))

{

hashTable[hashValue] = new List<string>();

}

else

{

if (hashTable[hashValue].Contains(word))

{

Console.WriteLine("Слово уже существует: " + word);

return;

}

}

hashTable[hashValue].Add(word);

}

public int SearchWord(Dictionary<int, List<string>> hashTable, string word)

{

int hashValue = HashFunction(word);

if (hashTable.ContainsKey(hashValue))

{

return 1;

}

else

{

return 0;

}

}

public bool RemoveWord(Dictionary<int, List<string>> hashTable, string word)

{

int hashValue = HashFunction(word);

if (hashTable.ContainsKey(hashValue))

{

List<string> words = hashTable[hashValue];

if (words.Contains(word))

{

words.Remove(word);

if (words.Count == 0)

{

hashTable.Remove(hashValue);

}

return true;

}

}

return false;

}

}

}

THeap.cs:

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp5

{

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)

{

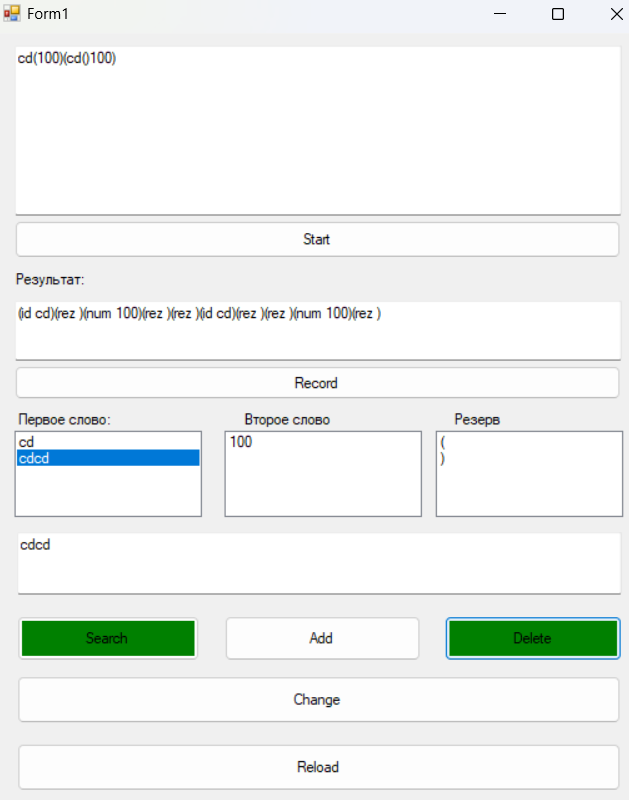
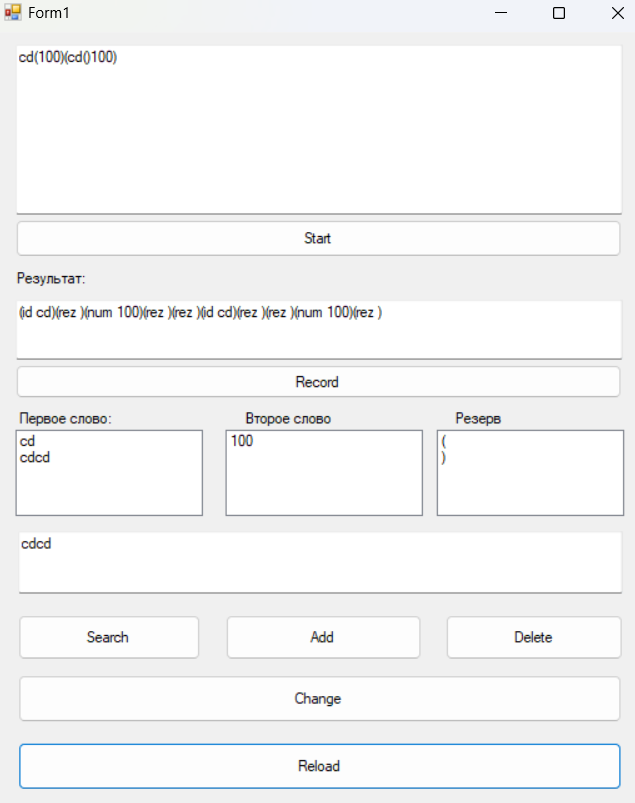
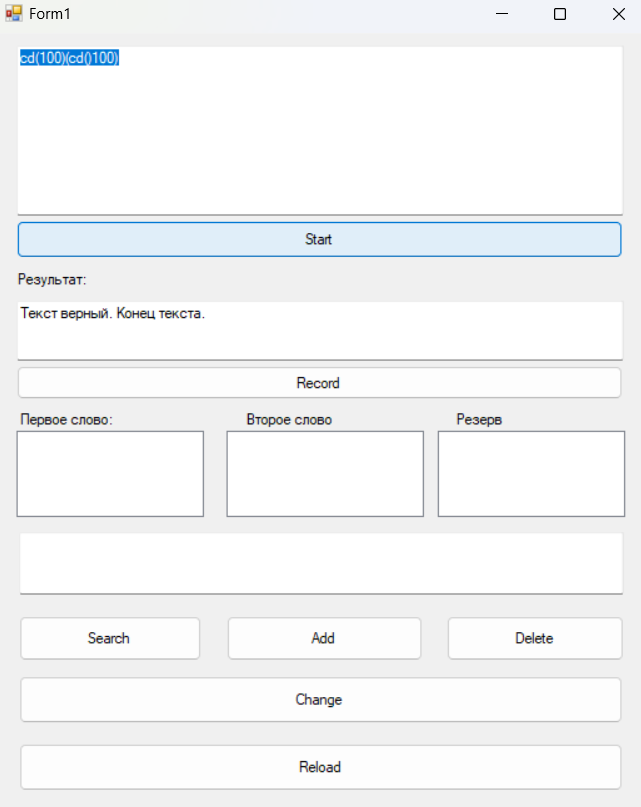
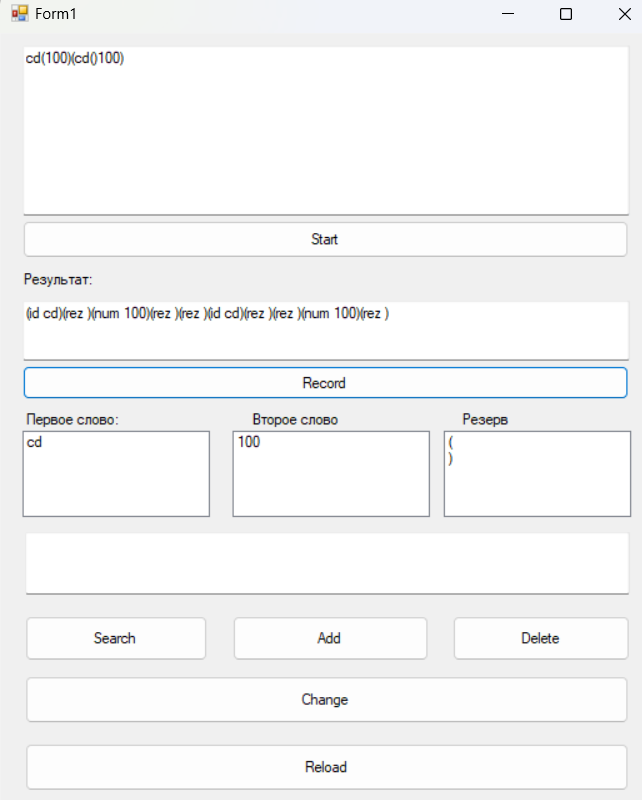
for (int i = 0; i < arrFHeapTable.Count; i++)

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

}

}

}

****

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

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

**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.

1. 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.

1. 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.

1. Упрощение дерева (по необходимости):

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

1. 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.

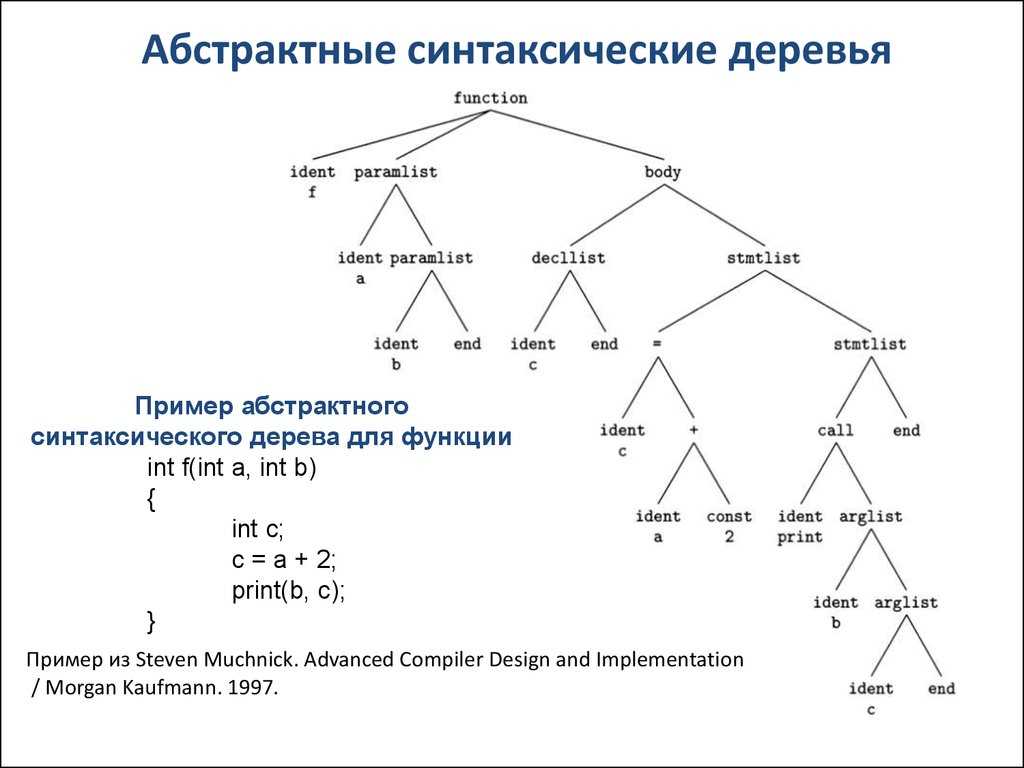
Аннотации и информация: синтаксическое дерево может быть аннотировано дополнительной информацией, такой как типы данных, области видимости, местоположение в исходном коде и другие атрибуты. Эта информация полезна при выполнении семантического анализа и других видов анализа.

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.

Пример синтаксического дерева:



**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 nsSynt;**

**using nsLex;**

**using static System.Windows.Forms.VisualStyles.VisualStyleElement;**

**namespace WindowsFormsApp15**

**{**

**public partial class Form1 : Form**

**{**

**public CHashTableList htl = new CHashTableList(3);**

**TreeInMemory OsnovnDerevo, Rez;**

**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();**

**}**

**public Form1()**

**{**

**InitializeComponent();**

**tbFSource.AppendText("cd(010100000)(cd(100)100000)");**

**int n = tbFSource.Lines.Length;**

**Rez = new TreeInMemory();**

**OsnovnDerevo = new TreeInMemory();**

**}**

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

**{**

**tbFMessage.Clear();**

**uSyntAnalyzer Synt = new uSyntAnalyzer();**

**Synt.Lex.strPSource = tbFSource.Lines;**

**Synt.Lex.strPMessage = tbFMessage.Lines;**

**Synt.Lex.enumPState = TState.Start;**

**try**

**{**

**Synt.Lex.NextToken();**

**Synt.S(Rez);**

**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 button4\_Click(object sender, EventArgs e)**

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource.Lines;**

**Lex.strPMessage = tbFMessage.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource.TextLength;**

**int y = tbFSource.Lines.Length;**

**tbFMessage.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**string s1 = "", s = "";**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**s1 = "id " + Lex.strPLexicalUnit;**

**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;**

**}**

**break;**

**}**

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

**tbFMessage.Text += m;**

**}**

**}**

**catch (Exception exc)**

**{**

**tbFMessage.Text += exc.Message;**

**tbFSource.Select();**

**tbFSource.SelectionStart = 0;**

**int n = 0;**

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

**n += Lex.intPSourceColSelection;**

**tbFSource.SelectionLength = n;**

**}**

**}**

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

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource1.Lines;**

**Lex.strPMessage = tbFMessage1.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource1.TextLength;**

**CLex Lex2 = new CLex();**

**Lex2.strPSource = tbFSource2.Lines;**

**Lex2.strPMessage = tbFMessage1.Lines;**

**Lex2.intPSourceColSelection = 0;**

**Lex2.intPSourceRowSelection = 0;**

**int x2 = tbFSource2.TextLength;**

**tbFMessage1.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**int b = 0;**

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

**{**

**while (Lex2.enumPState != TState.Finish)**

**{**

**Lex2.NextToken();**

**switch (Lex2.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**int b1 = 0;**

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

**{**

**htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 0);**

**TablesToMemo(this, e);**

**}**

**tbFMessage1.AppendText("(Идентификатор был изменен)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**tbFMessage1.AppendText("(Нельзя заменить идентификатор на номер)");**

**break;**

**}**

**}**

**}**

**}**

**else tbFMessage1.AppendText("(Идентификатор для удаления не найден)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**int b = 0;**

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

**{**

**while (Lex2.enumPState != TState.Finish)**

**{**

**Lex2.NextToken();**

**switch (Lex2.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**tbFMessage1.AppendText("(Нельзя заменить номер на идентификатор)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**int b1 = 0;**

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

**{**

**htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 1);**

**TablesToMemo(this, e);**

**}**

**tbFMessage1.AppendText("(Номер был изменен)");**

**break;**

**}**

**}**

**}**

**}**

**else tbFMessage1.AppendText("(Номер для удаления не найден)");**

**break;**

**}**

**}**

**}**

**}**

**catch (Exception exc)**

**{**

**tbFMessage1.Text += exc.Message;**

**tbFSource.Select();**

**tbFSource.SelectionStart = 0;**

**int n1 = 0;**

**for (int i = 0; i < Lex.intPSourceRowSelection; i++) n1 += tbFSource.Lines[i].Length + 2;**

**n1 += Lex.intPSourceColSelection;**

**tbFSource.SelectionLength = n1;**

**}**

**}**

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

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource.Lines;**

**Lex.strPMessage = tbFMessage.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource.TextLength;**

**int y = tbFSource.Lines.Length;**

**tbFMessage.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**string s1 = "", s = "";**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**s1 = "id " + Lex.strPLexicalUnit; 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 button3\_Click(object sender, EventArgs e)**

**{**

**tbFMessage2.Clear();**

**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();**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**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:**

**{**

**int b = 0;**

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

**{**

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

**}**

**else tbFMessage2.AppendText("(Номер не найден)");**

**break;**

**}**

**}**

**}**

**}**

**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 button6\_Click(object sender, EventArgs e)**

**{**

**tbFMessage.Clear();**

**uSyntAnalyzer Synt = new uSyntAnalyzer();**

**Synt.Lex.strPSource = tbFSource.Lines;**

**Synt.Lex.strPMessage = tbFMessage.Lines;**

**Synt.Lex.enumPState = TState.Start;**

**try**

**{**

**Synt.Lex.NextToken();**

**Synt.S(OsnovnDerevo);**

**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;**

**}**

**Star(OsnovnDerevo);**

**treeView1.ExpandAll();**

**treeView2.ExpandAll();**

**}**

**public void potomki(TreeInMemory q, TreeNode w)**

**{**

**if (q.doch != null)**

**{**

**w.Nodes.Add(q.doch.Name + "");**

**potomki(q.doch, w.Nodes[0]);**

**Console.WriteLine(q.doch);**

**if (q.doch.sestra != null)**

**{**

**int r = 1;**

**TreeInMemory e = q.doch.sestra;**

**TreeNode d = w;**

**while (e != null)**

**{**

**d.Nodes.Add(e.Name + "");**

**potomki(e, d.Nodes[r]);**

**r++;**

**e = e.sestra;**

**}**

**}**

**}**

**}**

**private void convertBtn\_Click(object sender, EventArgs e)**

**{**

**TreeConverter treeConverter = new TreeConverter();**

**treeConverter.ConvertBinaryToDecimal(treeView1);**

**}**

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

**{**

**TreeConverter treeConverter = new TreeConverter();**

**treeConverter.ConvertBinaryToDecimal(treeView1);**

**}**

**private void ConverterSwapBtn\_Click(object sender, EventArgs e)**

**{**

**TreeConverterSwap treeConverter = new TreeConverterSwap();**

**treeConverter.ConvertBinaryToDecimal(treeView2);**

**}**

**private void reload\_Click(object sender, EventArgs e)**

**{**

**treeView1.Nodes.Clear();**

**treeView2.Nodes.Clear();**

**listBox1.Items.Clear();**

**listBox2.Items.Clear();**

**//listBox3.Items.Clear();**

**}**

**public void Star(TreeInMemory q)**

**{**

**treeView1.Nodes.Clear();**

**treeView2.Nodes.Clear();**

**if (q.doch != null)**

**{**

**treeView1.Nodes.Add("S");**

**treeView2.Nodes.Add("S");**

**potomki(q, treeView1.Nodes[0]);**

**potomki(q, treeView2.Nodes[0]);**

**}**

**}**

**}**

**}**

**TreeInMemory.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApp15

{

public class TreeInMemory

{

public string Name;

public TreeInMemory doch;

public TreeInMemory sestra;

public void dobavitDoch(string novuzel)

{

doch = new TreeInMemory();

doch.Name = novuzel;

}

public void dobavitSestru(string novuzel)

{

sestra = new TreeInMemory();

sestra.Name = novuzel;

}

}

}

**HashTables.cs:**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace nsSynt

{

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;

// конструктор класса

public THashTable(ref THeap objAHeap)

{

objFHeap = objAHeap;

Init(7);

intFItemReserve = 0;

}

// инициализация хэш-таблицы

public void Init(int count)

{

arrFHashTable.Clear();

Resize(arrFHashTable, count);

intFCurrentPrimeNumber = count;

}

// Данный метод Resize изменяет размер списка list до size. Если size больше текущего размера списка, он добавляет пустые объекты в список, пока размер не станет равным size. Если size меньше текущего размера списка, он удаляет последние объекты из списка, пока размер не станет равным size.

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);

}

// Функция NextPrimeNumber принимает на вход число cardAOldPrimeNumber и возвращает следующее простое число

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

// функция вычисляет хэш-значение строки "strALexicalUnit", применяя операцию получения остатка от деления каждого символа на 12, и суммируя полученные значения

UInt32 HashFunction\_Wainberger(string strALexicalUnit)

{

UInt32 h = 0/\*, seed=131313\*/;

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

{

h += strALexicalUnit[i]; //хеширование

}

return h;

}

// Это функция ReHashFunction\_Line, которая применяется для рехеширования в линейном методе открытой адресации.

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); //рехеширование

}

} while (!boolVFinish);

return hi;

}

// Он использует две функции хеширования - HashFunction\_Wainberger и ReHashFunction\_Line - для вычисления индекса хеша. Индекс хеша сохраняется в переменную intFHashIndex.

public void HashIndex(string strALexicalUnit)

{

int h;

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

intFHashIndex = ReHashFunction\_Line(h, strALexicalUnit);

}

// Это функция TableReHashing(), которая выполняет перехеширование в хэш-таблице.

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);

}

// поиск лексической единицы

public bool SearchLexicalUnit(string strAlexicalUnit, ref int intALexicalCode)

{

HashIndex(strAlexicalUnit);

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

else

{

intALexicalCode = arrFHashTable[intFHashIndex];

return true;

}

}

// добавление лексической единицы

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;

}

}

// удаление лексической единицы

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();

}

}

}

// удаление элемента из таблицы по его лексическому коду

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; }

}

// игнорирует элементы с нулевым значением

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);

}

}

//HashTableList

public class CHashTableList

{

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

private bool boolFIsSaved;

public bool boolFIsLoaded;

private byte byteFTablesSize;

static THeap objFHeap = new THeap();

// конструктор класса

public CHashTableList(byte byteATableCount)

{

this.byteFTablesSize = byteATableCount;

objFHeap = new THeap();

Resize(arrFHashTableList, byteATableCount);

}

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

public byte GetTableNumber(int intALexicalCode)

{

return objFHeap.arrFHeapTable[intALexicalCode].byteFHashTable;

}

//кол-во элементов в списке

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; }

}

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

// 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;

}

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

}

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);

}

}

}

**TreeInMemory.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApp15

{

public class TreeInMemory

{

public string Name;

public TreeInMemory doch;

public TreeInMemory sestra;

public void dobavitDoch(string novuzel)

{

doch = new TreeInMemory();

doch.Name = novuzel;

}

public void dobavitSestru(string novuzel)

{

sestra = new TreeInMemory();

sestra.Name = novuzel;

}

}

}

**uLex.cs:**

**using System;**

**using System.Collections.Generic;**

**using System.Text;**

**namespace nsLex**

**{**

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

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

**public enum TToken**

**{**

**mn, minus, plus, lxmIdentifier, lxmNumber, lxmUnknown, lxmEmpty,**

**lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmComma, lxmdt, lxmText, lxmtz, lxmr, lxmrs, lxmls**

**};**

**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 == '\*' || 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 == ';' || chrFSelection == ',' || chrFSelection == '[' || chrFSelection == ']' || chrFSelection == '=' || chrFSelection == ':') enumFSelectionCharType = TCharType.ReservedSymbol;**

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

**enumFState = TState.Continue;**

**}**

**}**

**private void TakeSymbol()**

**{**

**char[] c = { chrFSelection };**

**String s = new string(c);**

**strFLexicalUnit += s;**

**GetSymbol();**

**}**

**public void NextToken()**

**{**

**strFLexicalUnit = "";**

**if (enumFState == TState.Start)**

**{**

**intFSourceRowSelection = 0;**

**intFSourceColSelection = -1;**

**GetSymbol();**

**}**

**if (chrFSelection == '/')**

**{**

**GetSymbol();**

**if (chrFSelection == '/')**

**while (enumFSelectionCharType != TCharType.EndRow)**

**{**

**GetSymbol();**

**}**

**GetSymbol();**

**}**

**//Вариант 17**

**switch (enumFSelectionCharType)**

**{**

**case TCharType.Letter:**

**{**

**// a b c d**

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

**// B | | | |CFin|**

**// CFin |CFin|CFin|CFin|CFin|**

**A:**

**{**

**if (chrFSelection == 'c')**

**{**

**TakeSymbol();**

**goto B;**

**}**

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

**}**

**B:**

**{**

**if (chrFSelection == 'd')**

**{**

**TakeSymbol();**

**goto CFin;**

**}**

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

**}**

**CFin:**

**{**

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

**{**

**TakeSymbol();**

**goto CFin;**

**}**

**else**

**{**

**enumFToken = TToken.lxmIdentifier;**

**return;**

**}**

**}**

**}**

**if (chrFSelection == '/')**

**{**

**GetSymbol();**

**if (chrFSelection == '/')**

**while (enumFSelectionCharType != TCharType.EndRow)**

**{**

**GetSymbol();**

**}**

**GetSymbol();**

**}**

**case TCharType.Digit:**

**{**

**// 0 1**

**// A | 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 == '0')**

**{**

**TakeSymbol();**

**goto A;**

**}**

**else throw new Exception("Ожидался 0");**

**E:**

**if (chrFSelection == '0')**

**{**

**TakeSymbol();**

**goto FFin;**

**}**

**else throw new Exception("Ожидалась 0");**

**FFin:**

**if (chrFSelection == '0')**

**{**

**TakeSymbol();**

**goto G;**

**}**

**else if (enumFSelectionCharType != TCharType.Digit)**

**{**

**enumFToken = TToken.lxmNumber;**

**return;**

**}**

**else throw new Exception("Ожидалась 0");**

**G:**

**if (chrFSelection == '0')**

**{**

**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;**

**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;**

**}**

**break;**

**}**

**case TCharType.EndText:**

**{**

**enumFToken = TToken.lxmEmpty;**

**break;**

**}**

**}**

**}**

**}**

**}**

**uSyntAnalyzer.cs:**

using System;

using System.Collections.Generic;

using System.Text;

using System.Windows.Forms;

using nsLex;

using WindowsFormsApp15;

namespace nsSynt

{

class uSyntAnalyzer

{

public TreeInMemory VremenDerevo;

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 int indexLxmLeftParenth = 0;

public int indexLxmRightParenth = 0;

public uSyntAnalyzer()

{

strFSource = new String[0];

strFMessage = new String[0];

strPSource = new String[0];

strPMessage = new String[0];

VremenDerevo = new TreeInMemory();

}

public uSyntAnalyzer(TreeInMemory mainTree)

{

strFSource = new String[0];

strFMessage = new String[0];

strPSource = new String[0];

strPMessage = new String[0];

VremenDerevo = new TreeInMemory();

VremenDerevo = mainTree;

}

public void S(TreeInMemory el)

{

if (el == VremenDerevo) { el.Name = "Start"; }

if (Lex.enumPToken == TToken.lxmIdentifier)

{

el.dobavitDoch(Lex.strPLexicalUnit);

Lex.NextToken();

if (Lex.enumPToken != TToken.lxmEmpty)

{

el.doch.dobavitSestru("S");

S(el.doch.sestra);

}

else throw new Exception("ожидалось идентификатор, число или скобка");

if (Lex.enumPToken != TToken.lxmEmpty || Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmLeftParenth)

{

el.doch.sestra.dobavitSestru("A");

A(el.doch.sestra.sestra);

}

else throw new Exception("ожидалось число или скобка");

}

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

{

el.dobavitDoch("A");

A(el.doch);

}

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

{

throw new Exception("ожидалось скобка (");

}

}

public void A(TreeInMemory el)

{

if (Lex.enumPToken == TToken.lxmNumber)

{

el.dobavitDoch(Lex.strPLexicalUnit);

Lex.NextToken();

}

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

{

el.dobavitDoch("(");

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmRightParenth)

{

el.doch.dobavitSestru(")");

Lex.NextToken();

}

else if (Lex.enumPToken != TToken.lxmRightParenth)

{

el.doch.dobavitSestru("S");

S(el.doch.sestra);

if (Lex.enumPToken == TToken.lxmRightParenth)

{

el.doch.sestra.dobavitSestru(")");

Lex.NextToken();

}

else throw new Exception("не хватает скобки )");

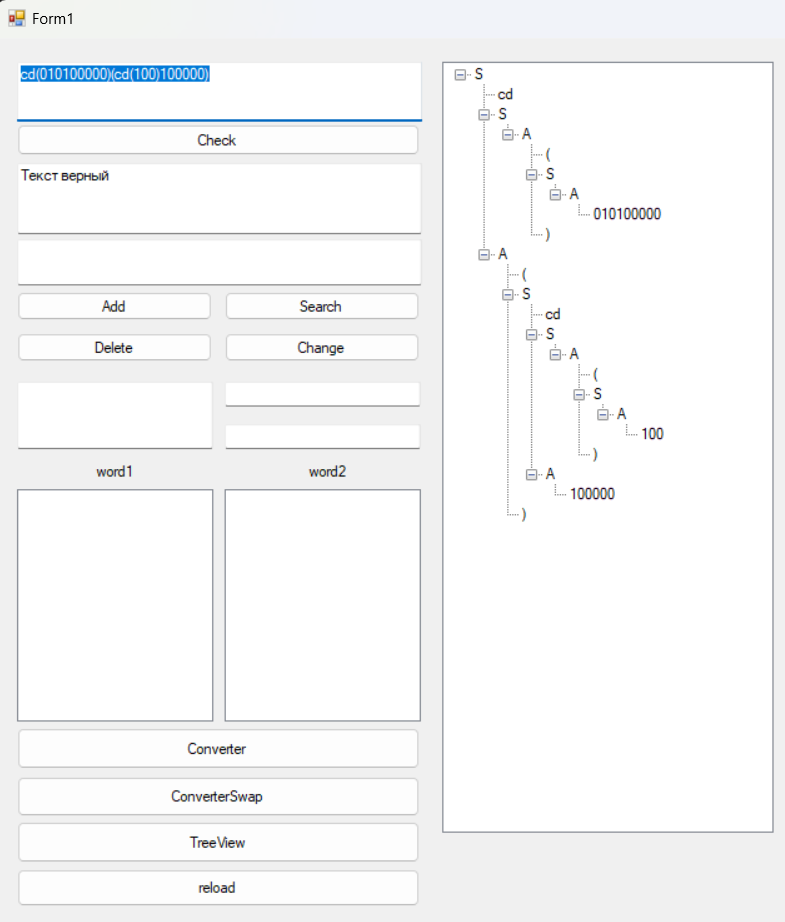
}

}

}

}

}

****

**Лабораторная работа № 6. Разработка генератора (2 часа)**

**Задание:**

1. Перевести все числа в десятичное представление.

2. Выполнить вывод исходного текста в структурированном виде.

**Краткое теоретическое обоснование:**

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.

Элементы структурированного представления TreeView:

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

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

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. Она включает:

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.

Генераторы могут быть написаны на различных языках программирования и часто используются в разработке программного обеспечения, чтобы сократить рутинную работу и обеспечить согласованность и точность в выходных данных.

**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 nsSynt;**

**using nsLex;**

**using static System.Windows.Forms.VisualStyles.VisualStyleElement;**

**namespace WindowsFormsApp15**

**{**

**public partial class Form1 : Form**

**{**

**public CHashTableList htl = new CHashTableList(3);**

**TreeInMemory OsnovnDerevo, Rez;**

**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();**

**}**

**public Form1()**

**{**

**InitializeComponent();**

**tbFSource.AppendText("cd(010100000)(cd(100)100000)");**

**int n = tbFSource.Lines.Length;**

**Rez = new TreeInMemory();**

**OsnovnDerevo = new TreeInMemory();**

**}**

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

**{**

**tbFMessage.Clear();**

**uSyntAnalyzer Synt = new uSyntAnalyzer();**

**Synt.Lex.strPSource = tbFSource.Lines;**

**Synt.Lex.strPMessage = tbFMessage.Lines;**

**Synt.Lex.enumPState = TState.Start;**

**try**

**{**

**Synt.Lex.NextToken();**

**Synt.S(Rez);**

**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 button4\_Click(object sender, EventArgs e)**

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource.Lines;**

**Lex.strPMessage = tbFMessage.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource.TextLength;**

**int y = tbFSource.Lines.Length;**

**tbFMessage.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**string s1 = "", s = "";**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**s1 = "id " + Lex.strPLexicalUnit;**

**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;**

**}**

**break;**

**}**

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

**tbFMessage.Text += m;**

**}**

**}**

**catch (Exception exc)**

**{**

**tbFMessage.Text += exc.Message;**

**tbFSource.Select();**

**tbFSource.SelectionStart = 0;**

**int n = 0;**

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

**n += Lex.intPSourceColSelection;**

**tbFSource.SelectionLength = n;**

**}**

**}**

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

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource1.Lines;**

**Lex.strPMessage = tbFMessage1.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource1.TextLength;**

**CLex Lex2 = new CLex();**

**Lex2.strPSource = tbFSource2.Lines;**

**Lex2.strPMessage = tbFMessage1.Lines;**

**Lex2.intPSourceColSelection = 0;**

**Lex2.intPSourceRowSelection = 0;**

**int x2 = tbFSource2.TextLength;**

**tbFMessage1.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**int b = 0;**

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

**{**

**while (Lex2.enumPState != TState.Finish)**

**{**

**Lex2.NextToken();**

**switch (Lex2.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**int b1 = 0;**

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

**{**

**htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 0);**

**TablesToMemo(this, e);**

**}**

**tbFMessage1.AppendText("(Идентификатор был изменен)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**tbFMessage1.AppendText("(Нельзя заменить идентификатор на номер)");**

**break;**

**}**

**}**

**}**

**}**

**else tbFMessage1.AppendText("(Идентификатор для удаления не найден)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**int b = 0;**

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

**{**

**while (Lex2.enumPState != TState.Finish)**

**{**

**Lex2.NextToken();**

**switch (Lex2.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**tbFMessage1.AppendText("(Нельзя заменить номер на идентификатор)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**int b1 = 0;**

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

**{**

**htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 1);**

**TablesToMemo(this, e);**

**}**

**tbFMessage1.AppendText("(Номер был изменен)");**

**break;**

**}**

**}**

**}**

**}**

**else tbFMessage1.AppendText("(Номер для удаления не найден)");**

**break;**

**}**

**}**

**}**

**}**

**catch (Exception exc)**

**{**

**tbFMessage1.Text += exc.Message;**

**tbFSource.Select();**

**tbFSource.SelectionStart = 0;**

**int n1 = 0;**

**for (int i = 0; i < Lex.intPSourceRowSelection; i++) n1 += tbFSource.Lines[i].Length + 2;**

**n1 += Lex.intPSourceColSelection;**

**tbFSource.SelectionLength = n1;**

**}**

**}**

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

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource.Lines;**

**Lex.strPMessage = tbFMessage.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource.TextLength;**

**int y = tbFSource.Lines.Length;**

**tbFMessage.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**string s1 = "", s = "";**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**s1 = "id " + Lex.strPLexicalUnit; 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 button3\_Click(object sender, EventArgs e)**

**{**

**tbFMessage2.Clear();**

**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();**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**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:**

**{**

**int b = 0;**

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

**{**

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

**}**

**else tbFMessage2.AppendText("(Номер не найден)");**

**break;**

**}**

**}**

**}**

**}**

**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 button6\_Click(object sender, EventArgs e)**

**{**

**tbFMessage.Clear();**

**uSyntAnalyzer Synt = new uSyntAnalyzer();**

**Synt.Lex.strPSource = tbFSource.Lines;**

**Synt.Lex.strPMessage = tbFMessage.Lines;**

**Synt.Lex.enumPState = TState.Start;**

**try**

**{**

**Synt.Lex.NextToken();**

**Synt.S(OsnovnDerevo);**

**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;**

**}**

**Star(OsnovnDerevo);**

**treeView1.ExpandAll();**

**treeView2.ExpandAll();**

**}**

**public void potomki(TreeInMemory q, TreeNode w)**

**{**

**if (q.doch != null)**

**{**

**w.Nodes.Add(q.doch.Name + "");**

**potomki(q.doch, w.Nodes[0]);**

**Console.WriteLine(q.doch);**

**if (q.doch.sestra != null)**

**{**

**int r = 1;**

**TreeInMemory e = q.doch.sestra;**

**TreeNode d = w;**

**while (e != null)**

**{**

**d.Nodes.Add(e.Name + "");**

**potomki(e, d.Nodes[r]);**

**r++;**

**e = e.sestra;**

**}**

**}**

**}**

**}**

**private void convertBtn\_Click(object sender, EventArgs e)**

**{**

**TreeConverter treeConverter = new TreeConverter();**

**treeConverter.ConvertBinaryToDecimal(treeView1);**

**}**

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

**{**

**TreeConverter treeConverter = new TreeConverter();**

**treeConverter.ConvertBinaryToDecimal(treeView1);**

**}**

**private void ConverterSwapBtn\_Click(object sender, EventArgs e)**

**{**

**TreeConverterSwap treeConverter = new TreeConverterSwap();**

**treeConverter.ConvertBinaryToDecimal(treeView2);**

**}**

**private void reload\_Click(object sender, EventArgs e)**

**{**

**treeView1.Nodes.Clear();**

**treeView2.Nodes.Clear();**

**listBox1.Items.Clear();**

**listBox2.Items.Clear();**

**//listBox3.Items.Clear();**

**}**

**public void Star(TreeInMemory q)**

**{**

**treeView1.Nodes.Clear();**

**treeView2.Nodes.Clear();**

**if (q.doch != null)**

**{**

**treeView1.Nodes.Add("S");**

**treeView2.Nodes.Add("S");**

**potomki(q, treeView1.Nodes[0]);**

**potomki(q, treeView2.Nodes[0]);**

**}**

**}**

**}**

**}**

**THashTable.cs:**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace nsSynt

{

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;

// конструктор класса

public THashTable(ref THeap objAHeap)

{

objFHeap = objAHeap;

Init(7);

intFItemReserve = 0;

}

// инициализация хэш-таблицы

public void Init(int count)

{

arrFHashTable.Clear();

Resize(arrFHashTable, count);

intFCurrentPrimeNumber = count;

}

// Данный метод Resize изменяет размер списка list до size. Если size больше текущего размера списка, он добавляет пустые объекты в список, пока размер не станет равным size. Если size меньше текущего размера списка, он удаляет последние объекты из списка, пока размер не станет равным size.

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);

}

// Функция NextPrimeNumber принимает на вход число cardAOldPrimeNumber и возвращает следующее простое число

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

// функция вычисляет хэш-значение строки "strALexicalUnit", применяя операцию получения остатка от деления каждого символа на 12, и суммируя полученные значения

UInt32 HashFunction\_Wainberger(string strALexicalUnit)

{

UInt32 h = 0/\*, seed=131313\*/;

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

{

h += strALexicalUnit[i]; //хеширование

}

return h;

}

// Это функция ReHashFunction\_Line, которая применяется для рехеширования в линейном методе открытой адресации.

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); //рехеширование

}

} while (!boolVFinish);

return hi;

}

// Он использует две функции хеширования - HashFunction\_Wainberger и ReHashFunction\_Line - для вычисления индекса хеша. Индекс хеша сохраняется в переменную intFHashIndex.

public void HashIndex(string strALexicalUnit)

{

int h;

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

intFHashIndex = ReHashFunction\_Line(h, strALexicalUnit);

}

// Это функция TableReHashing(), которая выполняет перехеширование в хэш-таблице.

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);

}

// поиск лексической единицы

public bool SearchLexicalUnit(string strAlexicalUnit, ref int intALexicalCode)

{

HashIndex(strAlexicalUnit);

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

else

{

intALexicalCode = arrFHashTable[intFHashIndex];

return true;

}

}

// добавление лексической единицы

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;

}

}

// удаление лексической единицы

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();

}

}

}

// удаление элемента из таблицы по его лексическому коду

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; }

}

// игнорирует элементы с нулевым значением

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);

}

}

//HashTableList

public class CHashTableList

{

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

private bool boolFIsSaved;

public bool boolFIsLoaded;

private byte byteFTablesSize;

static THeap objFHeap = new THeap();

// конструктор класса

public CHashTableList(byte byteATableCount)

{

this.byteFTablesSize = byteATableCount;

objFHeap = new THeap();

Resize(arrFHashTableList, byteATableCount);

}

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

public byte GetTableNumber(int intALexicalCode)

{

return objFHeap.arrFHeapTable[intALexicalCode].byteFHashTable;

}

//кол-во элементов в списке

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; }

}

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

// 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;

}

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

}

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 WindowsFormsApp15

{

internal class TreeConverter

{

public void ConvertBinaryToDecimal(TreeView tree)

{

foreach (TreeNode node in tree.Nodes)

{

ConvertBinaryToDecimal(node);

}

}

private void ConvertBinaryToDecimal(TreeNode node)

{

if (node.Text == "A" || node.Text == "S")

{

if (node.Nodes.Count > 0)

{

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

if (IsBin(binaryValue))

{

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

node2.Text = decimalValue.ToString();

}

}

}

}

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;

}

}

}

**TreeConverterSwap.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp15

{

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 void ConvertBinaryToDecimal(TreeView tree)

{

foreach (TreeNode node in tree.Nodes)

{

ConvertBinaryToDecimal(node);

}

foreach (var item in myDict)

{

list.Add(item.Key);

strings.Add(item.Value);

}

foreach (TreeNode node in tree.Nodes)

{

Perebor(node);

}

}

private void ConvertBinaryToDecimal(TreeNode node)

{

if (node.Text == "A")

{

if (node.Nodes.Count == 1)

{

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

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 void Perebor(TreeNode node)

{

if (node.Nodes.Count > 0)

{

foreach (TreeNode Node in node.Nodes)

{

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);

}

}

}

}

**TreeInMemory.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApp15

{

public class TreeInMemory

{

public string Name;

public TreeInMemory doch;

public TreeInMemory sestra;

public void dobavitDoch(string novuzel)

{

doch = new TreeInMemory();

doch.Name = novuzel;

}

public void dobavitSestru(string novuzel)

{

sestra = new TreeInMemory();

sestra.Name = novuzel;

}

}

}

**uLex.cs:**

**using System;**

**using System.Collections.Generic;**

**using System.Text;**

**namespace nsLex**

**{**

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

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

**public enum TToken**

**{**

**mn, minus, plus, lxmIdentifier, lxmNumber, lxmUnknown, lxmEmpty,**

**lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmComma, lxmdt, lxmText, lxmtz, lxmr, lxmrs, lxmls**

**};**

**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 == '\*' || 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 == ';' || chrFSelection == ',' || chrFSelection == '[' || chrFSelection == ']' || chrFSelection == '=' || chrFSelection == ':') enumFSelectionCharType = TCharType.ReservedSymbol;**

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

**enumFState = TState.Continue;**

**}**

**}**

**private void TakeSymbol()**

**{**

**char[] c = { chrFSelection };**

**String s = new string(c);**

**strFLexicalUnit += s;**

**GetSymbol();**

**}**

**public void NextToken()**

**{**

**strFLexicalUnit = "";**

**if (enumFState == TState.Start)**

**{**

**intFSourceRowSelection = 0;**

**intFSourceColSelection = -1;**

**GetSymbol();**

**}**

**if (chrFSelection == '/')**

**{**

**GetSymbol();**

**if (chrFSelection == '/')**

**while (enumFSelectionCharType != TCharType.EndRow)**

**{**

**GetSymbol();**

**}**

**GetSymbol();**

**}**

**//Вариант 17**

**switch (enumFSelectionCharType)**

**{**

**case TCharType.Letter:**

**{**

**// a b c d**

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

**// B | | | |CFin|**

**// CFin |CFin|CFin|CFin|CFin|**

**A:**

**{**

**if (chrFSelection == 'c')**

**{**

**TakeSymbol();**

**goto B;**

**}**

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

**}**

**B:**

**{**

**if (chrFSelection == 'd')**

**{**

**TakeSymbol();**

**goto CFin;**

**}**

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

**}**

**CFin:**

**{**

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

**{**

**TakeSymbol();**

**goto CFin;**

**}**

**else**

**{**

**enumFToken = TToken.lxmIdentifier;**

**return;**

**}**

**}**

**}**

**if (chrFSelection == '/')**

**{**

**GetSymbol();**

**if (chrFSelection == '/')**

**while (enumFSelectionCharType != TCharType.EndRow)**

**{**

**GetSymbol();**

**}**

**GetSymbol();**

**}**

**case TCharType.Digit:**

**{**

**// 0 1**

**// A | 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 == '0')**

**{**

**TakeSymbol();**

**goto A;**

**}**

**else throw new Exception("Ожидался 0");**

**E:**

**if (chrFSelection == '0')**

**{**

**TakeSymbol();**

**goto FFin;**

**}**

**else throw new Exception("Ожидалась 0");**

**FFin:**

**if (chrFSelection == '0')**

**{**

**TakeSymbol();**

**goto G;**

**}**

**else if (enumFSelectionCharType != TCharType.Digit)**

**{**

**enumFToken = TToken.lxmNumber;**

**return;**

**}**

**else throw new Exception("Ожидалась 0");**

**G:**

**if (chrFSelection == '0')**

**{**

**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;**

**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;**

**}**

**break;**

**}**

**case TCharType.EndText:**

**{**

**enumFToken = TToken.lxmEmpty;**

**break;**

**}**

**}**

**}**

**}**

**}**

**uSyntAnalyzer.cs:**

using System;

using System.Collections.Generic;

using System.Text;

using System.Windows.Forms;

using nsLex;

using WindowsFormsApp15;

namespace nsSynt

{

class uSyntAnalyzer

{

public TreeInMemory VremenDerevo;

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 int indexLxmLeftParenth = 0;

public int indexLxmRightParenth = 0;

public uSyntAnalyzer()

{

strFSource = new String[0];

strFMessage = new String[0];

strPSource = new String[0];

strPMessage = new String[0];

VremenDerevo = new TreeInMemory();

}

public uSyntAnalyzer(TreeInMemory mainTree)

{

strFSource = new String[0];

strFMessage = new String[0];

strPSource = new String[0];

strPMessage = new String[0];

VremenDerevo = new TreeInMemory();

VremenDerevo = mainTree;

}

public void S(TreeInMemory el)

{

if (el == VremenDerevo) { el.Name = "Start"; }

if (Lex.enumPToken == TToken.lxmIdentifier)

{

el.dobavitDoch(Lex.strPLexicalUnit);

Lex.NextToken();

if (Lex.enumPToken != TToken.lxmEmpty)

{

el.doch.dobavitSestru("S");

S(el.doch.sestra);

}

else throw new Exception("ожидалось идентификатор, число или скобка");

if (Lex.enumPToken != TToken.lxmEmpty || Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmLeftParenth)

{

el.doch.sestra.dobavitSestru("A");

A(el.doch.sestra.sestra);

}

else throw new Exception("ожидалось число или скобка");

}

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

{

el.dobavitDoch("A");

A(el.doch);

}

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

{

throw new Exception("ожидалось скобка (");

}

}

public void A(TreeInMemory el)

{

if (Lex.enumPToken == TToken.lxmNumber)

{

el.dobavitDoch(Lex.strPLexicalUnit);

Lex.NextToken();

}

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

{

el.dobavitDoch("(");

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmRightParenth)

{

el.doch.dobavitSestru(")");

Lex.NextToken();

}

else if (Lex.enumPToken != TToken.lxmRightParenth)

{

el.doch.dobavitSestru("S");

S(el.doch.sestra);

if (Lex.enumPToken == TToken.lxmRightParenth)

{

el.doch.sestra.dobavitSestru(")");

Lex.NextToken();

}

else throw new Exception("не хватает скобки )");

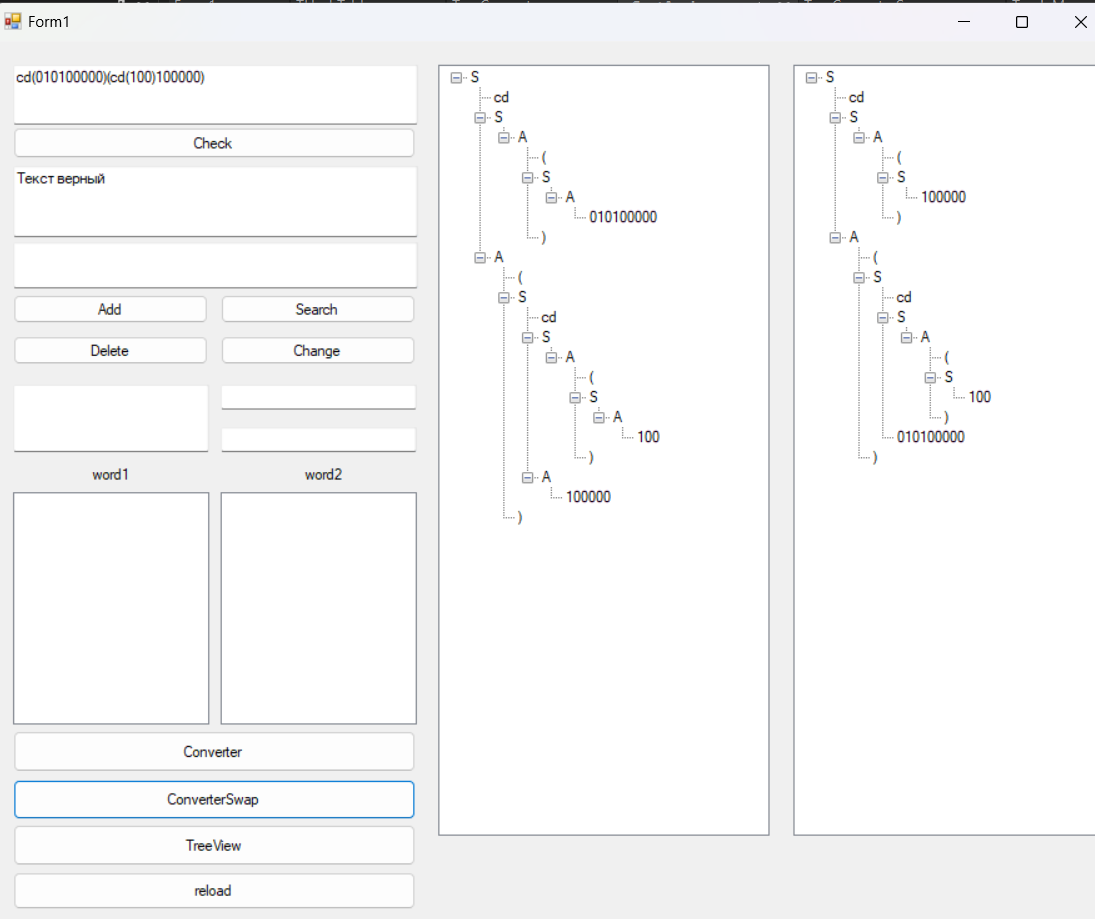
}

}

}

}

}

****

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

**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.

**Вариант 17:** все числа должны быть разными.

**Краткое теоретическое обоснование:**

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

1. **Определение семантических правил:** начните с определения семантических правил для вашего языка программирования. Семантические правила определяют, какие операции и действия разрешены в программе, и какие ограничения применяются к типам данных и выражениям.
2. **Аннотация синтаксического дерева:** в контекстном анализе сначала строится синтаксическое дерево (или абстрактное синтаксическое дерево) на основе синтаксического анализа. Синтаксическое дерево представляет собой иерархическую структуру программы. Затем синтаксическое дерево аннотируется семантической информацией. Это включает в себя привязку имен к их объявлениям (разрешение идентификаторов), вычисление типов данных выражений и другие проверки семантической корректности.
3. **Проверки семантических ошибок:** контекстный анализатор проводит проверки на наличие семантических ошибок в программе. Эти ошибки могут включать в себя попытку использования необъявленных переменных, присвоение значений переменным с неподходящими типами данных и другие нарушения семантических правил.
4. **Генерация семантического дерева или промежуточного представления:** после успешной проверки семантических правил, контекстный анализатор может сгенерировать семантическое дерево или промежуточное представление программы, которое будет использоваться для дальнейшего анализа и генерации кода.
5. **Проверка типов данных:** один из основных аспектов контекстного анализа - это проверка соответствия типов данных. Контекстный анализатор определяет типы данных выражений и операций, а также убеждается в их совместимости. Например, он проверяет, что операторы, применяемые к переменным или значениям, имеют совместимые типы.
6. **Проверка семантических правил:** контекстный анализатор применяет семантические правила языка программирования. Эти правила могут включать в себя ограничения на использование ключевых слов, операторов и структур данных.
7. **Выдача сообщений об ошибках:** контекстный анализатор генерирует сообщения об ошибках, если находит нарушения семантических правил. Эти сообщения об ошибках помогают разработчикам исправить проблемы в коде.

**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 nsSynt;**

**using nsLex;**

**using static System.Windows.Forms.VisualStyles.VisualStyleElement;**

**namespace WindowsFormsApp15**

**{**

**public partial class Form1 : Form**

**{**

**public CHashTableList htl = new CHashTableList(3);**

**TreeInMemory OsnovnDerevo, Rez;**

**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();**

**}**

**public Form1()**

**{**

**InitializeComponent();**

**tbFSource.AppendText("cd(010100000)(cd(100)100000)");**

**int n = tbFSource.Lines.Length;**

**Rez = new TreeInMemory();**

**OsnovnDerevo = new TreeInMemory();**

**}**

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

**{**

**tbFMessage.Clear();**

**uSyntAnalyzer Synt = new uSyntAnalyzer();**

**Synt.Lex.strPSource = tbFSource.Lines;**

**Synt.Lex.strPMessage = tbFMessage.Lines;**

**Synt.Lex.enumPState = TState.Start;**

**try**

**{**

**Synt.Lex.NextToken();**

**Synt.S(Rez);**

**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 button4\_Click(object sender, EventArgs e)**

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource.Lines;**

**Lex.strPMessage = tbFMessage.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource.TextLength;**

**int y = tbFSource.Lines.Length;**

**tbFMessage.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**string s1 = "", s = "";**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**s1 = "id " + Lex.strPLexicalUnit;**

**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;**

**}**

**break;**

**}**

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

**tbFMessage.Text += m;**

**}**

**}**

**catch (Exception exc)**

**{**

**tbFMessage.Text += exc.Message;**

**tbFSource.Select();**

**tbFSource.SelectionStart = 0;**

**int n = 0;**

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

**n += Lex.intPSourceColSelection;**

**tbFSource.SelectionLength = n;**

**}**

**}**

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

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource1.Lines;**

**Lex.strPMessage = tbFMessage1.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource1.TextLength;**

**CLex Lex2 = new CLex();**

**Lex2.strPSource = tbFSource2.Lines;**

**Lex2.strPMessage = tbFMessage1.Lines;**

**Lex2.intPSourceColSelection = 0;**

**Lex2.intPSourceRowSelection = 0;**

**int x2 = tbFSource2.TextLength;**

**tbFMessage1.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**int b = 0;**

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

**{**

**while (Lex2.enumPState != TState.Finish)**

**{**

**Lex2.NextToken();**

**switch (Lex2.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**int b1 = 0;**

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

**{**

**htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 0);**

**TablesToMemo(this, e);**

**}**

**tbFMessage1.AppendText("(Идентификатор был изменен)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**tbFMessage1.AppendText("(Нельзя заменить идентификатор на номер)");**

**break;**

**}**

**}**

**}**

**}**

**else tbFMessage1.AppendText("(Идентификатор для удаления не найден)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**int b = 0;**

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

**{**

**while (Lex2.enumPState != TState.Finish)**

**{**

**Lex2.NextToken();**

**switch (Lex2.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**tbFMessage1.AppendText("(Нельзя заменить номер на идентификатор)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**int b1 = 0;**

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

**{**

**htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 1);**

**TablesToMemo(this, e);**

**}**

**tbFMessage1.AppendText("(Номер был изменен)");**

**break;**

**}**

**}**

**}**

**}**

**else tbFMessage1.AppendText("(Номер для удаления не найден)");**

**break;**

**}**

**}**

**}**

**}**

**catch (Exception exc)**

**{**

**tbFMessage1.Text += exc.Message;**

**tbFSource.Select();**

**tbFSource.SelectionStart = 0;**

**int n1 = 0;**

**for (int i = 0; i < Lex.intPSourceRowSelection; i++) n1 += tbFSource.Lines[i].Length + 2;**

**n1 += Lex.intPSourceColSelection;**

**tbFSource.SelectionLength = n1;**

**}**

**}**

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

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource.Lines;**

**Lex.strPMessage = tbFMessage.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource.TextLength;**

**int y = tbFSource.Lines.Length;**

**tbFMessage.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**string s1 = "", s = "";**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**s1 = "id " + Lex.strPLexicalUnit; 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 button3\_Click(object sender, EventArgs e)**

**{**

**tbFMessage2.Clear();**

**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();**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**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:**

**{**

**int b = 0;**

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

**{**

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

**}**

**else tbFMessage2.AppendText("(Номер не найден)");**

**break;**

**}**

**}**

**}**

**}**

**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 button6\_Click(object sender, EventArgs e)**

**{**

**tbFMessage.Clear();**

**uSyntAnalyzer Synt = new uSyntAnalyzer();**

**Synt.Lex.strPSource = tbFSource.Lines;**

**Synt.Lex.strPMessage = tbFMessage.Lines;**

**Synt.Lex.enumPState = TState.Start;**

**try**

**{**

**Synt.Lex.NextToken();**

**Synt.S(OsnovnDerevo);**

**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;**

**}**

**Star(OsnovnDerevo);**

**treeView1.ExpandAll();**

**treeView2.ExpandAll();**

**}**

**public void potomki(TreeInMemory q, TreeNode w)**

**{**

**if (q.doch != null)**

**{**

**w.Nodes.Add(q.doch.Name + "");**

**potomki(q.doch, w.Nodes[0]);**

**Console.WriteLine(q.doch);**

**if (q.doch.sestra != null)**

**{**

**int r = 1;**

**TreeInMemory e = q.doch.sestra;**

**TreeNode d = w;**

**while (e != null)**

**{**

**d.Nodes.Add(e.Name + "");**

**potomki(e, d.Nodes[r]);**

**r++;**

**e = e.sestra;**

**}**

**}**

**}**

**}**

**private void convertBtn\_Click(object sender, EventArgs e)**

**{**

**TreeConverter treeConverter = new TreeConverter();**

**treeConverter.ConvertBinaryToDecimal(treeView1);**

**}**

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

**{**

**TreeConverter treeConverter = new TreeConverter();**

**treeConverter.ConvertBinaryToDecimal(treeView1);**

**}**

**private void ConverterSwapBtn\_Click(object sender, EventArgs e)**

**{**

**TreeConverterSwap treeConverter = new TreeConverterSwap();**

**treeConverter.ConvertBinaryToDecimal(treeView2);**

**}**

**private void reload\_Click(object sender, EventArgs e)**

**{**

**treeView1.Nodes.Clear();**

**treeView2.Nodes.Clear();**

**listBox1.Items.Clear();**

**listBox2.Items.Clear();**

**//listBox3.Items.Clear();**

**}**

**public void Star(TreeInMemory q)**

**{**

**treeView1.Nodes.Clear();**

**treeView2.Nodes.Clear();**

**if (q.doch != null)**

**{**

**treeView1.Nodes.Add("S");**

**treeView2.Nodes.Add("S");**

**potomki(q, treeView1.Nodes[0]);**

**potomki(q, treeView2.Nodes[0]);**

**}**

**}**

**}**

**}**

**THashTable.cs:**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace nsSynt

{

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;

// конструктор класса

public THashTable(ref THeap objAHeap)

{

objFHeap = objAHeap;

Init(7);

intFItemReserve = 0;

}

// инициализация хэш-таблицы

public void Init(int count)

{

arrFHashTable.Clear();

Resize(arrFHashTable, count);

intFCurrentPrimeNumber = count;

}

// Данный метод Resize изменяет размер списка list до size. Если size больше текущего размера списка, он добавляет пустые объекты в список, пока размер не станет равным size. Если size меньше текущего размера списка, он удаляет последние объекты из списка, пока размер не станет равным size.

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);

}

// Функция NextPrimeNumber принимает на вход число cardAOldPrimeNumber и возвращает следующее простое число

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

// функция вычисляет хэш-значение строки "strALexicalUnit", применяя операцию получения остатка от деления каждого символа на 12, и суммируя полученные значения

UInt32 HashFunction\_Wainberger(string strALexicalUnit)

{

UInt32 h = 0/\*, seed=131313\*/;

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

{

h += strALexicalUnit[i]; //хеширование

}

return h;

}

// Это функция ReHashFunction\_Line, которая применяется для рехеширования в линейном методе открытой адресации.

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); //рехеширование

}

} while (!boolVFinish);

return hi;

}

// Он использует две функции хеширования - HashFunction\_Wainberger и ReHashFunction\_Line - для вычисления индекса хеша. Индекс хеша сохраняется в переменную intFHashIndex.

public void HashIndex(string strALexicalUnit)

{

int h;

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

intFHashIndex = ReHashFunction\_Line(h, strALexicalUnit);

}

// Это функция TableReHashing(), которая выполняет перехеширование в хэш-таблице.

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);

}

// поиск лексической единицы

public bool SearchLexicalUnit(string strAlexicalUnit, ref int intALexicalCode)

{

HashIndex(strAlexicalUnit);

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

else

{

intALexicalCode = arrFHashTable[intFHashIndex];

return true;

}

}

// добавление лексической единицы

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;

}

}

// удаление лексической единицы

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();

}

}

}

// удаление элемента из таблицы по его лексическому коду

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; }

}

// игнорирует элементы с нулевым значением

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);

}

}

//HashTableList

public class CHashTableList

{

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

private bool boolFIsSaved;

public bool boolFIsLoaded;

private byte byteFTablesSize;

static THeap objFHeap = new THeap();

// конструктор класса

public CHashTableList(byte byteATableCount)

{

this.byteFTablesSize = byteATableCount;

objFHeap = new THeap();

Resize(arrFHashTableList, byteATableCount);

}

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

public byte GetTableNumber(int intALexicalCode)

{

return objFHeap.arrFHeapTable[intALexicalCode].byteFHashTable;

}

//кол-во элементов в списке

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; }

}

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

// 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;

}

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

}

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 WindowsFormsApp15

{

internal class TreeConverter

{

public void ConvertBinaryToDecimal(TreeView tree)

{

foreach (TreeNode node in tree.Nodes)

{

ConvertBinaryToDecimal(node);

}

}

private void ConvertBinaryToDecimal(TreeNode node)

{

if (node.Text == "A" || node.Text == "S")

{

if (node.Nodes.Count > 0)

{

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

if (IsBin(binaryValue))

{

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

node2.Text = decimalValue.ToString();

}

}

}

}

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;

}

}

}

**TreeConverterSwap.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp15

{

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 void ConvertBinaryToDecimal(TreeView tree)

{

foreach (TreeNode node in tree.Nodes)

{

ConvertBinaryToDecimal(node);

}

foreach (var item in myDict)

{

list.Add(item.Key);

strings.Add(item.Value);

}

foreach (TreeNode node in tree.Nodes)

{

Perebor(node);

}

}

private void ConvertBinaryToDecimal(TreeNode node)

{

if (node.Text == "A")

{

if (node.Nodes.Count == 1)

{

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

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 void Perebor(TreeNode node)

{

if (node.Nodes.Count > 0)

{

foreach (TreeNode Node in node.Nodes)

{

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);

}

}

}

}

**TreeInMemory.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApp15

{

public class TreeInMemory

{

public string Name;

public TreeInMemory doch;

public TreeInMemory sestra;

public void dobavitDoch(string novuzel)

{

doch = new TreeInMemory();

doch.Name = novuzel;

}

public void dobavitSestru(string novuzel)

{

sestra = new TreeInMemory();

sestra.Name = novuzel;

}

}

}

**uLex.cs:**

**using System;**

**using System.Collections.Generic;**

**using System.Text;**

**namespace nsLex**

**{**

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

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

**public enum TToken**

**{**

**mn, minus, plus, lxmIdentifier, lxmNumber, lxmUnknown, lxmEmpty,**

**lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmComma, lxmdt, lxmText, lxmtz, lxmr, lxmrs, lxmls**

**};**

**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 == '\*' || 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 == ';' || chrFSelection == ',' || chrFSelection == '[' || chrFSelection == ']' || chrFSelection == '=' || chrFSelection == ':') enumFSelectionCharType = TCharType.ReservedSymbol;**

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

**enumFState = TState.Continue;**

**}**

**}**

**private void TakeSymbol()**

**{**

**char[] c = { chrFSelection };**

**String s = new string(c);**

**strFLexicalUnit += s;**

**GetSymbol();**

**}**

**public void NextToken()**

**{**

**strFLexicalUnit = "";**

**if (enumFState == TState.Start)**

**{**

**intFSourceRowSelection = 0;**

**intFSourceColSelection = -1;**

**GetSymbol();**

**}**

**if (chrFSelection == '/')**

**{**

**GetSymbol();**

**if (chrFSelection == '/')**

**while (enumFSelectionCharType != TCharType.EndRow)**

**{**

**GetSymbol();**

**}**

**GetSymbol();**

**}**

**//Вариант 17**

**switch (enumFSelectionCharType)**

**{**

**case TCharType.Letter:**

**{**

**// a b c d**

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

**// B | | | |CFin|**

**// CFin |CFin|CFin|CFin|CFin|**

**A:**

**{**

**if (chrFSelection == 'c')**

**{**

**TakeSymbol();**

**goto B;**

**}**

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

**}**

**B:**

**{**

**if (chrFSelection == 'd')**

**{**

**TakeSymbol();**

**goto CFin;**

**}**

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

**}**

**CFin:**

**{**

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

**{**

**TakeSymbol();**

**goto CFin;**

**}**

**else**

**{**

**enumFToken = TToken.lxmIdentifier;**

**return;**

**}**

**}**

**}**

**if (chrFSelection == '/')**

**{**

**GetSymbol();**

**if (chrFSelection == '/')**

**while (enumFSelectionCharType != TCharType.EndRow)**

**{**

**GetSymbol();**

**}**

**GetSymbol();**

**}**

**case TCharType.Digit:**

**{**

**// 0 1**

**// A | 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 == '0')**

**{**

**TakeSymbol();**

**goto A;**

**}**

**else throw new Exception("Ожидался 0");**

**E:**

**if (chrFSelection == '0')**

**{**

**TakeSymbol();**

**goto FFin;**

**}**

**else throw new Exception("Ожидалась 0");**

**FFin:**

**if (chrFSelection == '0')**

**{**

**TakeSymbol();**

**goto G;**

**}**

**else if (enumFSelectionCharType != TCharType.Digit)**

**{**

**enumFToken = TToken.lxmNumber;**

**return;**

**}**

**else throw new Exception("Ожидалась 0");**

**G:**

**if (chrFSelection == '0')**

**{**

**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;**

**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;**

**}**

**break;**

**}**

**case TCharType.EndText:**

**{**

**enumFToken = TToken.lxmEmpty;**

**break;**

**}**

**}**

**}**

**}**

**}**

**uSyntAnalyzer.cs:**

using System;

using System.Collections.Generic;

using System.Text;

using System.Windows.Forms;

using nsLex;

using WindowsFormsApp15;

namespace nsSynt

{

class uSyntAnalyzer

{

public TreeInMemory VremenDerevo;

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 int indexLxmLeftParenth = 0;

public int indexLxmRightParenth = 0;

public uSyntAnalyzer()

{

strFSource = new String[0];

strFMessage = new String[0];

strPSource = new String[0];

strPMessage = new String[0];

VremenDerevo = new TreeInMemory();

}

public uSyntAnalyzer(TreeInMemory mainTree)

{

strFSource = new String[0];

strFMessage = new String[0];

strPSource = new String[0];

strPMessage = new String[0];

VremenDerevo = new TreeInMemory();

VremenDerevo = mainTree;

}

public void S(TreeInMemory el)

{

if (el == VremenDerevo) { el.Name = "Start"; }

if (Lex.enumPToken == TToken.lxmIdentifier)

{

el.dobavitDoch(Lex.strPLexicalUnit);

Lex.NextToken();

if (Lex.enumPToken != TToken.lxmEmpty)

{

el.doch.dobavitSestru("S");

S(el.doch.sestra);

}

else throw new Exception("ожидалось идентификатор, число или скобка");

if (Lex.enumPToken != TToken.lxmEmpty || Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmLeftParenth)

{

el.doch.sestra.dobavitSestru("A");

A(el.doch.sestra.sestra);

}

else throw new Exception("ожидалось число или скобка");

}

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

{

el.dobavitDoch("A");

A(el.doch);

}

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

{

throw new Exception("ожидалось скобка (");

}

}

public void A(TreeInMemory el)

{

if (Lex.enumPToken == TToken.lxmNumber)

{

el.dobavitDoch(Lex.strPLexicalUnit);

Lex.NextToken();

}

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

{

el.dobavitDoch("(");

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmRightParenth)

{

el.doch.dobavitSestru(")");

Lex.NextToken();

}

else if (Lex.enumPToken != TToken.lxmRightParenth)

{

el.doch.dobavitSestru("S");

S(el.doch.sestra);

if (Lex.enumPToken == TToken.lxmRightParenth)

{

el.doch.sestra.dobavitSestru(")");

Lex.NextToken();

}

else throw new Exception("не хватает скобки )");

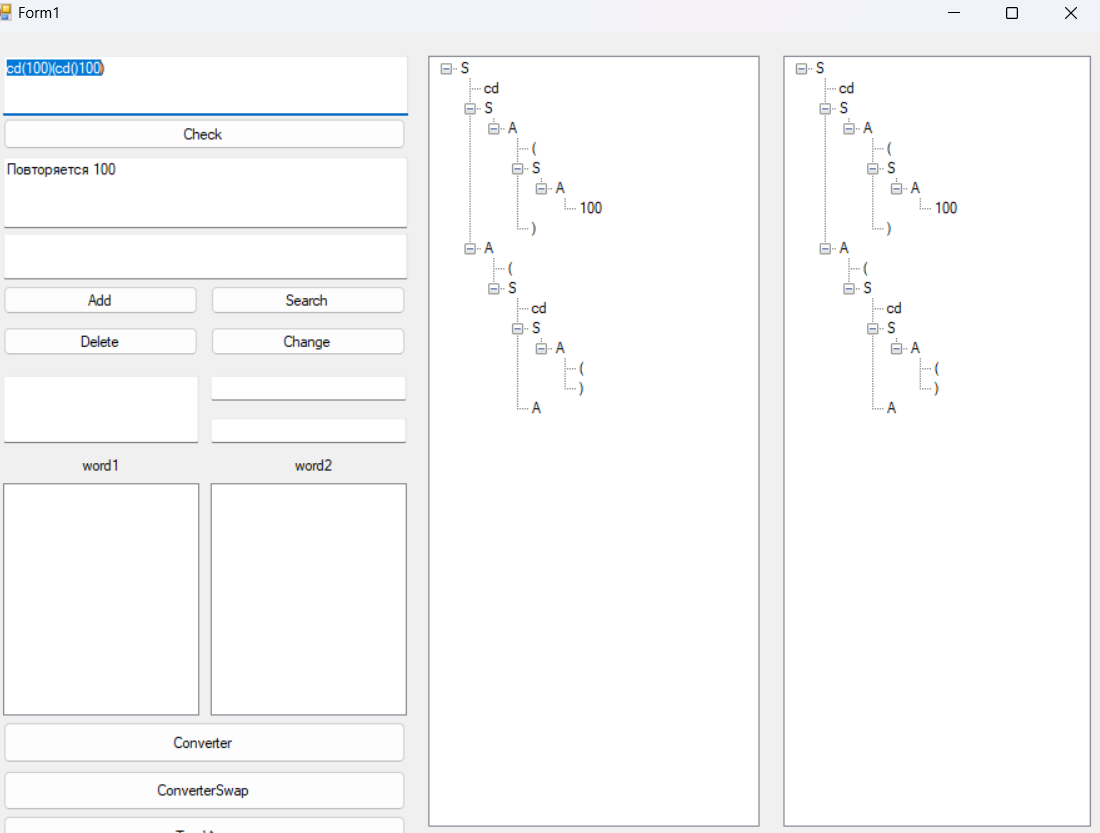
}

}

}

}

}



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

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

**Вариант:**

1)Слово начинается с:

cdc…

тогда: cdc…(100…) (число начинается с 100 в скобках)

2) Слово начинается с:

cdb…

тогда: cdb…(010100)

**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 анализатор использует 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).

**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 nsSynt;**

**using nsLex;**

**using static System.Windows.Forms.VisualStyles.VisualStyleElement;**

**namespace WindowsFormsApp15**

**{**

**public partial class Form1 : Form**

**{**

**public CHashTableList htl = new CHashTableList(3);**

**TreeInMemory OsnovnDerevo, Rez;**

**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();**

**}**

**public Form1()**

**{**

**InitializeComponent();**

**tbFSource.AppendText("cd(010100000)(cd(100)100000)");**

**int n = tbFSource.Lines.Length;**

**Rez = new TreeInMemory();**

**OsnovnDerevo = new TreeInMemory();**

**}**

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

**{**

**tbFMessage.Clear();**

**uSyntAnalyzer Synt = new uSyntAnalyzer();**

**Synt.Lex.strPSource = tbFSource.Lines;**

**Synt.Lex.strPMessage = tbFMessage.Lines;**

**Synt.Lex.enumPState = TState.Start;**

**try**

**{**

**Synt.Lex.NextToken();**

**Synt.S(Rez);**

**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 button4\_Click(object sender, EventArgs e)**

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource.Lines;**

**Lex.strPMessage = tbFMessage.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource.TextLength;**

**int y = tbFSource.Lines.Length;**

**tbFMessage.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**string s1 = "", s = "";**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**s1 = "id " + Lex.strPLexicalUnit;**

**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;**

**}**

**break;**

**}**

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

**tbFMessage.Text += m;**

**}**

**}**

**catch (Exception exc)**

**{**

**tbFMessage.Text += exc.Message;**

**tbFSource.Select();**

**tbFSource.SelectionStart = 0;**

**int n = 0;**

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

**n += Lex.intPSourceColSelection;**

**tbFSource.SelectionLength = n;**

**}**

**}**

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

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource1.Lines;**

**Lex.strPMessage = tbFMessage1.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource1.TextLength;**

**CLex Lex2 = new CLex();**

**Lex2.strPSource = tbFSource2.Lines;**

**Lex2.strPMessage = tbFMessage1.Lines;**

**Lex2.intPSourceColSelection = 0;**

**Lex2.intPSourceRowSelection = 0;**

**int x2 = tbFSource2.TextLength;**

**tbFMessage1.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**int b = 0;**

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

**{**

**while (Lex2.enumPState != TState.Finish)**

**{**

**Lex2.NextToken();**

**switch (Lex2.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**int b1 = 0;**

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

**{**

**htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 0);**

**TablesToMemo(this, e);**

**}**

**tbFMessage1.AppendText("(Идентификатор был изменен)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**tbFMessage1.AppendText("(Нельзя заменить идентификатор на номер)");**

**break;**

**}**

**}**

**}**

**}**

**else tbFMessage1.AppendText("(Идентификатор для удаления не найден)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**int b = 0;**

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

**{**

**while (Lex2.enumPState != TState.Finish)**

**{**

**Lex2.NextToken();**

**switch (Lex2.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**tbFMessage1.AppendText("(Нельзя заменить номер на идентификатор)");**

**break;**

**}**

**case TToken.lxmNumber:**

**{**

**int b1 = 0;**

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

**{**

**htl.DeleteLexicalUnit(Lex.strPLexicalUnit, 1);**

**TablesToMemo(this, e);**

**}**

**tbFMessage1.AppendText("(Номер был изменен)");**

**break;**

**}**

**}**

**}**

**}**

**else tbFMessage1.AppendText("(Номер для удаления не найден)");**

**break;**

**}**

**}**

**}**

**}**

**catch (Exception exc)**

**{**

**tbFMessage1.Text += exc.Message;**

**tbFSource.Select();**

**tbFSource.SelectionStart = 0;**

**int n1 = 0;**

**for (int i = 0; i < Lex.intPSourceRowSelection; i++) n1 += tbFSource.Lines[i].Length + 2;**

**n1 += Lex.intPSourceColSelection;**

**tbFSource.SelectionLength = n1;**

**}**

**}**

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

**{**

**CLex Lex = new CLex();**

**Lex.strPSource = tbFSource.Lines;**

**Lex.strPMessage = tbFMessage.Lines;**

**Lex.intPSourceColSelection = 0;**

**Lex.intPSourceRowSelection = 0;**

**int x = tbFSource.TextLength;**

**int y = tbFSource.Lines.Length;**

**tbFMessage.Text = "";**

**try**

**{**

**while (Lex.enumPState != TState.Finish)**

**{**

**Lex.NextToken();**

**string s1 = "", s = "";**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**s1 = "id " + Lex.strPLexicalUnit; 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 button3\_Click(object sender, EventArgs e)**

**{**

**tbFMessage2.Clear();**

**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();**

**switch (Lex.enumPToken)**

**{**

**case TToken.lxmIdentifier:**

**{**

**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:**

**{**

**int b = 0;**

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

**{**

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

**}**

**else tbFMessage2.AppendText("(Номер не найден)");**

**break;**

**}**

**}**

**}**

**}**

**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 button6\_Click(object sender, EventArgs e)**

**{**

**tbFMessage.Clear();**

**uSyntAnalyzer Synt = new uSyntAnalyzer();**

**Synt.Lex.strPSource = tbFSource.Lines;**

**Synt.Lex.strPMessage = tbFMessage.Lines;**

**Synt.Lex.enumPState = TState.Start;**

**try**

**{**

**Synt.Lex.NextToken();**

**Synt.S(OsnovnDerevo);**

**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;**

**}**

**Star(OsnovnDerevo);**

**treeView1.ExpandAll();**

**treeView2.ExpandAll();**

**}**

**public void potomki(TreeInMemory q, TreeNode w)**

**{**

**if (q.doch != null)**

**{**

**w.Nodes.Add(q.doch.Name + "");**

**potomki(q.doch, w.Nodes[0]);**

**Console.WriteLine(q.doch);**

**if (q.doch.sestra != null)**

**{**

**int r = 1;**

**TreeInMemory e = q.doch.sestra;**

**TreeNode d = w;**

**while (e != null)**

**{**

**d.Nodes.Add(e.Name + "");**

**potomki(e, d.Nodes[r]);**

**r++;**

**e = e.sestra;**

**}**

**}**

**}**

**}**

**private void convertBtn\_Click(object sender, EventArgs e)**

**{**

**TreeConverter treeConverter = new TreeConverter();**

**treeConverter.ConvertBinaryToDecimal(treeView1);**

**}**

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

**{**

**TreeConverter treeConverter = new TreeConverter();**

**treeConverter.ConvertBinaryToDecimal(treeView1);**

**}**

**private void ConverterSwapBtn\_Click(object sender, EventArgs e)**

**{**

**TreeConverterSwap treeConverter = new TreeConverterSwap();**

**treeConverter.ConvertBinaryToDecimal(treeView2);**

**}**

**private void reload\_Click(object sender, EventArgs e)**

**{**

**treeView1.Nodes.Clear();**

**treeView2.Nodes.Clear();**

**listBox1.Items.Clear();**

**listBox2.Items.Clear();**

**//listBox3.Items.Clear();**

**}**

**public void Star(TreeInMemory q)**

**{**

**treeView1.Nodes.Clear();**

**treeView2.Nodes.Clear();**

**if (q.doch != null)**

**{**

**treeView1.Nodes.Add("S");**

**treeView2.Nodes.Add("S");**

**potomki(q, treeView1.Nodes[0]);**

**potomki(q, treeView2.Nodes[0]);**

**}**

**}**

**}**

**}**

**THashTable.cs:**

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace nsSynt

{

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;

// конструктор класса

public THashTable(ref THeap objAHeap)

{

objFHeap = objAHeap;

Init(7);

intFItemReserve = 0;

}

// инициализация хэш-таблицы

public void Init(int count)

{

arrFHashTable.Clear();

Resize(arrFHashTable, count);

intFCurrentPrimeNumber = count;

}

// Данный метод Resize изменяет размер списка list до size. Если size больше текущего размера списка, он добавляет пустые объекты в список, пока размер не станет равным size. Если size меньше текущего размера списка, он удаляет последние объекты из списка, пока размер не станет равным size.

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);

}

// Функция NextPrimeNumber принимает на вход число cardAOldPrimeNumber и возвращает следующее простое число

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

// функция вычисляет хэш-значение строки "strALexicalUnit", применяя операцию получения остатка от деления каждого символа на 12, и суммируя полученные значения

UInt32 HashFunction\_Wainberger(string strALexicalUnit)

{

UInt32 h = 0/\*, seed=131313\*/;

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

{

h += strALexicalUnit[i]; //хеширование

}

return h;

}

// Это функция ReHashFunction\_Line, которая применяется для рехеширования в линейном методе открытой адресации.

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); //рехеширование

}

} while (!boolVFinish);

return hi;

}

// Он использует две функции хеширования - HashFunction\_Wainberger и ReHashFunction\_Line - для вычисления индекса хеша. Индекс хеша сохраняется в переменную intFHashIndex.

public void HashIndex(string strALexicalUnit)

{

int h;

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

intFHashIndex = ReHashFunction\_Line(h, strALexicalUnit);

}

// Это функция TableReHashing(), которая выполняет перехеширование в хэш-таблице.

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);

}

// поиск лексической единицы

public bool SearchLexicalUnit(string strAlexicalUnit, ref int intALexicalCode)

{

HashIndex(strAlexicalUnit);

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

else

{

intALexicalCode = arrFHashTable[intFHashIndex];

return true;

}

}

// добавление лексической единицы

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;

}

}

// удаление лексической единицы

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();

}

}

}

// удаление элемента из таблицы по его лексическому коду

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; }

}

// игнорирует элементы с нулевым значением

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);

}

}

//HashTableList

public class CHashTableList

{

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

private bool boolFIsSaved;

public bool boolFIsLoaded;

private byte byteFTablesSize;

static THeap objFHeap = new THeap();

// конструктор класса

public CHashTableList(byte byteATableCount)

{

this.byteFTablesSize = byteATableCount;

objFHeap = new THeap();

Resize(arrFHashTableList, byteATableCount);

}

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

public byte GetTableNumber(int intALexicalCode)

{

return objFHeap.arrFHeapTable[intALexicalCode].byteFHashTable;

}

//кол-во элементов в списке

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; }

}

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

// 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;

}

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

}

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 WindowsFormsApp15

{

internal class TreeConverter

{

public void ConvertBinaryToDecimal(TreeView tree)

{

foreach (TreeNode node in tree.Nodes)

{

ConvertBinaryToDecimal(node);

}

}

private void ConvertBinaryToDecimal(TreeNode node)

{

if (node.Text == "A" || node.Text == "S")

{

if (node.Nodes.Count > 0)

{

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

if (IsBin(binaryValue))

{

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

node2.Text = decimalValue.ToString();

}

}

}

}

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;

}

}

}

**TreeConverterSwap.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace WindowsFormsApp15

{

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 void ConvertBinaryToDecimal(TreeView tree)

{

foreach (TreeNode node in tree.Nodes)

{

ConvertBinaryToDecimal(node);

}

foreach (var item in myDict)

{

list.Add(item.Key);

strings.Add(item.Value);

}

foreach (TreeNode node in tree.Nodes)

{

Perebor(node);

}

}

private void ConvertBinaryToDecimal(TreeNode node)

{

if (node.Text == "A")

{

if (node.Nodes.Count == 1)

{

foreach (TreeNode node2 in node.Nodes)

{

string binaryValue = node2.Text;

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 void Perebor(TreeNode node)

{

if (node.Nodes.Count > 0)

{

foreach (TreeNode Node in node.Nodes)

{

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);

}

}

}

}

**TreeInMemory.cs:**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace WindowsFormsApp15

{

public class TreeInMemory

{

public string Name;

public TreeInMemory doch;

public TreeInMemory sestra;

public void dobavitDoch(string novuzel)

{

doch = new TreeInMemory();

doch.Name = novuzel;

}

public void dobavitSestru(string novuzel)

{

sestra = new TreeInMemory();

sestra.Name = novuzel;

}

}

}

**uLex.cs:**

**using System;**

**using System.Collections.Generic;**

**using System.Text;**

**namespace nsLex**

**{**

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

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

**public enum TToken**

**{**

**mn, minus, plus, lxmIdentifier, lxmNumber, lxmUnknown, lxmEmpty,**

**lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmComma, lxmdt, lxmText, lxmtz, lxmr, lxmrs, lxmls**

**};**

**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 == '\*' || 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 == ';' || chrFSelection == ',' || chrFSelection == '[' || chrFSelection == ']' || chrFSelection == '=' || chrFSelection == ':') enumFSelectionCharType = TCharType.ReservedSymbol;**

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

**enumFState = TState.Continue;**

**}**

**}**

**private void TakeSymbol()**

**{**

**char[] c = { chrFSelection };**

**String s = new string(c);**

**strFLexicalUnit += s;**

**GetSymbol();**

**}**

**public void NextToken()**

**{**

**strFLexicalUnit = "";**

**if (enumFState == TState.Start)**

**{**

**intFSourceRowSelection = 0;**

**intFSourceColSelection = -1;**

**GetSymbol();**

**}**

**if (chrFSelection == '/')**

**{**

**GetSymbol();**

**if (chrFSelection == '/')**

**while (enumFSelectionCharType != TCharType.EndRow)**

**{**

**GetSymbol();**

**}**

**GetSymbol();**

**}**

**//Вариант 17**

**switch (enumFSelectionCharType)**

**{**

**case TCharType.Letter:**

**{**

**// a b c d**

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

**// B | | | |CFin|**

**// CFin |CFin|CFin|CFin|CFin|**

**A:**

**{**

**if (chrFSelection == 'c')**

**{**

**TakeSymbol();**

**goto B;**

**}**

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

**}**

**B:**

**{**

**if (chrFSelection == 'd')**

**{**

**TakeSymbol();**

**goto CFin;**

**}**

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

**}**

**CFin:**

**{**

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

**{**

**TakeSymbol();**

**goto CFin;**

**}**

**else**

**{**

**enumFToken = TToken.lxmIdentifier;**

**return;**

**}**

**}**

**}**

**if (chrFSelection == '/')**

**{**

**GetSymbol();**

**if (chrFSelection == '/')**

**while (enumFSelectionCharType != TCharType.EndRow)**

**{**

**GetSymbol();**

**}**

**GetSymbol();**

**}**

**case TCharType.Digit:**

**{**

**// 0 1**

**// A | 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 == '0')**

**{**

**TakeSymbol();**

**goto A;**

**}**

**else throw new Exception("Ожидался 0");**

**E:**

**if (chrFSelection == '0')**

**{**

**TakeSymbol();**

**goto FFin;**

**}**

**else throw new Exception("Ожидалась 0");**

**FFin:**

**if (chrFSelection == '0')**

**{**

**TakeSymbol();**

**goto G;**

**}**

**else if (enumFSelectionCharType != TCharType.Digit)**

**{**

**enumFToken = TToken.lxmNumber;**

**return;**

**}**

**else throw new Exception("Ожидалась 0");**

**G:**

**if (chrFSelection == '0')**

**{**

**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;**

**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;**

**}**

**break;**

**}**

**case TCharType.EndText:**

**{**

**enumFToken = TToken.lxmEmpty;**

**break;**

**}**

**}**

**}**

**}**

**}**

**uSyntAnalyzer.cs:**

using System;

using System.Collections.Generic;

using System.Text;

using System.Windows.Forms;

using nsLex;

using WindowsFormsApp15;

namespace nsSynt

{

class uSyntAnalyzer

{

public TreeInMemory VremenDerevo;

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 int indexLxmLeftParenth = 0;

public int indexLxmRightParenth = 0;

public uSyntAnalyzer()

{

strFSource = new String[0];

strFMessage = new String[0];

strPSource = new String[0];

strPMessage = new String[0];

VremenDerevo = new TreeInMemory();

}

public uSyntAnalyzer(TreeInMemory mainTree)

{

strFSource = new String[0];

strFMessage = new String[0];

strPSource = new String[0];

strPMessage = new String[0];

VremenDerevo = new TreeInMemory();

VremenDerevo = mainTree;

}

public void S(TreeInMemory el)

{

if (el == VremenDerevo) { el.Name = "Start"; }

if (Lex.enumPToken == TToken.lxmIdentifier)

{

el.dobavitDoch(Lex.strPLexicalUnit);

Lex.NextToken();

if (Lex.enumPToken != TToken.lxmEmpty)

{

el.doch.dobavitSestru("S");

S(el.doch.sestra);

}

else throw new Exception("ожидалось идентификатор, число или скобка");

if (Lex.enumPToken != TToken.lxmEmpty || Lex.enumPToken == TToken.lxmNumber || Lex.enumPToken == TToken.lxmLeftParenth)

{

el.doch.sestra.dobavitSestru("A");

A(el.doch.sestra.sestra);

}

else throw new Exception("ожидалось число или скобка");

}

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

{

el.dobavitDoch("A");

A(el.doch);

}

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

{

throw new Exception("ожидалось скобка (");

}

}

public void A(TreeInMemory el)

{

if (Lex.enumPToken == TToken.lxmNumber)

{

el.dobavitDoch(Lex.strPLexicalUnit);

Lex.NextToken();

}

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

{

el.dobavitDoch("(");

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmRightParenth)

{

el.doch.dobavitSestru(")");

Lex.NextToken();

}

else if (Lex.enumPToken != TToken.lxmRightParenth)

{

el.doch.dobavitSestru("S");

S(el.doch.sestra);

if (Lex.enumPToken == TToken.lxmRightParenth)

{

el.doch.sestra.dobavitSestru(")");

Lex.NextToken();

}

else throw new Exception("не хватает скобки )");

}

}

}

}

}

