Министерство образования Российской Федерации

Федеральное агентство по образованию

КАЗАНСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

им. А.Н. Туполева - КАИ

Кафедра АСОИУ

Лабораторная работа №1-4

по дисциплине

«ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ»

Выполнил:

Студент группы 4309

Ильичев Д.А.

Проверил: Бикмуллина И.АИ.

Казань 2024

**Лабораторная работа № 1. Разработка транслитератора**

**Литера** — буква, один из знаков азбуки; в типографии наборная буква.

**Транслитератор** — это инструмент, который помогает переводить текст с одного языка на другой, сохраняя при этом структуру и порядок слов.

Существует два основных подхода к созданию транслитераторов: машинное обучение (ML) и основанный на правилах (rule-based).

Подход с использованием машинного обучения строится на вероятностных моделях последовательностей, таких как марковские модели. В основе этого метода лежит алгоритм Витерби: сегментация латинских слов и поиск наиболее вероятной последовательности скрытых состояний — кириллических комбинаций.

Этот метод можно рассматривать как задачу машинного перевода: создаётся таблица соответствий букв, и на её основе модель обучается вероятностным преобразованиям. ML требуется качественный датасет, который необходимо собрать.

Теперь перейдём к подходу, основанному на правилах. Для этого используются словари, таблицы сочетаемости букв.

**Как работает транслитератор:**

1. Анализируем входную последовательность символов.
2. Ищет комбинацию в заданном классе, словаре или в таблице транслитерации.
3. Если комбинация найдена, то преобразуем её и корректируем несочетаемости символов с помощью регулярных выражений.

Некоторые последовательности могут иметь неоднозначные интерпретации на русском языке. Для решения таких ситуаций создаются правила:

1. Преобразование CH. В начале слова это может быть Ч, Х или К, в конце — Ч или Х. Простой способ: если перед CH стоит И или Ы, то это преобразуется в КР, иначе — в ХР.
2. Разделительный твёрдый знак. Проверяем приставки, и, если следующий символ — йотированная гласная (Е, Ё, Ю, Я), ставим твёрдый знак перед ней.
3. Йотированные гласные. Они идут после приставок с твёрдым знаком или в начале слова. Ошибки возможны (например, IONY → ЁНЫ вместо ИОНЫ), но это можно минимизировать через словари, где для слов с ИО часто следуют буквы ТАД.
4. ИЙ или ЫЙ? Если перед ними стоят звуки ГДЖКЦЧШЩ, то используется ИЙ, иначе — ЫЙ.

Как можно улучшить эту модель:

1. Распознавать мягкий знак.
2. Использовать актуальный орфографический словарь для корректировки правил и их веса, что может постепенно привести к ML-подходу.
3. Добавить спеллчекер (система проверки правописания) для улучшения результатов транслитерации.

**Текст задания:**

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

**Код программы:**

**Form1.cs**

**using** System**;**

**using** System**.**Collections**.**Generic**;**

**using** System**.**ComponentModel**;**

**using** System**.**Data**;**

**using** System**.**Drawing**;**

**using** System**.**Text**;**

**using** System**.**Windows**.**Forms**;**

**namespace** Translator

**{**

**public** **partial** class Form1 **:** Form

**{**

**public** Form1**()**

**{**

InitializeComponent**();**

int n **=** tbFSource**.**Lines**.**Length**;** // Получение количества строк в tbFSource

**}**

// Обработчик события нажатия на кнопку btnFStart

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

**{**

// Создание экземпляра класса CLex для анализа текста

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

Lex**.**strPSource **=** tbFSource**.**Lines**;** // Устанавливаем источник текста из текстового поля tbFSource

Lex**.**strPMessage **=** tbFMessage**.**Lines**;** // Устанавливаем сообщения из текстового поля tbFMessage

int x **=** tbFSource**.**TextLength**;** // Получаем количество символов в tbFSource

int y **=** tbFSource**.**Lines**.**Length**;** // Получаем количество строк в tbFSource

tbFMessage**.**Text **=** ""**;** // Очищаем tbFMessage для нового результата

**try**

**{**

// Цикл продолжается, пока состояние парсера не станет "Finish"

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

**{**

Lex**.**GetSymbol**();** // Получение текущего символа из исходного текста

Lex**.**NextToken**();** // Получение следующего токена

String s **=** ""**;** // Переменная для хранения литеры

String s1 **=** ""**;** // Переменная для хранения типа литеры

// Определение типа символа и присваивание значений переменным s и s1

**switch** **(**Lex**.**enumFSelectionCharType**)**

**{**

**case** TCharType**.**EngLetter**:** **{** s1 **=** "EngLetter"**;** **break;** **}**

**case** TCharType**.**RusLetter**:** **{** s1 **=** "RusLetter"**;** **break;** **}**

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

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

**case** TCharType**.**Star**:** **{** s1 **=** "Star"**;** **break;** **}**

**case** TCharType**.**Exclamation**:** **{** s1 **=** "Exclamation"**;** **break;** **}**

**case** TCharType**.**Equal**:** **{** s1 **=** "Equal"**;** **break;** **}**

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

**case** TCharType**.**EndBracket**:** **{** s1 **=** "EndBracket"**;** **break;** **}**

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

**case** TCharType**.**OpenSquadBracket**:** **{** s1 **=** "OpenSquadBracket"**;** **break;** **}**

**case** TCharType**.**EndSquadBracket**:** **{** s1 **=** "EndSquadBracket"**;** **break;** **}**

**case** TCharType**.**Colon**:** **{** s1 **=** "Colon"**;** **break;** **}**

**case** TCharType**.**Minus**:** **{** s1 **=** "Minus"**;** **break;** **}**

**case** TCharType**.**Plus**:** **{** s1 **=** "Plus"**;** **break;** **}**

**case** TCharType**.**Comma**:** **{** s1 **=** "Comma"**;** **break;**

**case** TCharType**.**Dot**:** **{** s1 **=** "Dot"**;** **break;** **}**

**case** TCharType**.**AnotherSymbol**:** **{** s1 **=** "ReservedSymbol"**;** **break;** **}**

**case** TCharType**.**NoInd**:** **{** s1 **=** "NoInd"**;** **break;** **}**

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

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

**}**

// Создание строки с литерой и ее типом

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

// Добавление строки в tbFMessage

tbFMessage**.**Text **+=** m**;**

**}**

**}**

**catch** **(**Exception exc**)** // Обработка исключений

**{**

// Добавление сообщения об ошибке в tbFMessage

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

tbFSource**.**Select**();** // Устанавливаем фокус на tbFSource

tbFSource**.**SelectionStart **=** 0**;** // Устанавливаем начальную позицию выделения

int n **=** 0**;**

// Подсчет количества символов для выделения текста до текущей позиции

**for** **(**int i **=** 0**;** i **<** Lex**.**intPSourceRowSelection**;** i**++)**

n **+=** tbFSource**.**Lines**[**i**].**Length **+** 2**;** // +2 учитывает переход на новую строку

n **+=** Lex**.**intPSourceColSelection**;** // Добавляем текущую позицию в строке

tbFSource**.**SelectionLength **=** n**;** // Устанавливаем длину выделения

**}**

**}**

**}**

**}**

**uLex.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Translator

{

// Перечисление состояний анализа

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

// Перечисление возможных типов символов

public enum TCharType { EngLetter, RusLetter, Digit, EndRow, EndText, Space, Star, Slash, Exclamation, Equal, Semicolon, AnotherSymbol, OpenBracket, EndBracket, Colon, OpenSquadBracket, EndSquadBracket, Plus, Minus, Comma, Dot, NoInd }; // Тип символа

// Перечисление типов токенов

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

// Класс лексического анализатора

public class CLex

{

// Поля класса

private String[] strFSource; // Массив строк, представляющий исходный текст

private String[] strFMessage; // Массив строк для сообщений (возможно, для вывода результатов)

public TCharType enumFSelectionCharType; // Тип текущего символа

public char chrFSelection; // Текущий символ

private TState enumFState; // Текущее состояние анализатора

private int intFSourceRowSelection; // Номер текущей строки

private int intFSourceColSelection; // Номер текущей колонки в строке

private String strFLexicalUnit; // Текущая лексическая единица

private TToken enumFToken; // Текущий токен

// Свойства для доступа к полям класса

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; } }

// Конструктор класса

public CLex()

{

}

// Метод получения текущего символа из источника

public void GetSymbol()

{

// Проверяем, не вышли ли за пределы строки

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

{

intFSourceRowSelection++; // Переходим на следующую строку

if (intFSourceRowSelection <= strFSource.Length - 1)

{

// Если еще не конец текста, сбрасываем колонку и задаем символ конца строки

intFSourceColSelection = -1;

chrFSelection = '\0'; // Устанавливаем текущий символ в '\0'

enumFSelectionCharType = TCharType.EndRow; // Указываем, что достигнут конец строки

enumFState = TState.Continue; // Состояние анализа продолжается

}

else

{

// Если достигли конца всего текста

chrFSelection = '\0'; // Устанавливаем текущий символ в '\0'

enumFSelectionCharType = TCharType.EndText; // Указываем, что достигнут конец текста

enumFState = TState.Finish; // Меняем состояние на "Finish"

}

}

else

{

// Получаем текущий символ в строке

chrFSelection = strFSource[intFSourceRowSelection][intFSourceColSelection];

// Классифицируем символ

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

else if (chrFSelection >= 'a' && chrFSelection <= 'z') enumFSelectionCharType = TCharType.EngLetter;

else if (chrFSelection >= 'а' && chrFSelection <= 'я') enumFSelectionCharType = TCharType.RusLetter;

else if (chrFSelection >= '0' && chrFSelection <= '9') enumFSelectionCharType = TCharType.Digit;

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

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

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

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

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

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

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

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

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

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

else if (chrFSelection == '+') enumFSelectionCharType = TCharType.Plus;

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

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

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

else if (chrFSelection == '^' || chrFSelection == '%' || chrFSelection == '@' || chrFSelection == '<' || chrFSelection == '>' || chrFSelection == '?')

enumFSelectionCharType = TCharType.AnotherSymbol; // Считаем эти символы как другие (AnotherSymbol)

else enumFSelectionCharType = TCharType.NoInd; // Символ не распознан

enumFState = TState.Continue; // Продолжаем анализ

}

intFSourceColSelection++; // Переходим к следующему символу

}

// Метод добавления символа к текущей лексической единице

private void TakeSymbol()

{

char[] c = { chrFSelection }; // Создаем массив символов с текущим символом

String s = new string(c); // Преобразуем массив символов в строку

strFLexicalUnit += s; // Добавляем символ к текущей лексической единице

GetSymbol(); // Получаем следующий символ

}

// Метод перехода к следующему токену

public void NextToken()

{

strFLexicalUnit = ""; // Сбрасываем текущую лексическую единицу

// Начальная инициализация перед началом анализа

if (enumFState == TState.Start)

{

intFSourceRowSelection = 0; // Устанавливаем начальную строку

intFSourceColSelection = -1; // Устанавливаем начальный столбец

GetSymbol(); // Получаем первый символ

}

// Пропуск комментариев (если встречается '//')

if (chrFSelection == '/')

{

GetSymbol(); // Получаем следующий символ

if (chrFSelection == '/')

{

// Игнорируем все символы до конца строки

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

GetSymbol(); // Переходим к следующему символу после конца строки

}

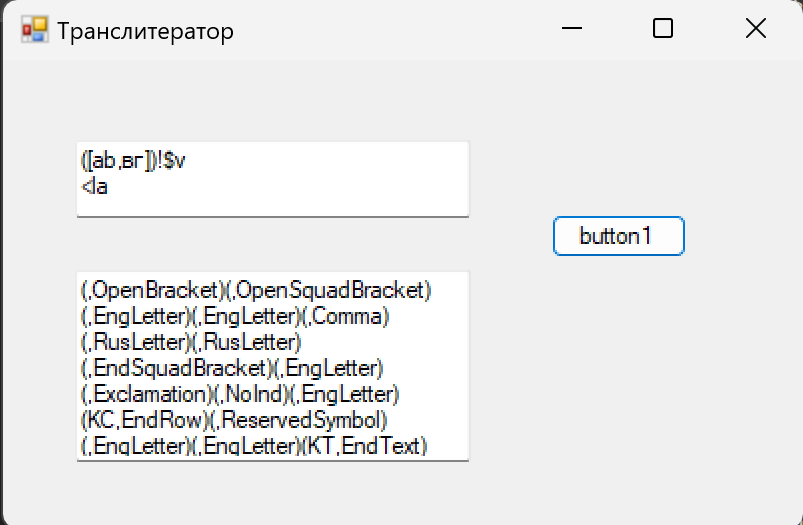
}

}

}

}

**Результаты тестирования:**



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

**1. Метод рекурсивного спуска:**

**Шаг 1. Инициализация**: установите входную строку и позицию чтения.

**Шаг 2. Определение токенов**: создайте функции для распознавания разных токенов (числа, идентификаторы и т.д.).

**Шаг 3. Сканирование**: вызывайте функции последовательно для нахождения токенов.

**Шаг 4. Обработка ошибок**: если токен не найден, выбросите ошибку.

**2. Метод конечных автоматов**

**Шаг 1. Определение состояний**: определите состояния для различных токенов.

**Шаг 2. Определение переходов**: установите правила переходов между состояниями.

**Шаг 3. Сканирование**: изменяйте состояния в зависимости от входных символов.

**Шаг 4. Обработка ошибок**: если перехода нет, выбросите ошибку.

**3. Метод регулярных выражений**

**Шаг 1. Определение шаблонов**: опишите регулярные выражения для токенов.

**Шаг 2. Сканирование**: используйте регулярные выражения для нахождения токенов в строке.

**Шаг 3. Обработка ошибок**: если совпадений нет, выбросите ошибку.

**Детерминированный конечный автомат (ДКА)** — это модель вычисления, которая в любой момент времени может находиться только в одном состоянии. Переходы между состояниями определяются детерминированно: для каждого состояния и входного символа есть ровно один переход.

**Недетерминированный конечный автомат (НДКА)** — это модель вычисления, которая может находиться в нескольких состояниях одновременно. Переходы между состояниями могут быть неопределёнными, то есть для одного состояния и входного символа, может быть, несколько возможных переходов, включая переходы по ε-переходам (переходам без чтения символа).

**Терминальный символ** — это символ, который принадлежит алфавиту языка и не может быть далее разложен на более простые символы. Эти символы являются конечными единицами входного текста, которые обрабатываются в процессе лексического анализа или транслитерации.

**Нетерминальный символ** — это символ, который используется в грамматике для обозначения групп символов или конструкций и может быть разложен на терминальные символы или другие нетерминальные символы. Нетерминальные символы помогают задавать структуру языка и используются для построения правил грамматики.

**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.

|  |  |  |
| --- | --- | --- |
| (001)\*101(110)\* | (a|b|c|d)+ | Длина не более 4 |

**Первое слово:**

(001)\*101(110)\*

A → 0B | 1C

B → 0D

C → 0Е

D → 1А

E → 1 | 1F

F → 1G

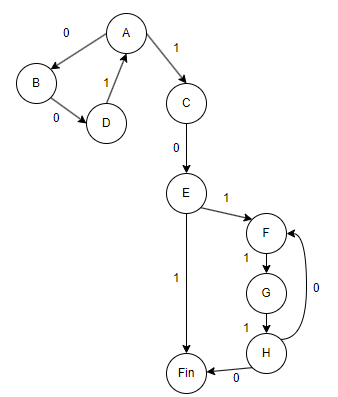
G → 1H

H → 0 | 0F

**Недетерминированная матрица:**

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

**Граф:**

****

**Детерминированная матрица:**

|  |  |  |
| --- | --- | --- |
|  | 0 | 1 |
| A | B | C |
| B | D |  |
| C | E |  |
| D |  | A |
| E |  | FFin |
| FFin |  | G |
| G |  | H |
| H | FFin |  |

**Второе слово:**

(a|b|c|d)+

Длина не более 4

A → a | b | c | d | aB | bB | cB | dB

B → a | b | c | d | aC | bC | cC | dC

C → a | b | c | d | aD | bD | cD | dD

D → a | b | c | d

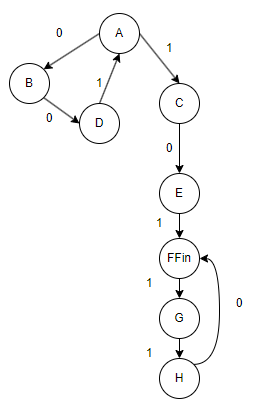
**Недетерминированная матрица:**

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

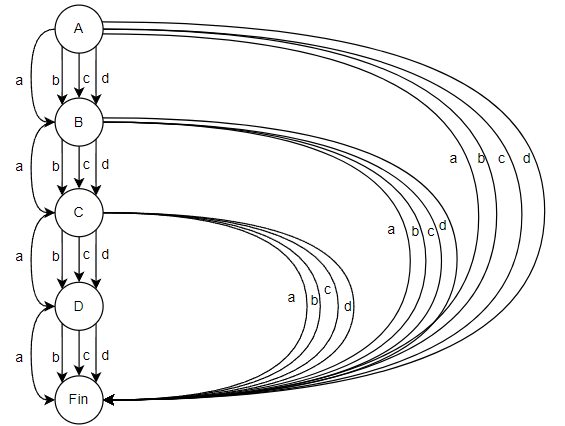
**Детерминированная матрица:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | a | b | c | d |
| A | BFin | BFin | BFin | BFin |
| BFin | CFin | CFin | CFin | CFin |
| CFin | D | D | D | D |
| D | Fin | Fin | Fin | Fin |
| Fin |  |  |  |  |

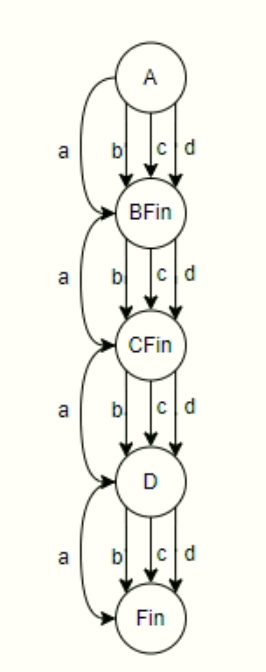
**Граф:**



**Граф:**



**Граф:**



**Код программы:**

**Form1.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

namespace Translator

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

int n = tbFSource.Lines.Length; // Получение количества строк в tbFSource

}

// Обработчик события нажатия на кнопку btnFStart

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

{

// Создание экземпляра класса CLex для анализа текста

CLex Lex = new CLex();

Lex.strPSource = tbFSource.Lines; // Устанавливаем источник текста из текстового поля tbFSource

Lex.strPMessage = tbFMessage.Lines; // Устанавливаем сообщения из текстового поля tbFMessage

int x = tbFSource.TextLength; // Получаем количество символов в tbFSource

int y = tbFSource.Lines.Length; // Получаем количество строк в tbFSource

tbFMessage.Text = ""; // Очищаем tbFMessage для нового результата

try

{

// Цикл продолжается, пока состояние парсера не станет "Finish"

while (Lex.enumPState != TState.Finish)

{

Lex.GetSymbol(); // Получение текущего символа из исходного текста

Lex.NextToken(); // Получение следующего токена

String s = ""; // Переменная для хранения литеры

String s1 = ""; // Переменная для хранения типа литеры

// Определение типа символа и присваивание значений переменным s и s1

//switch (Lex.enumFSelectionCharType)

//{

// case TCharType.EngLetter: { s1 = "EngLetter"; break; }

// case TCharType.RusLetter: { s1 = "RusLetter"; break; }

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

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

// case TCharType.Star: { s1 = "Star"; break; }

// case TCharType.Exclamation: { s1 = "Exclamation"; break; }

// case TCharType.Equal: { s1 = "Equal"; break; }

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

// case TCharType.EndBracket: { s1 = "EndBracket"; break; }

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

// case TCharType.OpenSquadBracket: { s1 = "OpenSquadBracket"; break; }

// case TCharType.EndSquadBracket: { s1 = "EndSquadBracket"; break; }

// case TCharType.Colon: { s1 = "Colon"; break; }

// case TCharType.Minus: { s1 = "Mius"; break; }

// case TCharType.Plus: { s1 = "Plus"; break; }

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

// case TCharType.Dot: { s1 = "Dot"; break; }

// case TCharType.AnotherSymbol: { s1 = "ReservedSymbol"; break; }

// case TCharType.NoInd: { s1 = "NoInd"; break; }

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

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

//}

switch (Lex.enumPToken)

{

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

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

}

// Создание строки с литерой и ее типом

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

// Добавление строки в tbFMessage

tbFMessage.Text += m;

}

}

catch (Exception exc) // Обработка исключений

{

// Добавление сообщения об ошибке в tbFMessage

tbFMessage.Text += exc.Message;

tbFSource.Select(); // Устанавливаем фокус на tbFSource

tbFSource.SelectionStart = 0; // Устанавливаем начальную позицию выделения

int n = 0;

// Подсчет количества символов для выделения текста до текущей позиции

for (int i = 0; i < Lex.intPSourceRowSelection; i++)

n += tbFSource.Lines[i].Length + 2; // +2 учитывает переход на новую строку

n += Lex.intPSourceColSelection; // Добавляем текущую позицию в строке

tbFSource.SelectionLength = n; // Устанавливаем длину выделения

}

}

}

}

**uLex.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Translator

{

// Перечисление состояний анализа

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

// Перечисление возможных типов символов

public enum TCharType { EngLetter, RusLetter, Digit, EndRow, EndText, Space, Star, Slash, Exclamation, Equal, Semicolon, AnotherSymbol, OpenBracket, EndBracket, Colon, OpenSquadBracket, EndSquadBracket, Plus, Minus, Comma, Dot, NoInd }; // Тип символа

// Перечисление типов токенов

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

// Класс лексического анализатора

public class CLex

{

// Поля класса

private String[] strFSource; // Массив строк, представляющий исходный текст

private String[] strFMessage; // Массив строк для сообщений (возможно, для вывода результатов)

public TCharType enumFSelectionCharType; // Тип текущего символа

public char chrFSelection; // Текущий символ

private TState enumFState; // Текущее состояние анализатора

private int intFSourceRowSelection; // Номер текущей строки

private int intFSourceColSelection; // Номер текущей колонки в строке

private String strFLexicalUnit; // Текущая лексическая единица

private TToken enumFToken; // Текущий токен

// Свойства для доступа к полям класса

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; } }

// Конструктор класса

public CLex()

{

}

// Метод получения текущего символа из источника

public void GetSymbol()

{

// Проверяем, не вышли ли за пределы строки

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

{

intFSourceRowSelection++; // Переходим на следующую строку

if (intFSourceRowSelection <= strFSource.Length - 1)

{

// Если еще не конец текста, сбрасываем колонку и задаем символ конца строки

intFSourceColSelection = -1;

chrFSelection = '\0'; // Устанавливаем текущий символ в '\0'

enumFSelectionCharType = TCharType.EndRow; // Указываем, что достигнут конец строки

enumFState = TState.Continue; // Состояние анализа продолжается

}

else

{

// Если достигли конца всего текста

chrFSelection = '\0'; // Устанавливаем текущий символ в '\0'

enumFSelectionCharType = TCharType.EndText; // Указываем, что достигнут конец текста

enumFState = TState.Finish; // Меняем состояние на "Finish"

}

}

else

{

// Получаем текущий символ в строке

chrFSelection = strFSource[intFSourceRowSelection][intFSourceColSelection];

// Классифицируем символ

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

else if (chrFSelection >= 'a' && chrFSelection <= 'z') enumFSelectionCharType = TCharType.EngLetter;

else if (chrFSelection >= 'а' && chrFSelection <= 'я') enumFSelectionCharType = TCharType.RusLetter;

else if (chrFSelection >= '0' && chrFSelection <= '9') enumFSelectionCharType = TCharType.Digit;

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

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

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

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

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

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

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

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

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

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

else if (chrFSelection == '+') enumFSelectionCharType = TCharType.Plus;

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

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

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

else if (chrFSelection == '^' || chrFSelection == '%' || chrFSelection == '@' || chrFSelection == '<' || chrFSelection == '>' || chrFSelection == '?')

enumFSelectionCharType = TCharType.AnotherSymbol; // Считаем эти символы как другие (AnotherSymbol)

else enumFSelectionCharType = TCharType.NoInd; // Символ не распознан

enumFState = TState.Continue; // Продолжаем анализ

}

intFSourceColSelection++; // Переходим к следующему символу

}

// Метод добавления символа к текущей лексической единице

private void TakeSymbol()

{

char[] c = { chrFSelection }; // Создаем массив символов с текущим символом

String s = new string(c); // Преобразуем массив символов в строку

strFLexicalUnit += s; // Добавляем символ к текущей лексической единице

GetSymbol(); // Получаем следующий символ

}

// Метод перехода к следующему токену

public void NextToken()

{

strFLexicalUnit = ""; // Сбрасываем текущую лексическую единицу

char[] allowedChars = { 'a', 'b', 'c', 'd' };

// Начальная инициализация перед началом анализа

if (enumFState == TState.Start)

{

intFSourceRowSelection = 0; // Устанавливаем начальную строку

intFSourceColSelection = -1; // Устанавливаем начальный столбец

GetSymbol(); // Получаем первый символ

}

// Пропуск комментариев (если встречается '//')

if (chrFSelection == '/')

{

GetSymbol(); // Получаем следующий символ

if (chrFSelection == '/')

{

// Игнорируем все символы до конца строки

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

GetSymbol(); // Переходим к следующему символу после конца строки

}

}

// Variant 13

switch (enumFSelectionCharType)

{

case TCharType.EngLetter:

{

// a b c d

// A |BFin|BFin|BFin|BFin|

// BFin |CFin|CFin|cFin|CFin|

// CFin | D | D | D | D |

// D |Fin | Fin| Fin| Fin|

// Fin | | | | |

A:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto BFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

BFin:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto CFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

CFin:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto D;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

D:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto Fin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

Fin:

{

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 == '0')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался 0");

C:

if (chrFSelection == '0')

{

TakeSymbol();

goto E;

}

else throw new Exception("Ожидался 0");

D:

if (chrFSelection == '1')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидалась 1");

E:

if (chrFSelection == '1')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 1");

FFin:

if (chrFSelection == '1')

{

TakeSymbol();

goto G;

}

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

else throw new Exception("Ожидалась 1");

G:

if (chrFSelection == '1')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался 0");

H:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидался 0");

}

case TCharType.AnotherSymbol:

{

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

{

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

}

GetSymbol();

}

if (chrFSelection == '(')

{

enumFToken = TToken.lxmLeftParenth;

GetSymbol();

return;

}

if (chrFSelection == ')')

{

enumFToken = TToken.lxmRightParenth;

GetSymbol();

return;

}

break;

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

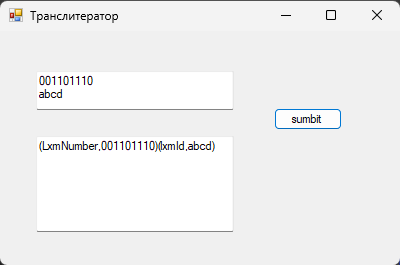
}

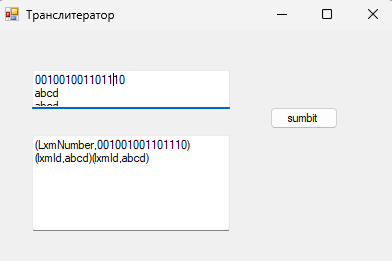
}

}

}

**Результаты тестирования:**

****

****

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

**Синтаксический анализатор** (или парсер) — это компонент компилятора или интерпретатора, который отвечает за анализ структуры входного текста (обычно исходного кода) с целью определения его синтаксической корректности согласно заданной грамматике.

**Левая рекурсия** — это ситуация в контекстно-свободной грамматике, когда определение нетерминального символа включает сам этот нетерминал как первый элемент в своей правой части.

**Избавление от левой рекурсии**

Для избавления от левой рекурсии мы преобразуем правила грамматики таким образом, чтобы рекурсия оказалась правой. Это можно сделать с использованием дополнительного нетерминала.

**Метод 1: Прямое преобразование**

**Шаг 1. Обнаружение**: Найти правила вида A → Aα | β.

**Шаг 2. Создание нового нетерминала**: Ввести A'.

**Шаг 3. Переписывание**:

* + A → β A'
  + A' → α A' | ε

**Метод 2:** **Факторизация**

**Шаг 1. Определение общего префикса:** Найдите правила, начинающиеся с одного префикса.

* + Пример: A → αβ | αγ | δ

**Шаг 2. Вынесение префикса:** Вынесите общий префикс, создав новый нетерминал.

* + Пример: A → αA' | δ и A' → β | γ

**Шаг 3. Проверка:** Убедитесь, что левой рекурсии больше нет.

**Шаг 4. Упрощение:** Пример до/после:

* + До: A → abcX | abcY | Z
  + После: A → abcA' | Z, A' → X | Y

**Метод 3:** **Предварительный анализ (Lookahead)**

**Шаг 1. Определение рекурсии:** Найдите рекурсивные правила, например: A → Aα | β.

**Шаг 2. Применение анализа:** Используйте k-символьный анализ для выбора правильного правила.

**Шаг 3. Избегание рекурсии:** Если встречен токен, соответствующий β, рекурсия избегается.

**Шаг 4. Построение дерева:** Парсер использует анализ на несколько символов вперед, устраняя левую рекурсию на этапе анализа.

**Текст задания:**

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

**Грамматика:**

S → S\_O | O

O → (SETQ\_V) | (COMMAND "LINE" <2><2>)

V → V\_C | C

С → <2><1>

**Освобождение от левой рекурсии:**

S → O | OA

A → \_O | \_A

O → (SETQ\_V) | (COMMAND "LINE" <2><2>)

V → С | CB

B → \_С | \_B

С → <2><1>

Простые предложения:

S – O – (SETQ V) → <2><1>

или

S – O – (COMMAND "LINE" <2><2>)

Пример: (SETQ abcd 101)

Сложные предложения

S – O – (SETQ V) → VС или С …

Пример (SETQ abcd 101 abcd 101 abcd 101)

**Код программы:**

**Form1.cs**

using nsSynt;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

namespace Translator

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

int n = tbFSource.Lines.Length; // Получение количества строк в tbFSource

}

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

{

tbFMessage.Clear();

uSyntAnalyzer Synt = new uSyntAnalyzer();

Synt.Lex.strPSource = tbFSource.Lines;

Synt.Lex.strPMessage = tbFMessage.Lines;

Synt.Lex.enumPState = TState.Start;

try

{

Synt.Lex.NextToken();

Synt.S();

throw new Exception("Текст верный");

}

catch (Exception exc) // Обработка исключений

{

// Добавление сообщения об ошибке в tbFMessage

tbFMessage.Text += exc.Message;

tbFSource.Select(); // Устанавливаем фокус на tbFSource

tbFSource.SelectionStart = 0; // Устанавливаем начальную позицию выделения

int n = 0;

// Подсчет количества символов для выделения текста до текущей позиции

for (int i = 0; i < Synt.Lex.intPSourceRowSelection; i++)

n += tbFSource.Lines[i].Length + 2; // +2 учитывает переход на новую строку

n += Synt.Lex.intPSourceColSelection; // Добавляем текущую позицию в строке

tbFSource.SelectionLength = n; // Устанавливаем длину выделения

}

}

}

}

**uLex.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Translator

{

// Перечисление состояний анализа

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

// Перечисление возможных типов символов

public enum TCharType { EngLetter, RusLetter, Digit, SETQ, CL, EndRow, EndText, Space, Star, Slash, Exclamation, Equal, Semicolon, AnotherSymbol, OpenBracket, EndBracket, Colon, OpenSquadBracket, EndSquadBracket, Plus, Minus, Comma, Dot, NoInd }; // Тип символа

// Перечисление типов токенов

public enum TToken { lxmIdentifier, lxmSETQ, lxmCL, lxmSpace, lxmOpenBracket, lxmEndBracket, lxmNumber, lxmUnknown, lxmEmpty, lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmComma, lxmText, lxmtz, lxmdt, lxmr, lxmrs, lxmls };

public class CLex

{

// Поля класса

private String[] strFSource; // Массив строк, представляющий исходный текст

private String[] strFMessage; // Массив строк для сообщений (возможно, для вывода результатов)

public TCharType enumFSelectionCharType; // Тип текущего символа

public char chrFSelection; // Текущий символ

private TState enumFState; // Текущее состояние анализатора

private int intFSourceRowSelection; // Номер текущей строки

private int intFSourceColSelection; // Номер текущей колонки в строке

private String strFLexicalUnit; // Текущая лексическая единица

private TToken enumFToken; // Текущий токен

// Свойства для доступа к полям класса

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; } }

// Конструктор класса

public CLex()

{

}

// Метод получения текущего символа из источника

public void GetSymbol()

{

// Проверяем, не вышли ли за пределы строки

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

{

intFSourceRowSelection++; // Переходим на следующую строку

if (intFSourceRowSelection <= strFSource.Length - 1)

{

// Если еще не конец текста, сбрасываем колонку и задаем символ конца строки

intFSourceColSelection = -1;

chrFSelection = '\0'; // Устанавливаем текущий символ в '\0'

enumFSelectionCharType = TCharType.EndRow; // Указываем, что достигнут конец строки

enumFState = TState.Continue; // Состояние анализа продолжается

}

else

{

// Если достигли конца всего текста

chrFSelection = '\0'; // Устанавливаем текущий символ в '\0'

enumFSelectionCharType = TCharType.EndText; // Указываем, что достигнут конец текста

enumFState = TState.Finish; // Меняем состояние на "Finish"

}

}

else

{

// Получаем текущий символ в строке

chrFSelection = strFSource[intFSourceRowSelection][intFSourceColSelection];

// Классифицируем символ

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

else if (chrFSelection >= 'a' && chrFSelection <= 'z') enumFSelectionCharType = TCharType.EngLetter;

else if (chrFSelection >= 'а' && chrFSelection <= 'я') enumFSelectionCharType = TCharType.RusLetter;

else if (chrFSelection >= '0' && chrFSelection <= '9') enumFSelectionCharType = TCharType.Digit;

else if (chrFSelection == 'S' || chrFSelection == 'E' || chrFSelection == 'T' || chrFSelection == 'Q') enumFSelectionCharType = TCharType.SETQ;

else if (chrFSelection == 'C' || chrFSelection == 'E' || chrFSelection == 'O' || chrFSelection == 'M' || chrFSelection == 'A' || chrFSelection == 'N' || chrFSelection == 'D' || chrFSelection == 'L' || chrFSelection == 'I' || chrFSelection == '"') enumFSelectionCharType = TCharType.CL;

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

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

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

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

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

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

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

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

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

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

else if (chrFSelection == '+') enumFSelectionCharType = TCharType.Plus;

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

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

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

else if (chrFSelection == '^' || chrFSelection == '%' || chrFSelection == '@' || chrFSelection == '<' || chrFSelection == '>' || chrFSelection == '?')

enumFSelectionCharType = TCharType.AnotherSymbol; // Считаем эти символы как другие (AnotherSymbol)

else enumFSelectionCharType = TCharType.NoInd; // Символ не распознан

enumFState = TState.Continue; // Продолжаем анализ

}

intFSourceColSelection++; // Переходим к следующему символу

}

// Метод добавления символа к текущей лексической единице

private void TakeSymbol()

{

char[] c = { chrFSelection }; // Создаем массив символов с текущим символом

String s = new string(c); // Преобразуем массив символов в строку

strFLexicalUnit += s; // Добавляем символ к текущей лексической единице

GetSymbol(); // Получаем следующий символ

}

// Метод перехода к следующему токену

public void NextToken()

{

strFLexicalUnit = ""; // Сбрасываем текущую лексическую единицу

char[] allowedChars = { 'a', 'b', 'c', 'd' };

// Начальная инициализация перед началом анализа

if (enumFState == TState.Start)

{

intFSourceRowSelection = 0; // Устанавливаем начальную строку

intFSourceColSelection = 0; // Устанавливаем начальный столбец

GetSymbol(); // Получаем первый символ

}

// Пропуск комментариев (если встречается '//')

if (chrFSelection == '/')

{

GetSymbol(); // Получаем следующий символ

if (chrFSelection == '/')

{

// Игнорируем все символы до конца строки

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

GetSymbol(); // Переходим к следующему символу после конца строки

}

}

// Variant 13

switch (enumFSelectionCharType)

{

case TCharType.EngLetter:

{

// a b c d

// A |BFin|BFin|BFin|BFin|

// BFin |CFin|CFin|cFin|CFin|

// CFin | D | D | D | D |

// D |Fin | Fin| Fin| Fin|

// Fin | | | | |

A:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto BFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

BFin:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto CFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

CFin:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto D;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

D:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto Fin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

Fin:

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

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 == '0')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался 0");

C:

if (chrFSelection == '0')

{

TakeSymbol();

goto E;

}

else throw new Exception("Ожидался 0");

D:

if (chrFSelection == '1')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидалась 1");

E:

if (chrFSelection == '1')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 1");

FFin:

if (chrFSelection == '1')

{

TakeSymbol();

goto G;

}

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

else throw new Exception("Ожидалась 1");

G:

if (chrFSelection == '1')

{

TakeSymbol();

goto H;

}

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

H:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидался 0");

}

// Создание токена служебного слова SETQ

case TCharType.SETQ:

{

A:

if (chrFSelection == 'S')

{

TakeSymbol();

goto B;

}

else throw new Exception("Ожидался S");

B:

if (chrFSelection == 'E')

{

TakeSymbol();

goto C;

}

else throw new Exception("Ожидался E");

C:

if (chrFSelection == 'T')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался T");

D:

if (chrFSelection == 'Q')

{

TakeSymbol();

goto Fin;

}

else throw new Exception("Ожидался Q");

Fin:

{

enumFToken = TToken.lxmSETQ;

return;

}

}

// Создание токена служебного слова COMMAND "LINE"

case TCharType.CL:

{

A:

if (chrFSelection == 'C')

{

TakeSymbol();

goto B;

}

else throw new Exception("Ожидался C");

B:

if (chrFSelection == 'O')

{

TakeSymbol();

goto C;

}

else throw new Exception("Ожидался O");

C:

if (chrFSelection == 'M')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался M");

D:

if (chrFSelection == 'M')

{

TakeSymbol();

goto E;

}

else throw new Exception("Ожидался M");

E:

if (chrFSelection == 'A')

{

TakeSymbol();

goto F;

}

else throw new Exception("Ожидался A");

F:

if (chrFSelection == 'N')

{

TakeSymbol();

goto G;

}

else throw new Exception("Ожидался N");

G:

if (chrFSelection == 'D')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался D");

H:

if (chrFSelection == '"')

{

TakeSymbol();

goto I;

}

else throw new Exception("Ожидался \"");

I:

if (chrFSelection == 'L')

{

TakeSymbol();

goto J;

}

else throw new Exception("Ожидался L");

J:

if (chrFSelection == 'I')

{

TakeSymbol();

goto K;

}

else throw new Exception("Ожидался I");

K:

if (chrFSelection == 'N')

{

TakeSymbol();

goto L;

}

else throw new Exception("Ожидался N");

L:

if (chrFSelection == 'E')

{

TakeSymbol();

goto M;

}

else throw new Exception("Ожидался E");

M:

if (chrFSelection == '"')

{

TakeSymbol();

goto Fin;

}

else throw new Exception("Ожидался \"");

Fin:

{

enumFToken = TToken.lxmCL;

return;

}

}

case TCharType.AnotherSymbol:

{

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

{

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

}

GetSymbol();

}

break;

}

// Добавление токенов

case TCharType.Space:

if (chrFSelection == ' ')

{

enumFToken = TToken.lxmSpace;

GetSymbol();

return;

}

break ;

case TCharType.OpenBracket:

if (chrFSelection == '(')

{

enumFToken = TToken.lxmOpenBracket;

GetSymbol();

return;

}

break ;

case TCharType.EndBracket:

if (chrFSelection == ')')

{

enumFToken = TToken.lxmEndBracket;

GetSymbol();

return;

}

break ;

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

}

}

}

}

**uSyntAnalyzer.cs**

using System;

using System.Collections.Generic;

using System.Text;

using Translator;

namespace nsSynt

{

class uSyntAnalyzer

{

private String[] strFSource;

private String[] strFMessage;

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

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

public CLex Lex = new CLex();

public void S()

{

O(); // разбор правила O

if (Lex.enumPToken == TToken.lxmSpace)

A(); // если есть пробелы, вызываем A

}

public void A()

{

if (Lex.enumPToken == TToken.lxmSpace) // разбор пробела

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmSETQ || Lex.enumPToken == TToken.lxmCL)

O(); // после пробела ожидается O

else

A(); // или продолжаем разбор пробела

}

}

public void O()

{

if (Lex.enumPToken == TToken.lxmOpenBracket) // проверяем открывающую скобку

{

Lex.NextToken(); // переходим к следующему токену после скобки

if (Lex.enumPToken == TToken.lxmSETQ) // разбор SETQ

{

Lex.NextToken(); // переходим после SETQ

if (Lex.enumPToken == TToken.lxmSpace) // проверка на пробел

{

Lex.NextToken();

V(); // после пробела ожидается V (переменная или список)

if (Lex.enumPToken == TToken.lxmEndBracket) // проверка на закрывающую скобку

{

Lex.NextToken(); // завершаем разбор SETQ

}

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

}

else throw new Exception("Ожидался пробел после SETQ");

}

else if (Lex.enumPToken == TToken.lxmCL) // разбор COMMAND"LINE"

{

Lex.NextToken(); // переходим к следующему токену после COMMAND

if (Lex.enumPToken == TToken.lxmSpace) // проверка на пробел

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmIdentifier) // ожидаем первый индефикатор

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmSpace) // проверка на пробел

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmIdentifier) // ожидаем второй индефикатор

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmEndBracket) // проверка на закрывающую скобку

{

Lex.NextToken(); // завершаем разбор COMMAND "LINE"

}

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

}

else throw new Exception("Ожидался второй индефикатор для LINE");

}

else throw new Exception("Ожидался пробел после первого индефикатора");

}

else throw new Exception("Ожидался первый индефикатор для LINE");

}

else throw new Exception("Ожидался пробел после COMMAND\"LINE\"");

}

else throw new Exception("Ожидался SETQ или COMMAND");

}

else throw new Exception("Ожидалась открывающая скобка");

}

public void V()

{

C(); // разбор C как часть V

if (Lex.enumPToken == TToken.lxmSpace) // проверяем на пробел для B

B(); // если есть пробел, вызываем B

}

public void B()

{

if (Lex.enumPToken == TToken.lxmSpace) // разбор пробела

{

Lex.NextToken();

C(); // вызываем C после пробела

if (Lex.enumPToken == TToken.lxmSpace) // продолжаем разбор пробела

B(); // продолжаем разбор B

}

}

public void C()

{

if (Lex.enumPToken == TToken.lxmIdentifier) // ожидаем индефикатор

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmSpace)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber) // ожидаем число

Lex.NextToken();

else throw new Exception("Ожидался числовое значение");

}

else throw new Exception("Ожидался пробел");

}

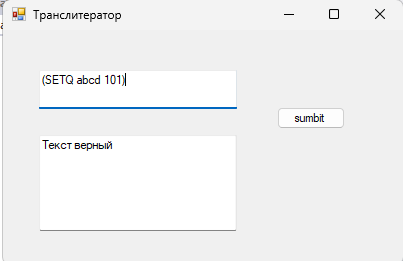
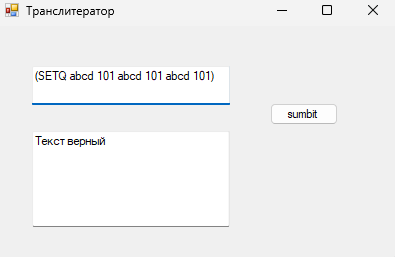
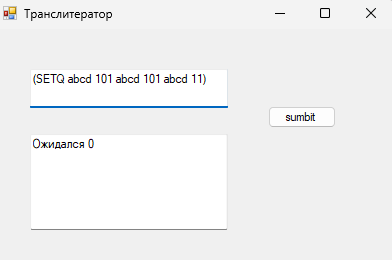
else throw new Exception("Ожидалось индефикатор");

}

}

}

**Результаты тестирования:**

**** **** 

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

**Хеширование** — это способ проверки данных, например, для паролей или блокчейна. Хеш — уникальный код для набора данных, по которому можно быстро проверить соответствие информации. Вместо хранения самих данных сохраняют их хеш, что помогает защитить данные и экономить память.

**Хеш-таблица** — структура данных, где хранятся пары "ключ-значение". Она позволяет добавлять, искать и удалять данные по ключу. Есть два типа хеш-таблиц: с цепочками и с открытой адресацией.

**Хеш-функция** преобразует данные в строку фиксированной длины. Коллизии (когда разные данные попадают в одну ячейку) — обычное явление. Для их решения используют цепочки или открытую адресацию.

**Метод цепочек** хранит несколько элементов в одной ячейке как список. Проблема возникает, если списки становятся слишком длинными, поэтому таблица автоматически увеличивается при необходимости.

**Метод открытой адресации** размещает элементы прямо в ячейках. Если ячейка занята, ищется следующая свободная по определённому алгоритму.

**Метод 1. SHA256:**

**Шаг 1. Преобразование строки в байты:**

Строка, переданная в метод, преобразуется в массив байт, используя кодировку UTF-8.

**Шаг 2. Вычисление хэша:**

Алгоритм SHA256 используется для вычисления хэша из байтов строки.

**Шаг 3. Преобразование результата в строку:**

Полученный хэш (массив байт) переводится в строку в шестнадцатеричном формате.

**Шаг 4. Возврат хэша:**

Результирующая строка хэша возвращается как итог работы метода.

**Метод 2. Алгоритм хеширования Вайнбергера**:

**Шаг 1. Инициализация переменной хеша:**

Переменная h инициализируется как 0. Переменная g используется для промежуточных вычислений.

**Шаг 2. Цикл по каждому символу строки:**

Проход по каждому символу строки:

* + h сдвигается влево на 4 бита и добавляется значение текущего символа.
  + Присваивается значение переменной g — это старшие 4 бита из переменной h (побитовое AND с маской 0xF0000000).

**Шаг 3. Устранение старших битов:**

Если g не равен 0:

* + Выполняется побитовая операция XOR между значением h, сдвинутым на 24 бита вправо, и значением g. Это убирает старшие биты, которые могут вызывать коллизии.

**Шаг 4. Возврат значения хеша:**

После прохождения всех символов строки возвращается значение хеша h.

**Метод 3.** **Алгоритм DJB2:**

**Шаг 1. Инициализация хеша:**

Устанавливаем начальное значение хеша как 5381.

**Шаг 2. Цикл по символам строки:**

Для каждого символа строки:

Умножаем текущее значение хеша на 33 (сдвиг влево на 5 бит и сложение с текущим значением).

Добавляем ASCII значение текущего символа.

**Шаг 3. Возврат хеша:**

После обработки всех символов строки возвращаем конечное значение хеша.

Методы избавления коллизии:

|  |  |  |
| --- | --- | --- |
| **Метод** | **Плюсы** | **Минусы** |
| **Метод цепочек (Chaining)** | - Простота реализации.  - Позволяет легко хранить значения с одинаковым хэшем.  - Размер таблицы может быть меньше, чем количество элементов. | - Меньшая производительность при большом количестве коллизий.  - Необходимо выделять память для списков, что может привести к фрагментации памяти.  - Возможные накладные расходы на управление списками. |
| **Открытая адресация (Open Addressing)** | - Все элементы хранятся в одном массиве, что упрощает управление памятью.  - Быстрый доступ к элементам без дополнительных структур данных. | - Необходимо поддерживать размер массива больше, чем количество элементов, чтобы избежать большого числа коллизий.  - Падение производительности при заполнении хэш-таблицы.  - Усложненная реализация с необходимостью поиска следующей свободной ячейки. |
| **Метод уникального ключа** | |  | | --- | | - Полностью устраняет коллизии, так как каждый элемент имеет уникальный ключ.  - Простота добавления и поиска слов с уникальными ключами. | | |  | | --- | | - Увеличение длины ключа может привести к большему использованию памяти.  - Увеличение сложности хэш-таблицы, так как потребуется проверка уникальности для каждого добавляемого слова.  - Потенциальные проблемы с производительностью при большом количестве элементов. | |

**Текст задания:**

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

**Код программы:**

**MyHash.cs**

using System;

using System.Collections.Generic;

using System.Security.Cryptography;

using System.Text;

namespace laba4

{

public class MyHash

{

// Метод для вычисления хэша SHA256

public string HashFunction(string word)

{

using (SHA256 sha256Hash = SHA256.Create())

{

// Преобразуем строку в байты и вычисляем хэш

byte[] bytes = sha256Hash.ComputeHash(Encoding.UTF8.GetBytes(word));

// Преобразуем байты в строку в шестнадцатеричном формате

StringBuilder builder = new StringBuilder();

foreach (byte b in bytes)

builder.Append(b.ToString("x2"));

return builder.ToString();

}

}

// Метод для добавления слова в хэш-таблицу

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

{

string hashValue = HashFunction(word); // вычисляется хэш-код для слова

string uniqueKey = hashValue + "\_" + word;

// Проверяем, содержится ли уже такой хэш-код в хэш-таблице

if (!hashTable.ContainsKey(uniqueKey))

hashTable[uniqueKey] = new List<string>(); // в таблице создается новая пустая коллекция

else

{

if (hashTable[uniqueKey].Contains(word)) // проверяем, содержится ли уже переданное слово

return;

}

hashTable[uniqueKey].Add(word); // добавляем слово в коллекцию

}

// Метод для поиска слова в хэш-таблице

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

{

string hashValue = HashFunction(word);

string uniqueKey = hashValue + "\_" + word;

if (hashTable.ContainsKey(uniqueKey)) // проверяем, содержится ли такой хэш-код

return hashTable[uniqueKey].Contains(word); // проверяем, содержится ли слово в коллекции

return false; // хэш-код не найден

}

// Метод для удаления слова из хэш-таблицы

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

{

string hashValue = HashFunction(word);

string uniqueKey = hashValue + "\_" + word;

if (hashTable.ContainsKey(uniqueKey)) // проверяем наличие в хэш-таблице

{

List<string> words = hashTable[uniqueKey];

if (words.Contains(word))

{

words.Remove(word); // если слово найдено, оно удаляется

if (words.Count == 0) // если коллекция становится пустой

hashTable.Remove(uniqueKey); // удаляем ее из хэш-таблицы

return true;

}

}

return false; // слово не найдено

}

}

}

**Form1.cs**

using nsSynt;

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Text;

using System.Windows.Forms;

using laba4;

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

namespace Translator

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

int n = tbFSource.Lines.Length; // Получение количества строк в tbFSource

}

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

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

Dictionary<string, List<string>> hashTableSpecial = new Dictionary<string, List<string>>();

public MyHash hashFunction = new MyHash();

private bool isFirstButtonValid = false;

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

{

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

listBox1.Items.Clear();

listBox2.Items.Clear();

listBox3.Items.Clear();

foreach (var entry in hashTableIdentifier)

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

listTable.Clear();

foreach (var entry in hashTableDigital)

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

listTable.Clear();

foreach (var entry in hashTableSpecial)

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

listTable.Clear();

}

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

{

tbFMessage.Clear();

uSyntAnalyzer Synt = new uSyntAnalyzer();

Synt.Lex.strPSource = tbFSource.Lines;

Synt.Lex.strPMessage = tbFMessage.Lines;

Synt.Lex.enumPState = TState.Start;

Synt.Lex.intPSourceRowSelection = 0;

Synt.Lex.intPSourceColSelection = 0;

isFirstButtonValid = false;

try

{

for (int i = 0; i < Synt.Lex.strPSource.Length; i++) // Обрабатываем каждую строку

{

string line = Synt.Lex.strPSource[i]; // Получаем текущую строку

Synt.Lex.NextToken(); // Получаем следующий токен

Synt.S(); // Проверяем синтаксис текущей строки

// Переход на следующую строку после завершения анализа одной строки

if (Synt.Lex.enumPState == TState.Continue && i < Synt.Lex.strPSource.Length - 1)

Synt.Lex.enumPState = TState.Start; // Возвращаем состояние в начало

}

tbFMessage.Text = "Текст верный"; // Успешное завершение

}

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;

// Проверяем, является ли сообщение "Текст верный"

if (exc.Message == "Текст верный")

{

isFirstButtonValid = true; // В случае успеха устанавливаем true

writeButton.Enabled = true; // Разблокируем кнопку

}

else

{

isFirstButtonValid = false; // В случае ошибки флаг остается false

writeButton.Enabled = false; // Отключаем кнопку writeButton

}

}

}

private void writeButton\_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 = "";

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.lxmEndBracket:

{

hashFunction.AddWord(hashTableSpecial, ")");

s1 = "spec ) " + Lex.strPLexicalUnit;

TablesToMemo(this, e);

break;

}

case TToken.lxmOpenBracket:

{

hashFunction.AddWord(hashTableSpecial, "(");

s1 = "spec ( " + Lex.strPLexicalUnit;

TablesToMemo(this, e);

break;

}

case TToken.lxmCL:

{

hashFunction.AddWord(hashTableSpecial, "COMMAND\"LINE\"");

s1 = "spec " + Lex.strPLexicalUnit;

TablesToMemo(this, e);

break;

}

case TToken.lxmSETQ:

{

hashFunction.AddWord(hashTableSpecial, "SETQ");

s1 = "spec " + Lex.strPLexicalUnit;

TablesToMemo(this, e);

break;

}

}

String m = "(" + 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 deleteButton\_Click(object sender, EventArgs e)

{

ListBox selectedListBox = GetSelectedListBox(); // Определяем выбранный ListBox

Dictionary<string, List<string>> selectedHashTable = GetSelectedHashTable(selectedListBox); // Получаем нужную хэш-таблицу

if (selectedListBox != null && selectedHashTable != null && selectedListBox.SelectedItem != null)

{

string selectedItem = selectedListBox.SelectedItem.ToString();

// Удаляем выбранное слово из соответствующей хэш-таблицы

if (hashFunction.RemoveWord(selectedHashTable, selectedItem))

deleteButton.BackColor = Color.Green;

else

deleteButton.BackColor = Color.Red;

}

else

deleteButton.BackColor = Color.Red;

}

private void reloadButton\_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 hashTableSpecial)

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

}

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

{

ListBox selectedListBox = GetSelectedListBox();

Dictionary<string, List<string>> selectedHashTable = GetSelectedHashTable(selectedListBox); // Метод для получения нужной хэш-таблицы

if (selectedHashTable != null && selectedListBox.SelectedItem != null)

{

string selectedItem = selectedListBox.SelectedItem.ToString();

if (hashFunction.FindhWord(selectedHashTable, selectedItem))

findButton.BackColor = Color.Green;

else

findButton.BackColor = Color.Red;

}

}

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

{

ListBox selectedListBox = GetSelectedListBox(); // Метод для определения выбранного ListBox

Dictionary<string, List<string>> selectedHashTable = GetSelectedHashTable(selectedListBox); // Метод для получения нужной хэш-таблицы

if (selectedListBox != null && selectedHashTable != null && selectedListBox.SelectedItem != null)

{

string selectedItem = selectedListBox.SelectedItem.ToString();

// Удаляем выбранное слово

if (hashFunction.RemoveWord(selectedHashTable, selectedItem))

{

// Добавляем новое слово из textBox1

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

changeButton.BackColor = Color.Green;

}

else

changeButton.BackColor = Color.Red;

}

else

changeButton.BackColor = Color.Red;

}

// Метод для определения, какой ListBox был выбран

private ListBox GetSelectedListBox()

{

if (listBox1.SelectedItem != null)

return listBox1;

if (listBox2.SelectedItem != null)

return listBox2;

if (listBox3.SelectedItem != null)

return listBox3;

return null; // Если ни один элемент не выбран

}

// Метод для выбора соответствующей хэш-таблицы

private Dictionary<string, List<string>> GetSelectedHashTable(ListBox listBox)

{

if (listBox == listBox1)

return hashTableIdentifier;

if (listBox == listBox2)

return hashTableDigital;

if (listBox == listBox3)

return hashTableSpecial;

return null; // Если ListBox не совпадает ни с одним из ожидаемых

}

}

}

**uLex.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Translator

{

// Перечисление состояний анализа

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

// Перечисление возможных типов символов

public enum TCharType { EngLetter, RusLetter, Digit, SETQ, CL, EndRow, EndText, Space, Star, Slash, Exclamation, Equal, Semicolon, AnotherSymbol, OpenBracket, EndBracket, Colon, OpenSquadBracket, EndSquadBracket, Plus, Minus, Comma, Dot, NoInd }; // Тип символа

// Перечисление типов токенов

public enum TToken { lxmIdentifier, lxmSETQ, lxmCL, lxmSpace, lxmOpenBracket, lxmEndBracket, lxmNumber, lxmUnknown, lxmEmpty, lxmLeftParenth, lxmRightParenth, lxmIs, lxmDot, lxmComma, lxmText, lxmtz, lxmdt, lxmr, lxmrs, lxmls };

public class CLex

{

// Поля класса

private String[] strFSource; // Массив строк, представляющий исходный текст

private String[] strFMessage; // Массив строк для сообщений (возможно, для вывода результатов)

public TCharType enumFSelectionCharType; // Тип текущего символа

public char chrFSelection; // Текущий символ

private TState enumFState; // Текущее состояние анализатора

private int intFSourceRowSelection; // Номер текущей строки

private int intFSourceColSelection; // Номер текущей колонки в строке

private String strFLexicalUnit; // Текущая лексическая единица

private TToken enumFToken; // Текущий токен

// Свойства для доступа к полям класса

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; } }

// Конструктор класса

public CLex()

{

}

// Метод получения текущего символа из источника

public void GetSymbol()

{

// Проверяем, не вышли ли за пределы строки

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

{

intFSourceRowSelection++; // Переходим на следующую строку

if (intFSourceRowSelection <= strFSource.Length - 1)

{

// Если еще не конец текста, сбрасываем колонку и задаем символ конца строки

intFSourceColSelection = -1;

chrFSelection = '\0'; // Устанавливаем текущий символ в '\0'

enumFSelectionCharType = TCharType.EndRow; // Указываем, что достигнут конец строки

enumFState = TState.Continue; // Состояние анализа продолжается

}

else

{

// Если достигли конца всего текста

chrFSelection = '\0'; // Устанавливаем текущий символ в '\0'

enumFSelectionCharType = TCharType.EndText; // Указываем, что достигнут конец текста

enumFState = TState.Finish; // Меняем состояние на "Finish"

}

}

else

{

// Получаем текущий символ в строке

chrFSelection = strFSource[intFSourceRowSelection][intFSourceColSelection];

// Классифицируем символ

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

else if (chrFSelection >= 'a' && chrFSelection <= 'z') enumFSelectionCharType = TCharType.EngLetter;

else if (chrFSelection >= 'а' && chrFSelection <= 'я') enumFSelectionCharType = TCharType.RusLetter;

else if (chrFSelection >= '0' && chrFSelection <= '9') enumFSelectionCharType = TCharType.Digit;

else if (chrFSelection == 'S' || chrFSelection == 'E' || chrFSelection == 'T' || chrFSelection == 'Q') enumFSelectionCharType = TCharType.SETQ;

else if (chrFSelection == 'C' || chrFSelection == 'E' || chrFSelection == 'O' || chrFSelection == 'M' || chrFSelection == 'A' || chrFSelection == 'N' || chrFSelection == 'D' || chrFSelection == 'L' || chrFSelection == 'I' || chrFSelection == '"') enumFSelectionCharType = TCharType.CL;

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

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

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

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

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

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

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

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

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

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

else if (chrFSelection == '+') enumFSelectionCharType = TCharType.Plus;

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

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

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

else if (chrFSelection == '^' || chrFSelection == '%' || chrFSelection == '@' || chrFSelection == '<' || chrFSelection == '>' || chrFSelection == '?')

enumFSelectionCharType = TCharType.AnotherSymbol; // Считаем эти символы как другие (AnotherSymbol)

else enumFSelectionCharType = TCharType.NoInd; // Символ не распознан

enumFState = TState.Continue; // Продолжаем анализ

}

intFSourceColSelection++; // Переходим к следующему символу

}

// Метод добавления символа к текущей лексической единице

private void TakeSymbol()

{

char[] c = { chrFSelection }; // Создаем массив символов с текущим символом

String s = new string(c); // Преобразуем массив символов в строку

strFLexicalUnit += s; // Добавляем символ к текущей лексической единице

GetSymbol(); // Получаем следующий символ

}

// Метод перехода к следующему токену

public void NextToken()

{

strFLexicalUnit = ""; // Сбрасываем текущую лексическую единицу

char[] allowedChars = { 'a', 'b', 'c', 'd' };

// Начальная инициализация перед началом анализа

if (enumFState == TState.Start)

GetSymbol(); // Получаем первый символ

// Пропуск комментариев (если встречается '//')

if (chrFSelection == '/')

{

GetSymbol(); // Получаем следующий символ

if (chrFSelection == '/')

{

// Игнорируем все символы до конца строки

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

GetSymbol(); // Переходим к следующему символу после конца строки

}

}

if (chrFSelection == '\0' && enumFSelectionCharType == TCharType.EndText)

enumFState = TState.Finish;

// Variant 13

switch (enumFSelectionCharType)

{

case TCharType.EngLetter:

{

// a b c d

// A |BFin|BFin|BFin|BFin|

// BFin |CFin|CFin|cFin|CFin|

// CFin | D | D | D | D |

// D |Fin | Fin| Fin| Fin|

// Fin | | | | |

A:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto BFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

BFin:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto CFin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

CFin:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto D;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

D:

{

if (allowedChars.Contains(chrFSelection))

{

TakeSymbol();

goto Fin;

}

else

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

Fin:

{

enumFToken = TToken.lxmIdentifier;

return;

}

}

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 == '0')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался 0");

C:

if (chrFSelection == '0')

{

TakeSymbol();

goto E;

}

else throw new Exception("Ожидался 0");

D:

if (chrFSelection == '1')

{

TakeSymbol();

goto A;

}

else throw new Exception("Ожидалась 1");

E:

if (chrFSelection == '1')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидалась 1");

FFin:

if (chrFSelection == '1')

{

TakeSymbol();

goto G;

}

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

else throw new Exception("Ожидалась 1");

G:

if (chrFSelection == '1')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался 0");

H:

if (chrFSelection == '0')

{

TakeSymbol();

goto FFin;

}

else throw new Exception("Ожидался 0");

}

// Создание токена служебного слова SETQ

case TCharType.SETQ:

{

A:

if (chrFSelection == 'S')

{

TakeSymbol();

goto B;

}

else throw new Exception("Ожидался S");

B:

if (chrFSelection == 'E')

{

TakeSymbol();

goto C;

}

else throw new Exception("Ожидался E");

C:

if (chrFSelection == 'T')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался T");

D:

if (chrFSelection == 'Q')

{

TakeSymbol();

goto Fin;

}

else throw new Exception("Ожидался Q");

Fin:

{

enumFToken = TToken.lxmSETQ;

return;

}

}

// Создание токена служебного слова COMMAND "LINE"

case TCharType.CL:

{

A:

if (chrFSelection == 'C')

{

TakeSymbol();

goto B;

}

else throw new Exception("Ожидался C");

B:

if (chrFSelection == 'O')

{

TakeSymbol();

goto C;

}

else throw new Exception("Ожидался O");

C:

if (chrFSelection == 'M')

{

TakeSymbol();

goto D;

}

else throw new Exception("Ожидался M");

D:

if (chrFSelection == 'M')

{

TakeSymbol();

goto E;

}

else throw new Exception("Ожидался M");

E:

if (chrFSelection == 'A')

{

TakeSymbol();

goto F;

}

else throw new Exception("Ожидался A");

F:

if (chrFSelection == 'N')

{

TakeSymbol();

goto G;

}

else throw new Exception("Ожидался N");

G:

if (chrFSelection == 'D')

{

TakeSymbol();

goto H;

}

else throw new Exception("Ожидался D");

H:

if (chrFSelection == '"')

{

TakeSymbol();

goto I;

}

else throw new Exception("Ожидался \"");

I:

if (chrFSelection == 'L')

{

TakeSymbol();

goto J;

}

else throw new Exception("Ожидался L");

J:

if (chrFSelection == 'I')

{

TakeSymbol();

goto K;

}

else throw new Exception("Ожидался I");

K:

if (chrFSelection == 'N')

{

TakeSymbol();

goto L;

}

else throw new Exception("Ожидался N");

L:

if (chrFSelection == 'E')

{

TakeSymbol();

goto M;

}

else throw new Exception("Ожидался E");

M:

if (chrFSelection == '"')

{

TakeSymbol();

goto Fin;

}

else throw new Exception("Ожидался \"");

Fin:

{

enumFToken = TToken.lxmCL;

return;

}

}

case TCharType.AnotherSymbol:

{

if (chrFSelection == '/')

{

GetSymbol();

if (chrFSelection == '/')

{

while (enumFSelectionCharType != TCharType.EndRow)

GetSymbol();

}

GetSymbol();

}

break;

}

// Добавление токенов

case TCharType.Space:

if (chrFSelection == ' ')

{

enumFToken = TToken.lxmSpace;

GetSymbol();

return;

}

break;

case TCharType.OpenBracket:

if (chrFSelection == '(')

{

enumFToken = TToken.lxmOpenBracket;

GetSymbol();

return;

}

break;

case TCharType.EndBracket:

if (chrFSelection == ')')

{

enumFToken = TToken.lxmEndBracket;

GetSymbol();

return;

}

break;

case TCharType.EndRow:

{

GetSymbol();

enumFToken = TToken.lxmEmpty;

break;

}

case TCharType.EndText:

{

enumFToken = TToken.lxmEmpty;

break;

}

}

}

}

}

**uSyntAnalyzer.cs**

using System;

using System.Collections.Generic;

using System.Text;

using Translator;

namespace nsSynt

{

class uSyntAnalyzer

{

private String[] strFSource;

private String[] strFMessage;

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

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

public CLex Lex = new CLex();

public void S()

{

O(); // разбор правила O

if (Lex.enumPToken == TToken.lxmSpace)

A(); // если есть пробелы, вызываем A

}

public void A()

{

if (Lex.enumPToken == TToken.lxmSpace) // разбор пробела

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmSETQ || Lex.enumPToken == TToken.lxmCL)

O(); // после пробела ожидается O

else

A(); // или продолжаем разбор пробела

}

}

public void O()

{

if (Lex.enumPToken == TToken.lxmOpenBracket) // проверяем открывающую скобку

{

Lex.NextToken(); // переходим к следующему токену после скобки

if (Lex.enumPToken == TToken.lxmSETQ) // разбор SETQ

{

Lex.NextToken(); // переходим после SETQ

if (Lex.enumPToken == TToken.lxmSpace) // проверка на пробел

{

Lex.NextToken();

V(); // после пробела ожидается V (переменная или список)

if (Lex.enumPToken == TToken.lxmEndBracket) { } // проверка на закрывающую скобку

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

}

else throw new Exception("Ожидался пробел после SETQ");

}

else if (Lex.enumPToken == TToken.lxmCL) // разбор COMMAND"LINE"

{

Lex.NextToken(); // переходим к следующему токену после COMMAND

if (Lex.enumPToken == TToken.lxmSpace) // проверка на пробел

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmIdentifier) // ожидаем первый индефикатор

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmSpace) // проверка на пробел

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmIdentifier) // ожидаем второй индефикатор

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmEndBracket) // проверка на закрывающую скобку

Lex.NextToken(); // завершаем разбор COMMAND "LINE"

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

}

else throw new Exception("Ожидался второй индефикатор для LINE");

}

else throw new Exception("Ожидался пробел после первого индефикатора");

}

else throw new Exception("Ожидался первый индефикатор для LINE");

}

else throw new Exception("Ожидался пробел после COMMAND\"LINE\"");

}

else throw new Exception("Ожидался SETQ или COMMAND");

}

else throw new Exception("Ожидалась открывающая скобка");

}

public void V()

{

C(); // разбор C как часть V

if (Lex.enumPToken == TToken.lxmSpace) // проверяем на пробел для B

B(); // если есть пробел, вызываем B

}

public void B()

{

if (Lex.enumPToken == TToken.lxmSpace) // разбор пробела

{

Lex.NextToken();

C(); // вызываем C после пробела

if (Lex.enumPToken == TToken.lxmSpace) // продолжаем разбор пробела

B(); // продолжаем разбор B

}

}

public void C()

{

if (Lex.enumPToken == TToken.lxmIdentifier) // ожидаем индефикатор

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmSpace)

{

Lex.NextToken();

if (Lex.enumPToken == TToken.lxmNumber) // ожидаем число

Lex.NextToken();

else throw new Exception("Ожидался числовое значение");

}

else throw new Exception("Ожидался пробел");

}

else throw new Exception("Ожидалось индефикатор");

}

}

}

**Результаты тестирования:**

