Государственное образовательное учреждение

высшего профессионального образования

Кубанский государственный технологический университет

(КубГТУ)

Армавирский механико-технологический институт

Кафедра внутризаводского электрооборудования и автоматики

#### ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине: Теория языков программирования

на тему: «Лексический и синтаксический анализатор языка высокого уровня»

Государственное образовательное учреждение

высшего профессионального образования

Кубанский государственный технологический университет

(КубГТУ)

Армавирский механико-технологический институт

Кафедра внутризаводского электрооборудования и автоматики

Утверждаю

Заведующий кафедрой ВЭА

Проф. \_\_\_\_\_\_\_\_\_\_\_\_ В.И. Куроедов

#### ЗАДАНИЕ

на курсовой проект

Студенту группы\_\_\_\_\_\_\_\_\_\_\_\_курса\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

специальности 230105(2204) «Программное обеспечение вычислительной техники и автоматизированных систем»

Тема проекта: «Лексический и синтаксический анализатор языка высокого уровня»

Содержание задания:Спроектировать и построить лексический и синтаксический анализаторы для заданного формального языка программирования. Построить и реализовать в лексическом анализаторе распознаватель лексем с заданной структурой: нагруженное дерево (один элемент дерева хранит один символ (букву входного алфавита) лексемы).

Учебный язык включает директиву using, функцию main(), описание переменных, констант, цикла for, операторов присваивания, арифметические операции. За основу лексики, синтаксиса и семантики учебного языка принять стандарты языка программирования С#.

Объем работы:

Пояснительная записка 35 – 40 листов

Графическая часть 2-3 листа формата А3

Рекомендуемая литература**:**

1. Ключко В.И. Теория вычислительных процессов и структур. Учебное пособие, - КубГТУ, 1998;

2. Соколов А.П. Системы программирования: теория, методы, алгоритмы: Учеб. Пособие, - М.:

Финансы и статистика, 2004. – 320 с.: ил.

1. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. - СПб.: Питер, 2001. - 736 с.

Срок выполнения проекта:

с «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_ «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г.

Срок защиты «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_ г.

Дата выдачи задания «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г.

Дата сдача проекта на кафедру «\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_20\_\_\_ г.

Руководитель канд. техн. наук, доцент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Задание принял студент\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Нормативные ссылки**

ГОСТ Р ИСО 9001-2001 Системы менеджмента качества. Требования.

ГОСТ 7.1-2003 СИБИД. Библиографическая запись. Библиографическое описание. Общие требования и правила составления.

ГОСТ 19.101-77 ЕСПД. Виды программ и программных документов.

ГОСТ 19.102-77 ЕСПД. Стадии разработки.

ГОСТ 19.103-77 ЕСПД. Обозначение программ и программных документов.

ГОСТ 19.105-78 ЕСПД. Общие требования к программным документам

ГОСТ 19.202-78 ЕСПД. Спецификация. Требования к содержанию и оформлению.

ГОСТ 19.404-79 ЕСПД. Пояснительная записка Требования к содержанию и оформлению.

ГОСТ 19.701-90 ЕСПД. Схемы алгоритмов, программ, данных и систем. Обозначения условные и правила выполнения.

**Реферат**

Пояснительная записка к курсовому проекту содержит 37 листов, 8 рисунков, 3 таблицы, 9 литературных источников, 2 приложения. К пояснительной записке прилагается 1 диск с готовой программой и материалами к ней, а также графическая часть, состоящая из трех листов.

СИНТАКСИС, ЛЕКСЕМА, КОНСТАНТА, АВТОМАТ – РАСПОЗНАВАТЕЛЬ, РЕГУЛЯРНОЕ МНОЖЕСТВО, ФОРМАЛЬНАЯ ГРАМАТИКА, ТЕРМИНАЛ, НЕТЕРМИНАЛ, АВТОМАТ С МАГАЗИННОЙ ПАМЯТЬЮ

Объект: лексический и синтаксический анализатор.

Цель: практическое применение теории формальных грамматик и теории автоматов для проектирования трансляторов.

Рассмотрен синтаксис заданного языка программирования, разработана грамматика регулярных множеств. Спроектированы автоматы для лексического анализа и распознавания лексем. Разработана формальная LL(1) грамматика для заданного языка программирования, спроектирован автомат с магазинной памятью для нисходящего анализа программы. Написана программа на языке высокого уровня Microsoft Visual C++ для лексического и синтаксического анализа текста на учебном языке программирования.

**Содержание**

[Введение](#Введение)

[1 Синтез лексического анализатора (сканера)](#_Toc230072770)

[1.1 Описание синтаксиса формального языка программирования](#_Toc230072771)

[1.2 Система регулярных выражений](#_Toc230072772)

[1.3 Распознаватели констант и служебных слов](#_Toc230072773)

[1.4 Управляющая таблица конечного автомата лексического анализа](#_Toc230072773)

[2 Синтез синтаксического анализатора](#_Toc230072774)

[2.1 Описание формальной грамматики](#_Toc230072775)

[2.2 Построение множества ВЫБОР(n)](#_Toc230072776)

[2.3 Построение управляющей таблицы](#_Toc230072777)

[3 Описание программы](#_Toc230072778)

[4 Результаты тестирования](#_Toc230072779)

[5 Руководство пользователя](#_Toc230072780)

[Заключение](#_Toc230072781)

[Список используемой литературы](#_Toc230072782)

[Приложение А. Листинг лексического анализатора](#_Toc230072783)

[Приложение Б. Листинг синтаксического анализатора](#_Toc230072784)

**Введение**

Целью данного проекта является: практическое применение теории формальных грамматик и теории автоматов для проектирования трансляторов.

Задачи: написать транслирующую грамматику для учебного языка программирования; спроектировать и построить лексический анализатор; спроектировать и построить синтаксический анализатор.

Входной информацией является файл, содержащий текст программы, написанной на учебном языке.

Выходной информацией являются таблицы лексем и имен, а также подтверждение того, что код соответствует синтаксису.

Результатом курсового проекта должна быть программа-анализатор, состоящая из двух частей: лексического анализатора, разбивающего исходный текст программы на лексемы и заполняющего таблицу имен; синтаксического анализатора, проверяющего соответствие текста заданной грамматике.

**1 Синтез лексического анализатора (сканера)**

**1.1 Описание синтаксиса формального языка программирования**

Директива using позволяет в текущем пространстве имен использовать типы данных, определенные в другом пространстве имен. Синтаксис:

using System.Text;

В данном случае лексема using является ключевым словом.

С помощью ключевого слова class определяются классы. Например:

public class TestClass

{

// Определение полей и методов класса

}

Ключевое слово public определяет уровень доступности класса.

Поля класса определяются как переменные:

public class TestClass

{

public uint a, b = 35, i;

public bool c, d;

public const long int e = 9L;

}

Для каждого поля прописывается модификатор доступа (public) и тип поля (double, int или decimal).

Методы класса определяются так же, как и обычные функции:

public class TestClass

{

public int Main(Param1, Param2)

{ }

}

Как и для полей класса, для методов задается модификатор доступа и тип возвращаемого значения метода.

Тело метода класса согласно учебному языку может содержать:

–определение цикла со счетчиком:

for (<выражение>; <условие>; <выражение>)

{ <блок операторов> }

–вызовы процедур:

write (<список параметров>);

read (<список параметров>);

–операторы присваивания:

a = <выражение>;

d \*= <выражение>;

f /= <выражение>;

–арифметические выражения, содержащие бинарные операции:

a = b - (c + d);

Так же в тексте программы могут содержаться многострочные комментарии:

/\* многострочный

комментарий \*/

Для записи идентификаторов используются буквы английского языка и цифры. Идентификаторы начинаются с буквы. Целые константы записываются арабскими цифрами.

**1.2 Система регулярных выражений**

Для записи грамматики лексем языка применим форму Бэкуса-Наура.

Для записи идентификаторов используются буквы английского языка и цифры. Идентификаторы начинаются с буквы. Синтаксис идентификаторов определяется праволинейной регулярной грамматикой:

<S> -> L <I>( 1)

<I> -> L <I>( 2)

<I> -> D <I>( 3)

<I> -> е( 4)

где L - буква множества (A..Z), D - цифра множества (0..9), е - пустая цепочка или символ окончания лексемы.

Целые константы записываются арабскими цифрами.

Праволинейная грамматика целого числа:

<ЦЧ> → +<Ц>|-<Ц>

<Ц> → н<Ц>

<Ц> → н|е

**1.3 Распознаватели констант и служебных слов**

Заданная грамматика может бать реализована автоматом со следующими состояниями:

S - состояние ожидания первого символа лексемы;

I - состояние ожидания символов идентификаторов: буквы, цифры;

С - состояние ожидания символов целой части числа;

E -состояние ошибки;

R - состояние допуска лексемы.

Автомат переходит в допустимое состояние R из состояния I для идентификаторов и из состояния C для чисел.

Регулярная грамматика для заданных условий записи лексем задается следующими множествами:

Р: [P1, P2, … ,P4] – множество правил;

G: [S, I, C, E, R] – множество состояний, где S – начальный символ;

[0..9, A..Z, «–», «#», «(», «)», «\*», «,», «.», «/», «:», «;», «{«, «}», «+», «=»] – множество входных символов, из них разделительные символы и уникальные лексемы [«–», «#», «(», «)», «\*», «,», «/», «:», «;», «{«, «}», «+», «=»].

Символы пробел и табуляции означают конец лексемы. Эти символы не является лексемой и требуют выполнения операции «СДВИГ» над входной строкой. По символу пробел автомат допускает лексему и переходит в начальное состояние анализа следующего символа входной строки автомата.

Символы: «–», «#», «(», «)», «\*», «,», «/», «:», «;», «{«, «}», «+», «=» являются одновременно разделительными знаками и началом следующей лексемы, даже если перед ними нет символа конца лексемы. Операция «СДВИГ» после этих символов не требуется, автомат допускает лексему и переходит в начальное состояние для анализа этих же символов.

Таблица 1 – Таблица переходов автомата распознавателя идентификаторов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | L | D | e |  |
| S | I | E | S | Нач. состояние |
| I | I | I | R |  |
| E | Ошибка | | | |
| R | Допустить | | | |

Идентификация служебных слов реализована автоматом распознавателя: нагруженное дерево (Рисунок 1). Множество служебных слов: BREAK CLASS CONST CONTINUE LONG INT BOOL FOR UINT PUBLIC READ USING WRITE.

*КОРЕНЬ*

*B*

*C*

*R*

*L*

*O*

*N*

*E*

*A*

*A*

*K*

*S*

*S*

*S*

*T*

*T*

*I*

*N*

*U*

*E*

Рисунок 1 – Нагруженное дерево (часть)

Структура данных для формирования нагруженного дерева:

protected struct KeywordTree

{

public bool bIsWordEnd;

public char cLetter;

public List<KeywordTree> pNextList;

}

KeywordTree pKeywordsList;

Процедура заполнения нагруженного дерева:

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

{String[] KeyWords = {"BREAK", "CLASS", "CONST", "CONTINUE", "LONG", "BOOL", "FOR",

"INT", "UINT", "PUBLIC", "READ", "USING", "WRITE"};

KeywordTree p, q;

pKeywordsList = new KeywordTree();

pKeywordsList.cLetter = '#';

pKeywordsList.pNextList = new List<KeywordTree>();

pKeywordsList.bIsWordEnd = true;

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

{String KeyWord = KeyWords[i];

p = pKeywordsList;

for (int j = 0; j < KeyWord.Length; j++)

{if (p.pNextList.Count == 0)

{q = new KeywordTree();

q.cLetter = KeyWord[j];

q.pNextList = new List<KeywordTree>();

//если последний символ в ключ. слове

q.bIsWordEnd = (j + 1 == KeyWord.Length) ? (q.bIsWordEnd =

true) : (q.bIsWordEnd = false);

p.pNextList.Add(q);

p = q;

}

else

{bool bIsAlready = false;

for (int k = 0; k < p.pNextList.Count; k++)

if (p.pNextList[k].cLetter == KeyWord[j])

{bIsAlready = true;

p = p.pNextList[k];

break;

}

if (bIsAlready == false)

{q = new KeywordTree();

q.cLetter = KeyWord[j];

q.pNextList = new List<KeywordTree>();

q.bIsWordEnd = false;

p.pNextList.Add(q);

p = q;

}}}}}

Функция идентификации служебных слов:

private bool IsKeyword(String sLexeme)

{int j, k;

KeywordTree p = pKeywordsList;

for (j = 0; j < sLexeme.Length; j++)

{if (p.pNextList.Count == 0) break;

else

{bool bIsFound = false;

for (k = 0; k < p.pNextList.Count; k++)

if (p.pNextList[k].cLetter == sLexeme[j])

{bIsFound = true;

p = p.pNextList[k];

break;

}

if (!bIsFound) break;

}}

return ((j == sLexeme.Length) && (p.bIsWordEnd));

}

**1.4 Управляющая таблица конечного автомата лексического анализа**

Управляющая таблица лексического анализатора для заданной выше грамматики показана в таблице 2. Листинг программы, реализующей данную управляющую таблицу, приведен в приложении А.

При составлении таблицы использовались следующие обозначения:

– первым символом указывается состояние, в которое переходит автомат при поступлении соответствующего символа;

– символ «+» означает добавление входного символа к лексеме;

– символ «>>» означает сдвиг входной строки на один символ.

Таблица 2 – Управляющая таблица конечного автомата лексического анализатора

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Spaces | Letters | Digits | Symbols |
| S | S, >> | I, +, >> | C, +, >> | R, +, >> |
| C | R | E | C, +, >> | R |
| I | R | I, +, >> | I, +, >> | R |
| E | Ошибка | | | |
| R | S, Допустить | | | |

Spaces – множество символов пробела (сам пробел и знак табуляции);

Letters – множество букв латинского алфавита [A..Z, «\_»];

Digits – множество арабских цифр [0..9];

Symbols – множество разделительных символов [«–», «#», «(», «)», «\*», «,», «/», «:», «;», «{«, «}», «+», «=»].

**2 Синтез синтаксического анализатора**

**2.1 Описание формальной грамматики**

Грамматика целевого символа:

( 1) <S> -> using <USING\_LIST> <NEXT>

( 2) <S> -> public <CLASS> <NEXT>

( 3) <NEXT> -> ; <S>

( 4) <NEXT> -> e

Грамматика описания using:

( 5) <USING\_LIST> -> ID <NEXT\_USING>

( 6) <NEXT\_USING> -> . <USING\_LIST>

( 7) <NEXT\_USING> -> e

Грамматика описания класса:

( 8) <CLASS> -> class ID { <CLASS\_BODY> }

( 9) <CLASS\_BODY> -> public <DEF> <NEXT\_BODY>

(10) <NEXT\_BODY> -> ; <CLASS\_BODY>

(11) <NEXT\_BODY> -> e

Грамматика описания определения полей и методов класса:

(12) <DEF> -> uint <DEF\_LIST>

(13) <DEF> -> bool <DEF\_LIST>

(14) <DEF> -> const long int <DEF\_LIST>

(15) <DEF\_LIST> -> ID <NEXT\_DEF>

(16) <NEXT\_DEF> -> ( <VAR\_LIST> ) { <OPER\_LIST> }

(17) <NEXT\_DEF> -> , <VAR\_LIST>

(18) <NEXT\_DEF> -> = <EXPR> <VAR\_LIST>

(19) <NEXT\_DEF> -> e

(20) <VAR\_LIST> -> ID <NEXT\_VAR>

(21) <NEXT\_VAR> -> , <VAR\_LIST>

(22) <NEXT\_VAR> -> = <EXPR> <VAR\_LIST>

(23) <NEXT\_VAR> -> e

Грамматика описания списка операторов:

(24) <OPER\_LIST> -> <OPERATOR> <NEXT\_OPER>

(25) <NEXT\_OPER> -> ; <OPER\_LIST>

(26) <NEXT\_OPER> -> e

Грамматика описания операторов:

(27) <OPERATOR> -> for ( ID = <EXPR> ; <COND> ; ID <LET> ) { <OPER\_LIST> }

(28) <OPERATOR> -> break

(29) <OPERATOR> -> continue

(30) <OPERATOR> -> write ( <VAR\_LIST> )

(31) <OPERATOR> -> read ( <VAR\_LIST> )

(32) <OPERATOR> -> ID <LET>

(33) <LET> -> = <EXPR>

(34) <LET> -> \* = <EXPR>

(35) <LET> -> / = <EXPR>

Грамматика описания арифметического выражения:

(36) <EXPR> -> ( <EXPR> ) <OPERATION>

(37) <EXPR> -> ID <OPERATION>

(38) <EXPR> -> NUM <OPERATION>

(39) <OPERATION> -> + <EXPR>

(40) <OPERATION> -> - <EXPR>

(41) <OPERATION> -> e

Грамматика описания условия:

(42) <COND> -> ( <COND> ) <RELATION>

(43) <COND> -> <EXPR> <RELATION>

(44) <RELATION> -> > <COND>

(45) <RELATION> -> < <COND>

(46) <RELATION> -> = = <COND>

(47) <RELATION> -> e

**2.2 Построение множества ВЫБОР(n)**

Построение множества ПЕРВ(n). Шаг 0. Для построения множества ПЕРВ(n) = FIRST(1, A), где А - нетерминал в правой части n – го правила, определим множества первых символов, стоящих в начале правых частей правил, для каждого нетерминала в левой части правил.

|  |  |
| --- | --- |
| ПЕРВ(<S>) | using public |
| ПЕРВ(<NEXT>) | ; |
| ПЕРВ(<USING\_LIST>) | ID |
| ПЕРВ(<NEXT\_USING>) | . |
| ПЕРВ(<CLASS>) | class |
| ПЕРВ(<CLASS\_BODY>) | public |
| ПЕРВ(<NEXT\_BODY>) | ; |
| ПЕРВ(<DEF>) | uint bool const |
| ПЕРВ(<DEF\_LIST>) | ID |
| ПЕРВ(<NEXT\_DEF>) | ( , = |
| ПЕРВ(<VAR\_LIST>) | ID |
| ПЕРВ(<NEXT\_VAR>) | , = |
| ПЕРВ(<OPER\_LIST>) |  |
| ПЕРВ(<NEXT\_OPER>) | ; |
| ПЕРВ(<OPERATOR>) | for break continue write read ID |
| ПЕРВ(<LET>) | = \* / |
| ПЕРВ(<EXPR>) | ( ID NUM |
| ПЕРВ(<OPERATION>) | + - |
| ПЕРВ(<COND>) | ( |
| ПЕРВ(<RELATION>) | > < = |

Шаг 1. Внесем во множество первых символов ПЕРВ(n)i для каждого правила символы, стоящие в начале правых частей правил. Индекс i = 0 – номер итерации.

|  |  |
| --- | --- |
| 1. 0 | using |
| 1. 0 | public |
| 1. 0 | ; |
| 1. 0 |  |
| 1. 0 | ID |
| 1. 0 | . |
| 1. 0 |  |
| 1. 0 | class |
| 1. 0 | public |
| 1. 0 | ; |
| 1. 0 |  |
| 1. 0 | uint |
| 1. 0 | bool |
|  | const |
| 1. 0 | ID |
| 1. 0 | ( |
| 1. 0 | , |
|  | = |
| 1. 0 |  |
| 1. 0 | ID |
| 1. 0 | , |
|  | = |
| 1. 0 |  |
| 1. 0 | <OPERATOR> |
| 1. 0 | ; |
| 1. 0 |  |
| 1. 0 | for |
| 1. 0 | break |
| 1. 0 | continue |
| 1. 0 | write |
| 1. 0 | read |
| 1. 0 | ID |
| 1. 0 | = |
| 1. 0 | \* |
| 1. 0 | / |
| 1. 0 | ( |
| 1. 0 | ID |
|  | NUM |
| 1. 0 | + |
| 1. 0 | - |
| 1. 0 |  |
| 1. 0 | ( |
| 1. 0 | <EXPR> |
| 1. 0 | > |
| 1. 0 | < |
| 1. 0 | = |
| 1. 0 |  |

Шаг 2. Если множество первых символов содержит нетерминал В, то включить в него символы множества ПЕРВ(В), определенное на шаге 0.

|  |  |
| --- | --- |
| 1. 1 | using |
| 1. 1 | public |
| 1. 1 | ; |
| 1. 1 |  |
| 1. 1 | ID |
| 1. 1 | . |
| 1. 1 |  |
| 1. 1 | class |
| 1. 1 | public |
| 1. 1 | ; |
| 1. 1 |  |
| 1. 1 | uint |
| 1. 1 | bool |
| 1. 1 | const |
| 1. 1 | ID |
| 1. 1 | ( |
| 1. 1 | , |
| 1. 1 | = |
| 1. 1 |  |
| 1. 1 | ID |
| 1. 1 | , |
| 1. 1 | = |
| 1. 1 |  |
| 1. 1 | for break continue write read ID <OPERATOR> |
| 1. 1 | ; |
| 1. 1 |  |
| 1. 1 | for |
| 1. 1 | break |
| 1. 1 | continue |
| 1. 1 | write |
| 1. 1 | read |
| 1. 1 | ID |
| 1. 1 | = |
| 1. 1 | \* |
| 1. 1 | / |
| 1. 1 | ( |
| 1. 1 | ID |
| 1. 1 | NUM |
| 1. 1 | + |
| 1. 1 | - |
| 1. 1 |  |
| 1. 1 | ( |
| 1. 1 | ( ID NUM <EXPR> |
| 1. 1 | > |
| 1. 1 | < |
| 1. 1 | = |
| 1. 1 |  |

Шаг 3. Если ПЕРВ(n)i+1 ≠ ПЕРВ(n)i, то i = i + 1 и перейти к шагу 2, иначе закончить.

Шаг 2. Если множество первых символов содержит нетерминал В, то включить в него символы множества ПЕРВ(В), определенное на шаге 0.

|  |  |
| --- | --- |
| 1. 2 | using |
| 1. 2 | public |
| 1. 2 | ; |
| 1. 2 |  |
| 1. 2 | ID |
| 1. 2 | . |
| 1. 2 |  |
| 1. 2 | class |
| 1. 2 | public |
| 1. 2 | ; |
| 1. 2 |  |
| 1. 2 | uint |
| 1. 2 | bool |
| 1. 2 | const |
| 1. 2 | ID |
| 1. 2 | ( |
| 1. 2 | , |
| 1. 2 | = |
| 1. 2 |  |
| 1. 2 | ID |
| 1. 2 | , |
| 1. 2 | = |
| 1. 2 |  |
| 1. 2 | for break continue write read ID |
| 1. 2 | ; |
| 1. 2 |  |
| 1. 2 | for |
| 1. 2 | break |
| 1. 2 | continue |
| 1. 2 | write |
| 1. 2 | read |
| 1. 2 | ID |
| 1. 2 | = |
| 1. 2 | \* |
| 1. 2 | / |
| 1. 2 | ( |
| 1. 2 | ID |
| 1. 2 | NUM |
| 1. 2 | + |
| 1. 2 | - |
| 1. 2 |  |
| 1. 2 | ( |
| 1. 2 | ( ID NUM |
| 1. 2 | > |
| 1. 2 | < |
| 1. 2 | = |
| 1. 2 |  |

Шаг 3. Если ПЕРВ(n)i+1 ≠ ПЕРВ(n)i, то i = i + 1 и перейти к шагу 2, иначе закончить.

Построение множества ПЕРВ(А). Множества ПЕРВ(А) необходимо для определения множеств СЛЕД(А).

Шаг 1. Для построения множества ПЕРВ(А) = FIRST(1, A), где А - нетерминал в правой части правил, определим множества первых символов, стоящих в начале правых частей правил, для каждого нетерминала в левой части правил.

|  |  |
| --- | --- |
| ПЕРВ(<S>) | using public |
| ПЕРВ(<NEXT>) | ; |
| ПЕРВ(<USING\_LIST>) | ID |
| ПЕРВ(<NEXT\_USING>) | . |
| ПЕРВ(<CLASS>) | class |
| ПЕРВ(<CLASS\_BODY>) | public |
| ПЕРВ(<NEXT\_BODY>) | ; |
| ПЕРВ(<DEF>) | uint bool const |
| ПЕРВ(<DEF\_LIST>) | ID |
| ПЕРВ(<NEXT\_DEF>) | ( , = |
| ПЕРВ(<VAR\_LIST>) | ID |
| ПЕРВ(<NEXT\_VAR>) | , = |
| ПЕРВ(<OPER\_LIST>) | <OPERATOR> |
| ПЕРВ(<NEXT\_OPER>) | ; |
| ПЕРВ(<OPERATOR>) | for break continue write read ID |
| ПЕРВ(<LET>) | = \* / |
| ПЕРВ(<EXPR>) | ( ID NUM |
| ПЕРВ(<OPERATION>) | + - |
| ПЕРВ(<COND>) | <EXPR> ( |
| ПЕРВ(<RELATION>) | > < = |

Шаг 2. Если множество первых символов содержит нетерминал В, то включить в него символы множества ПЕРВ(В), i=0.

|  |  |
| --- | --- |
| ПЕРВ(<S>)0 | using public |
| ПЕРВ(<NEXT>)0 | ; |
| ПЕРВ(<USING\_LIST>)0 | ID |
| ПЕРВ(<NEXT\_USING>)0 | . |
| ПЕРВ(<CLASS>)0 | class |
| ПЕРВ(<CLASS\_BODY>)0 | public |
| ПЕРВ(<NEXT\_BODY>)0 | ; |
| ПЕРВ(<DEF>)0 | uint bool const |
| ПЕРВ(<DEF\_LIST>)0 | ID |
| ПЕРВ(<NEXT\_DEF>)0 | ( , = |
| ПЕРВ(<VAR\_LIST>)0 | ID |
| ПЕРВ(<NEXT\_VAR>)0 | , = |
| ПЕРВ(<OPER\_LIST>)0 | for break continue write read ID |
| ПЕРВ(<NEXT\_OPER>)0 | ; |
| ПЕРВ(<OPERATOR>)0 | for break continue write read ID |
| ПЕРВ(<LET>)0 | = \* / |
| ПЕРВ(<EXPR>)0 | ( ID NUM |
| ПЕРВ(<OPERATION>)0 | + - |
| ПЕРВ(<COND>)0 | ( ID NUM |
| ПЕРВ(<RELATION>)0 | > < = |

Шаг 3. Выполнение шага 2 привело к изменению множеств ПЕРВ(А), поэтому повторим шаг 2.

Шаг 2. Если множество первых символов содержит нетерминал В, то включить в него символы множества ПЕРВ(В), i=1.

|  |  |
| --- | --- |
| ПЕРВ(<S>)1 | using public |
| ПЕРВ(<NEXT>)1 | ; |
| ПЕРВ(<USING\_LIST>)1 | ID |
| ПЕРВ(<NEXT\_USING>)1 | . |
| ПЕРВ(<CLASS>)1 | class |
| ПЕРВ(<CLASS\_BODY>)1 | public |
| ПЕРВ(<NEXT\_BODY>)1 | ; |
| ПЕРВ(<DEF>)1 | uint bool const |
| ПЕРВ(<DEF\_LIST>)1 | ID |
| ПЕРВ(<NEXT\_DEF>)1 | ( , = |
| ПЕРВ(<VAR\_LIST>)1 | ID |
| ПЕРВ(<NEXT\_VAR>)1 | , = |
| ПЕРВ(<OPER\_LIST>)1 | for break continue write read ID |
| ПЕРВ(<NEXT\_OPER>)1 | ; |
| ПЕРВ(<OPERATOR>)1 | for break continue write read ID |
| ПЕРВ(<LET>)1 | = \* / |
| ПЕРВ(<EXPR>)1 | ( ID NUM |
| ПЕРВ(<OPERATION>)1 | + - |
| ПЕРВ(<COND>)1 | ( ID NUM |
| ПЕРВ(<RELATION>)1 | > < = |

Шаг 3. Дальнейшее выполнения шага 2 не приведет к изменению множеств ПЕРВ(А), поэтому закончим.

Построение множества СЛЕД(А). Множества СЛЕД(А) строятся для всех нетрминальных символов грамматики методом последовательного приближения.

Шаг 1. Внесем во множество последующих символов СЛЕД(А) = FOLLOW(1, A) для каждого нетерминала А все символы, которые в правых частях правил встречаются непосредственно за символом А; i=0.

|  |  |
| --- | --- |
| СЛЕД(<S>)0 |  |
| СЛЕД(<NEXT>)0 |  |
| СЛЕД(<USING\_LIST>)0 | <NEXT> |
| СЛЕД(<NEXT\_USING>)0 |  |
| СЛЕД(<CLASS>)0 | <NEXT> |
| СЛЕД(<CLASS\_BODY>)0 | } |
| СЛЕД(<NEXT\_BODY>)0 |  |
| СЛЕД(<DEF>)0 | <NEXT\_BODY> |
| СЛЕД(<DEF\_LIST>)0 |  |
| СЛЕД(<NEXT\_DEF>)0 |  |
| СЛЕД(<VAR\_LIST>)0 | ) |
| СЛЕД(<NEXT\_VAR>)0 |  |
| СЛЕД(<OPER\_LIST>)0 | } |
| СЛЕД(<NEXT\_OPER>)0 |  |
| СЛЕД(<OPERATOR>)0 | <NEXT\_OPER> |
| СЛЕД(<LET>)0 | ) |
| СЛЕД(<EXPR>)0 | <VAR\_LIST> ; ) <RELATION> |
| СЛЕД(<OPERATION>)0 |  |
| СЛЕД(<COND>)0 | ; ) |
| СЛЕД(<RELATION>)0 |  |

Шаг 2. Внесем пустую цепочку (или символ конца строки) во множество последующих символов для целевого символа <S>.

|  |  |
| --- | --- |
| СЛЕД(<S>)0 | # |
| СЛЕД(<NEXT>)0 |  |
| СЛЕД(<USING\_LIST>)0 | <NEXT> |
| СЛЕД(<NEXT\_USING>)0 |  |
| СЛЕД(<CLASS>)0 | <NEXT> |
| СЛЕД(<CLASS\_BODY>)0 | } |
| СЛЕД(<NEXT\_BODY>)0 |  |
| СЛЕД(<DEF>)0 | <NEXT\_BODY> |
| СЛЕД(<DEF\_LIST>)0 |  |
| СЛЕД(<NEXT\_DEF>)0 |  |
| СЛЕД(<VAR\_LIST>)0 | ) |
| СЛЕД(<NEXT\_VAR>)0 |  |
| СЛЕД(<OPER\_LIST>)0 | } |
| СЛЕД(<NEXT\_OPER>)0 |  |
| СЛЕД(<OPERATOR>)0 | <NEXT\_OPER> |
| СЛЕД(<LET>)0 | ) |
| СЛЕД(<EXPR>)0 | <VAR\_LIST> ; ) <RELATION> |
| СЛЕД(<OPERATION>)0 |  |
| СЛЕД(<COND>)0 | ; ) |
| СЛЕД(<RELATION>)0 |  |

Шаг 3. Для всех нетерминальных символов A включить во множества СЛЕД(A) множества ПЕРВ(В), где B ∈ СЛЕД(A).

|  |  |
| --- | --- |
| СЛЕД(<S>)1 | # |
| СЛЕД(<NEXT>)1 |  |
| СЛЕД(<USING\_LIST>)1 | ; |
| СЛЕД(<NEXT\_USING>)1 |  |
| СЛЕД(<CLASS>)1 | ; |
| СЛЕД(<CLASS\_BODY>)1 | } |
| СЛЕД(<NEXT\_BODY>)1 |  |
| СЛЕД(<DEF>)1 | ; |
| СЛЕД(<DEF\_LIST>)1 |  |
| СЛЕД(<NEXT\_DEF>)1 |  |
| СЛЕД(<VAR\_LIST>)1 | ) |
| СЛЕД(<NEXT\_VAR>)1 |  |
| СЛЕД(<OPER\_LIST>)1 | } |
| СЛЕД(<NEXT\_OPER>)1 |  |
| СЛЕД(<OPERATOR>)1 | ; |
| СЛЕД(<LET>)1 | ) |
| СЛЕД(<EXPR>)1 | ID ; ) < > = |
| СЛЕД(<OPERATION>)1 |  |
| СЛЕД(<COND>)1 | ; ) |
| СЛЕД(<RELATION>)1 |  |

Шаг 4. Для всех нетерминальных символов A включить во множества СЛЕД(A) множества СЛЕД (В), где B ∈ СЛЕД(A), если существует правило B=e.

|  |  |
| --- | --- |
| СЛЕД(<S>)2 | # |
| СЛЕД(<NEXT>)2 |  |
| СЛЕД(<USING\_LIST>)2 | ; |
| СЛЕД(<NEXT\_USING>)2 |  |
| СЛЕД(<CLASS>)2 | ; |
| СЛЕД(<CLASS\_BODY>)2 | } |
| СЛЕД(<NEXT\_BODY>)2 |  |
| СЛЕД(<DEF>)2 | ; |
| СЛЕД(<DEF\_LIST>)2 |  |
| СЛЕД(<NEXT\_DEF>)2 |  |
| СЛЕД(<VAR\_LIST>)2 | ) |
| СЛЕД(<NEXT\_VAR>)2 |  |
| СЛЕД(<OPER\_LIST>)2 | } |
| СЛЕД(<NEXT\_OPER>)2 |  |
| СЛЕД(<OPERATOR>)2 | ; |
| СЛЕД(<LET>)2 | ) |
| СЛЕД(<EXPR>)2 | ID ; ) < > = |
| СЛЕД(<OPERATION>)2 |  |
| СЛЕД(<COND>)2 | ; ) |
| СЛЕД(<RELATION>)2 |  |

Шаг 5. Для всех нетерминальных символов A во множество СЛЕД(A) включить множества СЛЕД (В), если существует правило B=αA, α ∈ (VT ∪ VN).

|  |  |
| --- | --- |
| СЛЕД(<S>)3 | # |
| СЛЕД(<NEXT>)3 | # |
| СЛЕД(<USING\_LIST>)3 | ; |
| СЛЕД(<NEXT\_USING>)3 | ; |
| СЛЕД(<CLASS>)3 | ; |
| СЛЕД(<CLASS\_BODY>)3 | } |
| СЛЕД(<NEXT\_BODY>)3 | } |
| СЛЕД(<DEF>)3 | ; |
| СЛЕД(<DEF\_LIST>)3 | ; |
| СЛЕД(<NEXT\_DEF>)3 | ; |
| СЛЕД(<VAR\_LIST>)3 | ) |
| СЛЕД(<NEXT\_VAR>)3 | ) |
| СЛЕД(<OPER\_LIST>)3 | } |
| СЛЕД(<NEXT\_OPER>)3 | } |
| СЛЕД(<OPERATOR>)3 | ; |
| СЛЕД(<LET>)3 | ) |
| СЛЕД(<EXPR>)3 | ID ; ) < > = |
| СЛЕД(<OPERATION>)3 | ID ; ) < > = |
| СЛЕД(<COND>)3 | ; ) |
| СЛЕД(<RELATION>)3 | ; ) |

Шаг 6 Так как множества СЛЕД(А) не изменились на шагах 3,4,5, то закончим.

Построение множества ВЫБОР(n). Для формирования множеств ВЫБОР требуется определить аннулирующие правила. Для этого из общего списка правил исключают правила, содержащие в правых частях хотя бы один терминал. Из оставшихся правил исключают непродуктивные правила, т.е. те правила, которые не порождают цепочки терминалов. Оставшиеся правила будут аннулирующими.

Таблица 3 содержит правила, которые не содержат терминальные символы. Слева – непродуктивные правила. Справа – аннулирующие правила.

Таблица 3 – Непродуктивные и аннулирующие правила.

|  |  |
| --- | --- |
| Непродуктивные правила | Аннулирующие правила |
| (24) <OPER\_LIST> -> <OPERATOR> <NEXT\_OPER>  (43) <COND> -> <EXPR> <RELATION> | ( 4) <NEXT> -> e  ( 7) <NEXT\_USING> -> e  (11) <NEXT\_BODY> -> e  (19) <NEXT\_DEF> -> e  (23) <NEXT\_VAR> -> e  (26) <NEXT\_OPER> -> e  (41) <OPERATION> -> e  (47) <RELATION> -> e |

Множества ВЫБОР определим на основании следующих выражений: для не аннулирующих правил - ВЫБОР(n) = ПЕРВ(n), для аннулирующих правил - ВЫБОР(n) = СЛЕД(A), где А – нетерминальный символ в левой части правила n.

|  |  |
| --- | --- |
|  | using |
|  | public |
|  | ; |
|  | # |
|  | ID |
|  | . |
|  | ; |
|  | class |
|  | public |
|  | ; |
|  | } |
|  | uint |
|  | bool |
|  | const |
|  | ID |
|  | ( |
|  | , |
|  | = |
|  | ; |
|  | ID |
|  | , |
|  | = |
|  | ) |
|  | for break continue write read ID |
|  | ; |
|  | } |
|  | for |
|  | break |
|  | continue |
|  | write |
|  | read |
|  | ID |
|  | = |
|  | \* |
|  | / |
|  | ( |
|  | ID |
|  | NUM |
|  | + |
|  | - |
|  | ID ; ) < > = |
|  | ( |
|  | ( ID NUM |
|  | > |
|  | < |
|  | = |
|  | ; ) |

**2.3 Построение управляющей таблицы**

Для составления управляющей таблицы (таблица 4) необходимо определить множество магазинных символов, множество символов поступающих на вход автомата и определить операции МПА для каждой пары: верхний символ магазина – символ на входе автомата.

Множество магазинных символов включает все нетерминальные символы, символ дна магазина и терминальные символы за исключением тех, которые занимают в правых частях правил только первые позиции: # ( ) ; { } <CLASS\_BODY> <CLASS> <COND> <DEF\_LIST> <DEF> <EXPR> <LET> <NEXT\_BODY> <NEXT\_DEF> <NEXT\_OPER> <NEXT\_USING> <NEXT\_VAR> <NEXT> <OPER\_LIST> <OPERATION> <OPERATOR> <RELATION> <S> <USING\_LIST> <VAR\_LIST> = decimal ID.

Множество символов входной строки включает все терминальные символы и символ конца строки: – # ( ) \* , . / ; { } + < = > break class const continue decimal double for ID int NUM public read using writeline.

При построении таблицы использовались следующие обозначения и сокращения:

– Выт. – вытолкнуть верхний символ из магазина;

– Сдв. – сдвинуть входную строку на одну лексему;

– Зам. (n, m) – заменить верхний символ магазина на символы правила номер n, начиная с символа m;

– ДОП. – синтаксический анализ прошел успешно.

Листинг программы, реализующей данную управляющую таблицу, приведен в приложении А.

**3 Описание программы**

Программа выполнена в виде Windows-приложения с оконным интерфейсом (рисунок 2 в среде Microsoft Visual Studio.

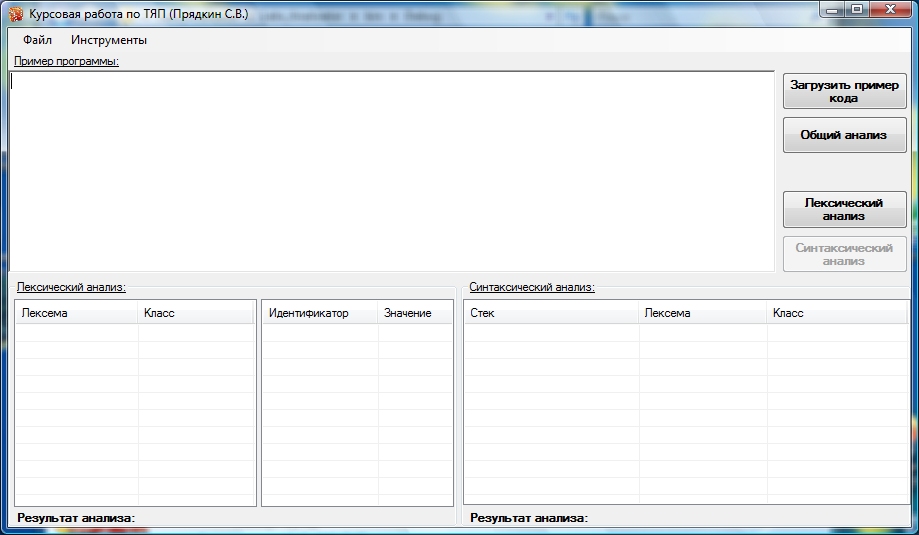


Рисунок 2 - Интерфейс программы

Листинг программы на формальном языке программирования загружается из текстового файла. Результаты работы обоих анализаторов (лексического и синтаксического) отображаются в виде таблиц с помощью компоненты ListView (рисунок 3).

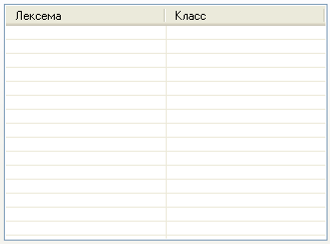


Рисунок 3 - Компонента ListView

Для распознавания ключевых слов используется собственная структура данных, позволяющая реализовать нагруженое дерево (структура определена в п. 2.2).

Программа состоит из нескольких процедур и функций, наиболее важными из которых являются: процедура лексического анализа и процедура синтаксического анализа. Обе они приведены в приложении А. Данные процедуры служат соответствующими обработчиками кнопок «Лексический анализ» и «Синтаксический анализ».

Обмен данными между процедурами анализаторов осуществлен с помощью компонент ListView. Лексический анализатор добавляет в компонент ListView распознанные лексемы, а по завершении анализа эти лексемы в виде таблицы передаются на вход синтаксического анализатора.

**4 Результаты тестирования**

Для тестирования программы использовался следующий код на формальном языке программирования:

using System.Text;

/\* многострочный

комменатрий \*/

public class TestClass

{

public uint a, b = 35, i;

public bool c, d;

public const long int e = 9L;

public uint Main(Param1, Param2)

{

read(a, b);

for(i = 0; i < 10; i = i + 1)

{

write(a, b, c, d, e);

a = b - (c + 2L);

d \*= e - 2;

e /= 123;

break;

};

for(i = 0; i < 10; i \*= 2) { continue; }

}

}

Результаты работы лексического и синтаксического анализаторов показаны на рисунке 4. В качестве результата лексического анализатора представлены две таблицы: таблица лексем и таблица имен. В первой отображены все распознанные лексемы и их класс. Во второй - имена переменных. На выходе синтаксического анализатора получается одна таблица, на которой показан весь процесс анализа, а именно: верхний символ магазина, символ входной лексемы и ее класс.

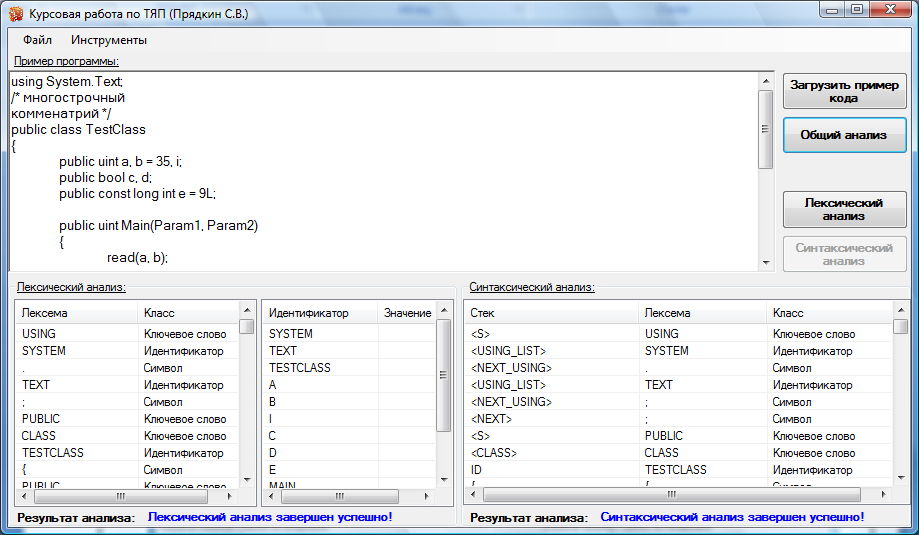


Рисунок 4 - Результат работы анализаторов.

Для проверки правильной работоспособности программы, допустим ошибку в коде:

public bool c, 1d;

В результате программа выдаст сообщение об ошибке на этапе лексического анализа (рисунок 5).

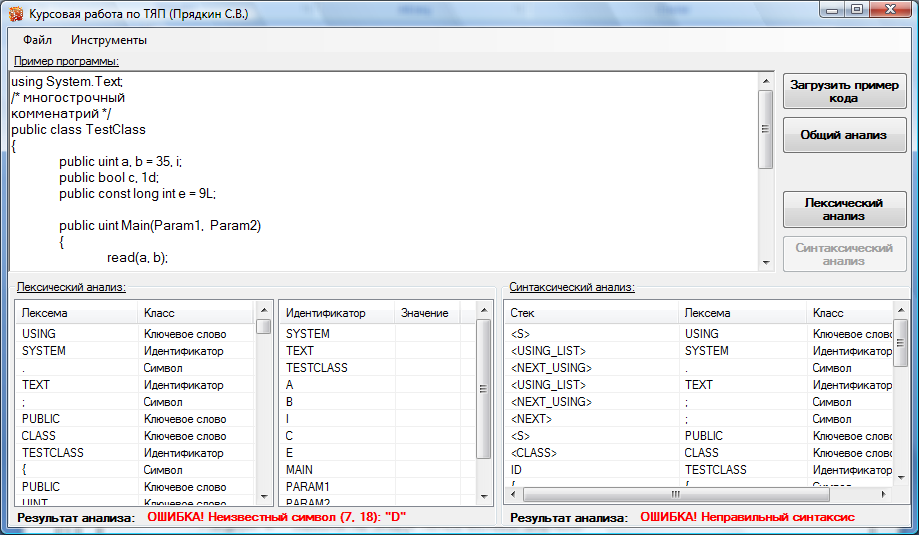


Рисунок 5 - Ошибка на этапе лексического анализа.

Для проверки синтаксического анализатора, сделаем еще одну ошибку:

public const int e = 9L;

На рисунке 6 показан результат работы синтаксического анализатора.

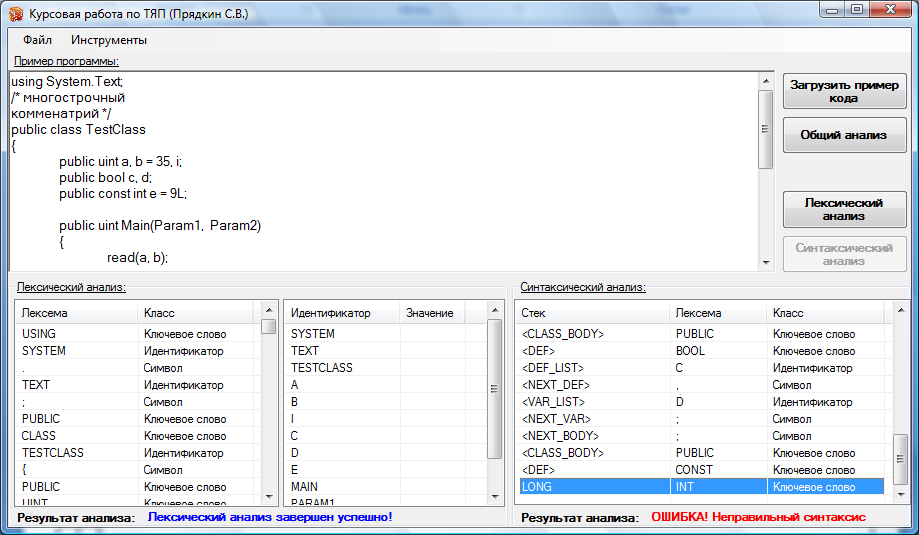


Рисунок 6 - Ошибка на этапе синтаксического анализа.

**5 Руководство пользователя**

Программа запускается из файла Leks\_Analizator.exe. В интерфейсе программы присутствуют четыре кнопки (рисунок 7). При нажатии на кнопку «Загрузить пример кода» (рисунок 7) пользователю будет дана возможность загрузки кода. Также код можно ввести с клавиатуры.

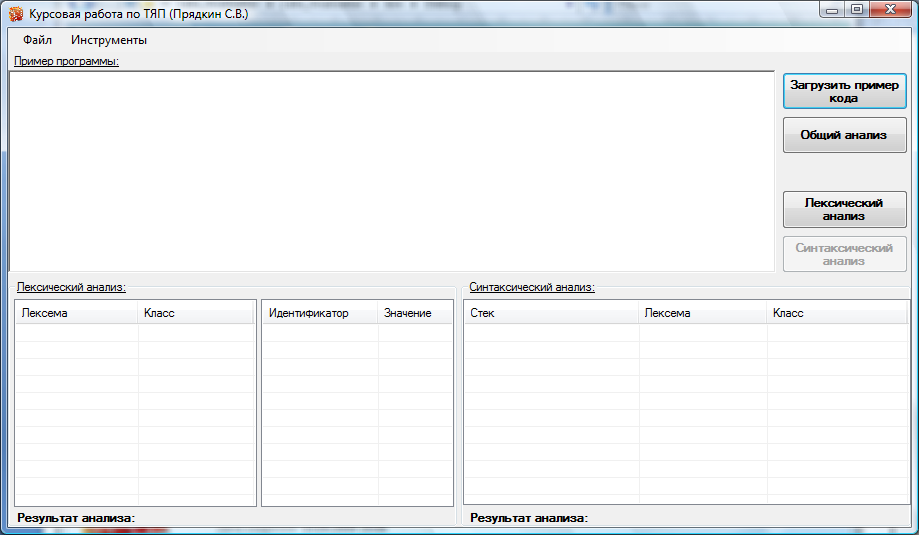


Рисунок 7 - Интерфейс программы

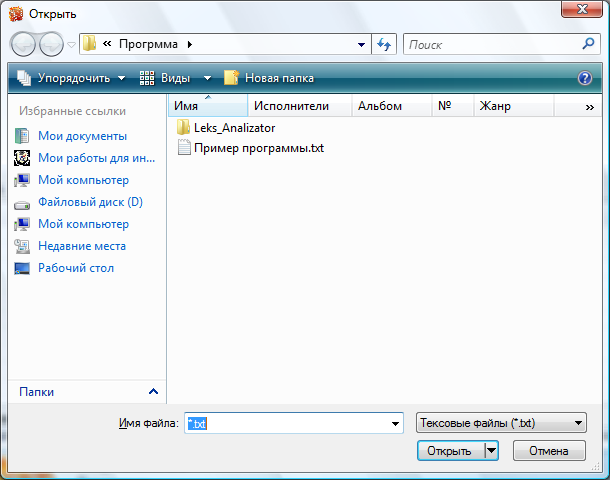


Рисунок 8 - Диалог загрузки файла

После ввода кода программы, при нажатии на кнопку «Лексический анализ», запускается процесс анализа. Результатом анализа будет строчка внизу формы.

По кнопке «Синтаксический анализ» запускается процесс анализа. Результат отображается внизу формы.

При нажатии на кнопке «Общий анализ» будет запущен по порядку лексический анализ, а затем синтаксический анализ.

**Заключение**

В данном курсовом проекте была разработана программа-анализатор, состоящая из двух частей: лексического анализатора, разбивающего исходный текст программы на лексемы и заполняющего таблицу имен; синтаксического анализатора, проверяющего соответствие текста заданной грамматике.

В ходе выполнения курсового проекта были получены следующие результаты:

* написана грамматика для учебного языка программирования;
* спроектирован и построен лексический анализатор;
* спроектирован и построен синтаксический анализатор.

Для написания курсовой работы мною были самостоятельно изучены литературные и справочные материалами по данной теме.

**Список используемой литературы**

1. Ключко В.И. Теория вычислительных процессов и структур. Учебное пособие, -КубГТУ, 1998.

2. Теория вычислительных процессов и структур. Методические указания к курсовой работе для студентов специальности 220400. Составитель проф. В.И.Ключко. -КубГТУ,1997. -27 с.

3. Соколов А.П. Системы программирования: теория, методы, алгоритмы: Учеб. Пособие, - М.: Финансы и статистика, 2004. – 320 с.: ил.

4. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. – СПб.: Питер, 2001. – 736 с.

5. Ахо А, Сети Р., Ульман Д. Компиляторы: принципы, технологии инструменты. : Пер. с англ. – М.: Издательский дом «Вильямс» , 2003. – 768 с.:ил. ISBN5-8459-0189-8(рус), ББК 32.973.26.-018.2.75

6. Вольфенгаген В.Э. Конструкции языков программирования. Приемы описания. - М.: АО «Центр ЮрИнфоР», 2001.-276 с.

7. Карпов В.Э. Теория компиляторов. Учебное пособие — Московский государственный институт электроники и математики. М., 2001. – 79 с (электронный вариант)

8. Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman . Compilers Principles, Techniques, and Tools. 2000. (электронный вариант).

9. Серебряков В.А. Лекции по конструированию компиляторов. Москва, 1993 (электронный вариант).

**Приложение А**

**Листинг лексического анализатора**

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

{

char[] Letters={'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'G',

'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',

'U', 'V', 'W', 'X', 'Y', 'Z', '\_'};

char[] Digits={'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};

char[] Symbols={',', ';', '(', ')', '=', '<', '>', '{', '}', '.', '+', '-', '\*', '/'};

char[] Spaces={' ', char.Parse(Char.ConvertFromUtf32(9))};

LexAnalizerState CommentState = LexAnalizerState.Start;

lvLexTable.Items.Clear();

lvIdTable.Items.Clear();

lLexProgress.Text = "Лексический анализ завершен успешно!";

lLexProgress.ForeColor = Color.Blue;

lLexProgress.Visible = false;

for (int nNumStr = 0; nNumStr < rSrcFile.Lines.Length; nNumStr++)

{

String sSrcLine = rSrcFile.Lines[nNumStr].ToUpper();

if (String.IsNullOrEmpty(sSrcLine)) continue;

int nSrcSymbol = 0;

String sLexeme = "", sClass = "";

LexAnalizerState LexState = CommentState;

while(LexState != LexAnalizerState.Stop)

{

switch(LexState)

{

case LexAnalizerState.Start:

{

if (sSrcLine[nSrcSymbol] == '/')

{

sLexeme += sSrcLine[nSrcSymbol];

nSrcSymbol++;

LexState = LexAnalizerState.Comment;

}

else if (Array.IndexOf(Spaces, sSrcLine[nSrcSymbol]) != -1)

{

nSrcSymbol++;

LexState = (nSrcSymbol == sSrcLine.Length)?

(LexAnalizerState.Stop):(LexAnalizerState.Start);

}

else if (Array.IndexOf(Letters, sSrcLine[nSrcSymbol]) != -1)

{

LexState = LexAnalizerState.Identify;

}

else if (Array.IndexOf(Symbols, sSrcLine[nSrcSymbol]) != -1)

{

sLexeme += sSrcLine[nSrcSymbol];

sClass = "Символ";

nSrcSymbol++;

LexState = LexAnalizerState.Ready;

}

else if (Array.IndexOf(Digits, sSrcLine[nSrcSymbol]) != -1)

{

sLexeme += sSrcLine[nSrcSymbol];

nSrcSymbol++;

LexState = LexAnalizerState.Num;

}

else LexState = LexAnalizerState.Error;

}

break;

case LexAnalizerState.Comment:

{

if (sSrcLine[nSrcSymbol] == '\*')

{

sLexeme = "";

sClass = "";

nSrcSymbol++;

LexState = (nSrcSymbol == sSrcLine.Length)?

(LexAnalizerState.Stop):(LexAnalizerState.CommentStart);

CommentState = LexAnalizerState.CommentStart;

}

else

{

LexState = LexAnalizerState.Ready;

CommentState = LexAnalizerState.Start;

sClass = "Символ";

}

}

break;

case LexAnalizerState.CommentStart:

{

if (sSrcLine[nSrcSymbol] == '\*')

{

nSrcSymbol++;

LexState = (nSrcSymbol == sSrcLine.Length)?

(LexAnalizerState.Stop):(LexAnalizerState.CommentEnd);

CommentState = LexAnalizerState.CommentEnd;

}

else

{

nSrcSymbol++;

LexState = (nSrcSymbol == sSrcLine.Length)?

(LexAnalizerState.Stop):(LexAnalizerState.CommentStart);

}

}

break;

case LexAnalizerState.CommentEnd:

{

if (sSrcLine[nSrcSymbol] == '/')

{

nSrcSymbol++;

LexState = (nSrcSymbol == sSrcLine.Length)?

(LexAnalizerState.Stop):(LexAnalizerState.Start);

CommentState = LexAnalizerState.Start;

}

else

{

nSrcSymbol++;

LexState = (nSrcSymbol == sSrcLine.Length)?

(LexAnalizerState.Stop):(LexAnalizerState.CommentStart);

CommentState = LexAnalizerState.CommentStart;

}

}

break;

case LexAnalizerState.Identify:

{

if ((Array.IndexOf(Digits, sSrcLine[nSrcSymbol]) != -1)

|| (Array.IndexOf(Letters, sSrcLine[nSrcSymbol]) != -1))

{

sLexeme += sSrcLine[nSrcSymbol];

nSrcSymbol++;

LexState = (nSrcSymbol == sSrcLine.Length)?

(LexAnalizerState.Ready):(LexAnalizerState.Identify);

}

else if ((Array.IndexOf(Spaces, sSrcLine[nSrcSymbol]) != -1)

|| (Array.IndexOf(Symbols, sSrcLine[nSrcSymbol]) != -1))

{

LexState = LexAnalizerState.Ready;

}

else LexState = LexAnalizerState.Error;

sClass = "Идентификатор";

}

break;

case LexAnalizerState.Num:

{

if (Array.IndexOf(Digits, sSrcLine[nSrcSymbol]) != -1)

{

sLexeme += sSrcLine[nSrcSymbol];

nSrcSymbol++;

LexState = (nSrcSymbol == sSrcLine.Length)?

(LexAnalizerState.Ready):(LexAnalizerState.Num);

}

else if (sSrcLine[nSrcSymbol] == 'L')

{

sLexeme += sSrcLine[nSrcSymbol];

nSrcSymbol++;

LexState = LexAnalizerState.Ready;

}

else if ((Array.IndexOf(Spaces, sSrcLine[nSrcSymbol]) != -1)

|| (Array.IndexOf(Symbols, sSrcLine[nSrcSymbol]) != -1))

{

LexState = LexAnalizerState.Ready;

}

else LexState = LexAnalizerState.Error;

sClass = "Число";

}

break;

case LexAnalizerState.Ready:

{

sClass = (IsKeyword(sLexeme) == true)?("Ключевое слово"):(sClass);

if (sClass == "Идентификатор") AddId(sLexeme);

AddLexeme(sLexeme, sClass);

sLexeme = "";

LexState = (nSrcSymbol == sSrcLine.Length)?

(LexAnalizerState.Stop):(LexAnalizerState.Start);

}

break;

case LexAnalizerState.Error:

{

lLexProgress.Text = "ОШИБКА! Неизвестный символ (" +

(nNumStr + 1).ToString() + ", " + (nSrcSymbol + 1).ToString() +

"): \"" + sSrcLine[nSrcSymbol].ToString() + "\"";

lLexProgress.ForeColor = Color.Red;

LexState = LexAnalizerState.Stop;

}

break;

}

}

}

AddLexeme("#", "Конец");

lLexProgress.Visible = true;

btnSintax.Enabled = true;

синтаксическийАнализToolStripMenuItem.Enabled = true;

}**Приложение Б.**

**Листинг синтаксического анализатора**

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

{

string[,] Grammatic = {

/\*0\*/{"<S>", "USING", "<USING\_LIST>", "<NEXT>", "", "", "", "", "", "", "", "", "", "", ""},

/\*1\*/{"<S>", "PUBLIC", "<CLASS>", "<NEXT>", "", "", "", "", "", "", "", "", "", "", ""},

/\*2\*/{"<NEXT>", ";", "<S>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*3\*/{"<NEXT>", "e", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*4\*/{"<USING\_LIST>","ID", "<NEXT\_USING>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*5\*/{"<NEXT\_USING>",".", "<USING\_LIST>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*6\*/{"<NEXT\_USING>","e", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*7\*/{"<CLASS>", "CLASS", "ID", "{", "<CLASS\_BODY>", "}", "", "", "", "", "", "", "", "", ""},

/\*8\*/{"<CLASS\_BODY>","PUBLIC", "<DEF>", "<NEXT\_BODY>", "", "", "", "", "", "", "", "", "", "", ""},

/\*9\*/{"<NEXT\_BODY>", ";", "<CLASS\_BODY>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*10\*/{"<NEXT\_BODY>", "e", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*11\*/{"<DEF>", "UINT", "<DEF\_LIST>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*12\*/{"<DEF>", "BOOL", "<DEF\_LIST>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*13\*/{"<DEF>", "CONST", "LONG", "INT", "<DEF\_LIST>", "", "", "", "", "", "", "", "", "", ""},

/\*14\*/{"<DEF\_LIST>", "ID", "<NEXT\_DEF>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*15\*/{"<NEXT\_DEF>", "(", "<VAR\_LIST>", ")", "{", "<OPER\_LIST>", "}", "", "", "", "", "", "", "", ""},

/\*16\*/{"<NEXT\_DEF>", ",", "<VAR\_LIST>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*17\*/{"<NEXT\_DEF>", "=", "<EXPR>", "<NEXT\_VAR>", "", "", "", "", "", "", "", "", "", "", ""},

/\*18\*/{"<NEXT\_DEF>", "e", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*19\*/{"<VAR\_LIST>", "ID", "<NEXT\_VAR>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*20\*/{"<NEXT\_VAR>", ",", "<VAR\_LIST>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*21\*/{"<NEXT\_VAR>", "=", "<EXPR>", "<NEXT\_VAR>", "", "", "", "", "", "", "", "", "", "", ""},

/\*22\*/{"<NEXT\_VAR>", "e", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*23\*/{"<OPER\_LIST>", "<OPERATOR>", "<NEXT\_OPER>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*24\*/{"<NEXT\_OPER>", ";", "<OPER\_LIST>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*25\*/{"<NEXT\_OPER>", "e", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*26\*/{"<OPERATOR>", "FOR", "(", "ID", "=", "<EXPR>", ";", "<COND>", ";", "ID", "<LET>", ")", "{", "<OPER\_LIST>", "}"},

/\*27\*/{"<OPERATOR>", "break", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*28\*/{"<OPERATOR>", "continue", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*29\*/{"<OPERATOR>", "WRITE", "(", "<VAR\_LIST>", ")", "", "", "", "", "", "", "", "", "", ""},

/\*30\*/{"<OPERATOR>", "READ", "(", "<VAR\_LIST>", ")", "", "", "", "", "", "", "", "", "", ""},

/\*31\*/{"<OPERATOR>", "ID", "<LET>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*32\*/{"<LET>", "=", "<EXPR>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*33\*/{"<LET>", "\*", "=", "<EXPR>", "", "", "", "", "", "", "", "", "", "", ""},

/\*34\*/{"<LET>", "/", "=", "<EXPR>", "", "", "", "", "", "", "", "", "", "", ""},

/\*35\*/{"<EXPR>", "(", "<EXPR>", ")", "<OPERATION>", "", "", "", "", "", "", "", "", "", ""},

/\*36\*/{"<EXPR>", "ID", "<OPERATION>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*37\*/{"<EXPR>", "NUM", "<OPERATION>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*38\*/{"<OPERATION>", "+", "<EXPR>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*39\*/{"<OPERATION>", "-", "<EXPR>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*40\*/{"<OPERATION>", "e", "", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*41\*/{"<COND>", "(", "<COND>", ")", "<RELATION>", "", "", "", "", "", "", "", "", "", ""},

/\*42\*/{"<COND>", "<EXPR>", "<RELATION>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*43\*/{"<RELATION>", ">", "<COND>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*44\*/{"<RELATION>", "<", "<COND>", "", "", "", "", "", "", "", "", "", "", "", ""},

/\*45\*/{"<RELATION>", "=", "=", "<COND>", "", "", "", "", "", "", "", "", "", "", ""},

/\*46\*/{"<RELATION>", "e", "", "", "", "", "", "", "", "", "", "", "", "", ""},

};

System.Collections.Stack SintaxStack = new System.Collections.Stack();

SintaxStack.Push("#");

SintaxStack.Push("<S>");

String sTopStack, sLexeme, sClass;

SinAnalizerState SintaxState = SinAnalizerState.InProcess;

lSinProgress.Visible = false;

lvSinTable.Items.Clear();

int nLexNum = 0;

while (SintaxState == SinAnalizerState.InProcess)

{

sTopStack = (String) SintaxStack.Peek();

sLexeme = lvLexTable.Items[nLexNum].SubItems[0].Text;

sClass = lvLexTable.Items[nLexNum].SubItems[1].Text;

String[] subItems = {sTopStack, sLexeme, sClass};

lvSinTable.Items.Add(new ListViewItem(subItems));

if (sTopStack == "#")

{

SintaxState = (sLexeme == "#")?(SinAnalizerState.Accept):(SinAnalizerState.Error);

}

else if (sTopStack == "<S>")

{

if (sLexeme == "#")

{

SintaxStack.Pop();

}

else if (sLexeme == "PUBLIC")

{

Replace(SintaxStack, Grammatic, 1, 2);

nLexNum++;

}

else if (sLexeme == "USING")

{

Replace(SintaxStack, Grammatic, 0, 2);

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "LONG")

{

if (sLexeme == "LONG")

{

SintaxStack.Pop();

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "INT")

{

if (sLexeme == "INT")

{

SintaxStack.Pop();

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "(")

{

if (sLexeme == "(")

{

SintaxStack.Pop();

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == ")")

{

if (sLexeme == ")")

{

SintaxStack.Pop();

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == ";")

{

if (sLexeme == ";")

{

SintaxStack.Pop();

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "{")

{

if (sLexeme == "{")

{

SintaxStack.Pop();

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "}")

{

if (sLexeme == "}")

{

SintaxStack.Pop();

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<CLASS\_BODY>")

{

if (sLexeme == "PUBLIC")

{

Replace(SintaxStack, Grammatic, 8, 2);

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<CLASS>")

{

if (sLexeme == "CLASS")

{

Replace(SintaxStack, Grammatic, 7, 2);

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<DEF\_LIST>")

{

if (sClass == "Идентификатор")

{

Replace(SintaxStack, Grammatic, 14, 2);

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<DEF>")

{

if (sLexeme == "UINT")

{

Replace(SintaxStack, Grammatic, 11, 2);

nLexNum++;

}

else if (sLexeme == "BOOL")

{

Replace(SintaxStack, Grammatic, 12, 2);

nLexNum++;

}

else if (sLexeme == "CONST")

{

Replace(SintaxStack, Grammatic, 13, 2);

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<EXPR>")

{

if (sLexeme == "(")

{

Replace(SintaxStack, Grammatic, 35, 2);

nLexNum++;

}

else if (sClass == "Идентификатор")

{

Replace(SintaxStack, Grammatic, 36, 2);

nLexNum++;

}

else if (sClass == "Число")

{

Replace(SintaxStack, Grammatic, 37, 2);

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<LET>")

{

if (sLexeme == "/")

{

Replace(SintaxStack, Grammatic, 34, 2);

nLexNum++;

}

else if (sLexeme == "\*")

{

Replace(SintaxStack, Grammatic, 33, 2);

nLexNum++;

}

else if (sLexeme == "=")

{

Replace(SintaxStack, Grammatic, 32, 2);

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<NEXT\_BODY>")

{

if (sLexeme == ";")

{

Replace(SintaxStack, Grammatic, 9, 2);

nLexNum++;

}

else if (sLexeme == "}")

{

SintaxStack.Pop();

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<NEXT\_DEF>")

{

if (sLexeme == "(")

{

Replace(SintaxStack, Grammatic, 15, 2);

nLexNum++;

}

else if (sLexeme == ",")

{

Replace(SintaxStack, Grammatic, 16, 2);

nLexNum++;

}

else if (sLexeme == "=")

{

Replace(SintaxStack, Grammatic, 17, 2);

nLexNum++;

}

else if (sLexeme == ";")

{

SintaxStack.Pop();

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<NEXT\_OPER>")

{

if (sLexeme == ";")

{

Replace(SintaxStack, Grammatic, 24, 2);

nLexNum++;

}

else if (sLexeme == "}")

{

SintaxStack.Pop();

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<NEXT\_USING>")

{

if (sLexeme == ".")

{

Replace(SintaxStack, Grammatic, 5, 2);

nLexNum++;

}

else if (sLexeme == ";")

{

SintaxStack.Pop();

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<NEXT\_VAR>")

{

if (sLexeme == ",")

{

Replace(SintaxStack, Grammatic, 20, 2);

nLexNum++;

}

else if (sLexeme == "=")

{

Replace(SintaxStack, Grammatic, 21, 2);

nLexNum++;

}

else if ((sLexeme == ")") || (sLexeme == ";"))

{

SintaxStack.Pop();

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<NEXT>")

{

if (sLexeme == ";")

{

Replace(SintaxStack, Grammatic, 2, 2);

nLexNum++;

}

else if (sLexeme == "#")

{

SintaxStack.Pop();

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<OPER\_LIST>")

{

if ((sLexeme == "FOR") ||

(sLexeme == "CONTINUE") ||

(sLexeme == "BREAK") ||

(sLexeme == "READ") ||

(sLexeme == "WRITE") ||

(sClass == "Идентификатор"))

{

Replace(SintaxStack, Grammatic, 23, 1);

}

else if (sLexeme == "}")

{

SintaxStack.Pop();

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<OPERATION>")

{

if (sLexeme == "+")

{

Replace(SintaxStack, Grammatic, 38, 2);

nLexNum++;

}

else if (sLexeme == "-")

{

Replace(SintaxStack, Grammatic, 39, 2);

nLexNum++;

}

else if ((sLexeme == ")") ||

(sLexeme == ";") ||

(sLexeme == ",") ||

(sLexeme == ">") ||

(sLexeme == "<") ||

(sLexeme == "="))

{

SintaxStack.Pop();

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<OPERATOR>")

{

if (sLexeme == "FOR")

{

Replace(SintaxStack, Grammatic, 26, 2);

nLexNum++;

}

else if (sLexeme == "BREAK")

{

SintaxStack.Pop();

nLexNum++;

}

else if (sLexeme == "CONTINUE")

{

SintaxStack.Pop();

nLexNum++;

}

else if (sLexeme == "READ")

{

Replace(SintaxStack, Grammatic, 30, 2);

nLexNum++;

}

else if (sLexeme == "WRITE")

{

Replace(SintaxStack, Grammatic, 29, 2);

nLexNum++;

}

else if (sClass == "Идентификатор")

{

Replace(SintaxStack, Grammatic, 31, 2);

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<USING\_LIST>")

{

if (sClass == "Идентификатор")

{

Replace(SintaxStack, Grammatic, 4, 2);

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<COND>")

{

if (sLexeme == "(")

{

Replace(SintaxStack, Grammatic, 41, 2);

nLexNum++;

}

else if ((sClass == "Идентификатор") ||

(sClass == "Число"))

{

Replace(SintaxStack, Grammatic, 42, 1);

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<RELATION>")

{

if (sLexeme == ">")

{

Replace(SintaxStack, Grammatic, 43, 2);

nLexNum++;

}

else if (sLexeme == "<")

{

Replace(SintaxStack, Grammatic, 44, 2);

nLexNum++;

}

else if (sLexeme == "=")

{

Replace(SintaxStack, Grammatic, 45, 2);

nLexNum++;

}

else if ((sLexeme == ")") ||

(sLexeme == ";"))

{

SintaxStack.Pop();

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "<VAR\_LIST>")

{

if (sClass == "Идентификатор")

{

Replace(SintaxStack, Grammatic, 19, 2);

nLexNum++;

}

else if (sLexeme == ";")

{

SintaxStack.Pop();

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "=")

{

if (sLexeme == "=")

{

SintaxStack.Pop();

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else if (sTopStack == "ID")

{

if (sClass == "Идентификатор")

{

SintaxStack.Pop();

nLexNum++;

}

else SintaxState = SinAnalizerState.Error;

}

else SintaxState = SinAnalizerState.Error;

if (SintaxState == SinAnalizerState.Error)

{

lSinProgress.Text = "ОШИБКА! Неправильный синтаксис";

lSinProgress.ForeColor = Color.Red;

}

else if (SintaxState == SinAnalizerState.Accept)

{

lSinProgress.Text = "Синтаксический анализ завершен успешно!";

lSinProgress.ForeColor = Color.Blue;

}

}

lSinProgress.Visible = true;

btnSintax.Enabled = false;

синтаксическийАнализToolStripMenuItem.Enabled = false;

}

Цель данного проекта: практическое применение теории формальных грамматик и теории автоматов для проектирования трансляторов.

Постановка задания:Спроектировать и построить лексический и синтаксический анализаторы для заданного формального языка программирования. Построить и реализовать в лексическом анализаторе распознаватель лексем с заданной структурой: Нагруженное дерево (один элемент дерева хранит один символ (букву входного алфавита) лексемы).

Учебный язык включает директиву using, функцию main(), описание переменных, констант, массива переменной длины, операторов присваивания, арифметические операции. За основу лексики, синтаксиса и семантики учебного языка принять стандарты языка программирования С#.

Рассмотренная регулярная грамматика может быть реализована автоматом с множеством состояний G: [S, I, C, E, R], где

S - состояние ожидания первого символа лексемы;

I - состояние ожидания символов идентификаторов: буквы, цифры;

С - состояние ожидания символов целой части числа;

E -состояние ошибки;

R - состояние допуска лексемы.

Автомат переходит в допустимое состояние R из состояний S, I, C по символу конца лексемы - пробел или разделительного знака. Автомат определен множеством перечисленных состояний, множеством правил [1, 2, … , 47] и множеством входных символов [00..9, A..Z, «–», «#», «(», «)», «\*», «,», «.», «/», «:», «;», «{«, «}», «+», «=»], из которых символы подмножества [«–», «#», «(», «)», «\*», «,», «/», «:», «;», «{«, «}», «+», «=»] являются одновременно разделительными символами и уникальными лексемами. Таблица переходов автомата распознавателя идентификаторов представлена ниже на таблице.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | L | D | e |  |
| S | I | E | S | Нач. состояние |
| I | I | I | R |  |
| E | Ошибка | | | |
| R | Допустить | | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Множества ВЫБОР:   |  |  | | --- | --- | |  | using | |  | public | |  | ; | |  | # | |  | ID | |  | . | |  | ; | |  | class | |  | public | |  | ; | |  | } | |  | uint | |  | bool | |  | const | |  | ID | |  | ( | |  | , | |  | = | |  | ; | |  | ID | |  | , | |  | = | |  | ) | |  | for break continue write read ID | |  | ; | |  | } | |  | for | |  | break | |  | continue | |  | write | |  | read | |  | ID | |  | = | |  | \* | |  | / | |  | ( | |  | ID | |  | NUM | |  | + | |  | - | |  | ID ; ) < > = | |  | ( | |  | ( ID NUM | |  | > | |  | < | |  | = | |  | ; ) | | Формальная грамматика языка программирования:  Грамматика целевого символа:  ( 1) <S> -> using <USING\_LIST> <NEXT>  ( 2) <S> -> public <CLASS> <NEXT>  ( 3) <NEXT> -> ; <S>  ( 4) <NEXT> -> e  Грамматика описания using:  ( 5) <USING\_LIST> -> ID <NEXT\_USING>  ( 6) <NEXT\_USING> -> . <USING\_LIST>  ( 7) <NEXT\_USING> -> e  Грамматика описания класса:  ( 8) <CLASS> -> class ID { <CLASS\_BODY> }  ( 9) <CLASS\_BODY> -> public <DEF> <NEXT\_BODY>  (10) <NEXT\_BODY> -> ; <CLASS\_BODY>  (11) <NEXT\_BODY> -> e  Грамматика описания определения полей и методов класса:  (12) <DEF> -> uint <DEF\_LIST>  (13) <DEF> -> bool <DEF\_LIST>  (14) <DEF> -> const long int <DEF\_LIST>  (15) <DEF\_LIST> -> ID <NEXT\_DEF>  (16) <NEXT\_DEF> -> ( <VAR\_LIST> ) { <OPER\_LIST> }  (17) <NEXT\_DEF> -> , <VAR\_LIST>  (18) <NEXT\_DEF> -> = <EXPR> <VAR\_LIST>  (19) <NEXT\_DEF> -> e  (20) <VAR\_LIST> -> ID <NEXT\_VAR>  (21) <NEXT\_VAR> -> , <VAR\_LIST>  (22) <NEXT\_VAR> -> = <EXPR> <VAR\_LIST>  (23) <NEXT\_VAR> -> e  Грамматика описания списка операторов:  (24) <OPER\_LIST> -> <OPERATOR> <NEXT\_OPER>  (25) <NEXT\_OPER> -> ; <OPER\_LIST>  (26) <NEXT\_OPER> -> e | Грамматика описания операторов:  (27) <OPERATOR> -> for ( ID = <EXPR> ; <COND> ; ID <LET> ) { <OPER\_LIST> }  (28) <OPERATOR> -> break  (29) <OPERATOR> -> continue  (30) <OPERATOR> -> write ( <VAR\_LIST> )  (31) <OPERATOR> -> read ( <VAR\_LIST> )  (32) <OPERATOR> -> ID <LET>  (33) <LET> -> = <EXPR>  (34) <LET> -> \* = <EXPR>  (35) <LET> -> / = <EXPR>  Грамматика описания арифметического выражения:  (36) <EXPR> -> ( <EXPR> ) <OPERATION>  (37) <EXPR> -> ID <OPERATION>  (38) <EXPR> -> NUM <OPERATION>  (39) <OPERATION> -> + <EXPR>  (40) <OPERATION> -> - <EXPR>  (41) <OPERATION> -> e  Грамматика описания условия:  (42) <COND> -> ( <COND> ) <RELATION>  (43) <COND> -> <EXPR> <RELATION>  (44) <RELATION> -> > <COND>  (45) <RELATION> -> < <COND>  (46) <RELATION> -> = = <COND>  (47) <RELATION> -> e |

Управляющая таблица синтаксического анализатора

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | / | - | # | ( | ) | \* | , | . | ; | { | } | + | < | = | > | bool | break | class | const | continue | for | ID | int | long | NUM | public | read | using | write |
| # |  |  | Д |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ( |  |  |  | В С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ) |  |  |  |  | В С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ; |  |  |  |  |  |  |  |  | В С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| { |  |  |  |  |  |  |  |  |  | В С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| } |  |  |  |  |  |  |  |  |  |  | В С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <CLASS\_BODY> |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | З(9, 2) С |  |  |  |
| <CLASS> |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | З(8, 2) С |  |  |  |  |  |  |  |  |  |  |  |
| <COND> |  |  |  | З(42, 2) С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | З(43, 1) |  |  | З(43, 1) |  |  |  |  |
| <DEF\_LIST> |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | З(15, 2) С |  |  |  |  |  |  |  |
| <DEF> |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | З(13, 2) С |  |  | З(14, 2) С |  |  |  |  |  |  |  |  |  |  |
| <EXPR> |  |  |  | З(36, 2) С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | З(37, 2) С |  |  | З(38, 2) С |  |  |  |  |
| <LET> | З(35, 2) С |  |  |  |  | З(34, 2) С |  |  |  |  |  |  |  | З(33, 2) С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <NEXT\_BODY> |  |  |  |  |  |  |  |  | З(10, 2) С |  | В |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <NEXT\_DEF> |  |  |  | З(16, 2) С |  |  | З(17, 2) С |  | В |  |  |  |  | З(18, 2) С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <NEXT\_OPER> |  |  |  |  |  |  |  |  | З(25, 2) С |  | В |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <NEXT\_USING> |  |  |  |  |  |  |  | З(6, 2) С | В |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <NEXT\_VAR> |  |  |  |  | В |  | З(21, 2) С |  | В |  |  |  |  | З(22, 2) С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <NEXT> |  |  | В |  |  |  |  |  | З(3, 2) С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <OPER\_LIST> |  |  |  |  |  |  |  |  |  |  | В |  |  |  |  |  | З(24, 1) |  |  | З(24, 1) | З(24, 1) | З(24, 1) |  |  |  |  | З(24, 1) |  | З(24, 1) |
| <OPERATION> |  | З(40, 2) С |  |  | В |  |  |  | В |  |  | З(39, 2) С | В | В | В |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <OPERATOR> |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | В С |  |  | В С | З(27, 2) С | З(32, 2) С |  |  |  |  | З(31, 2) С |  | З(30, 2) С |
| <RELATION> |  |  |  |  | В |  |  |  | В |  |  |  | З(45, 2) С | З(46, 2) С | З(44, 2) С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| <S> |  |  | В |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | З(2, 2) С |  | З(1, 2) С |  |
| <USING\_LIST> |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | З(5, 2) С |  |  |  |  |  |  |  |
| <VAR\_LIST> |  |  |  |  |  |  |  |  | В |  |  |  |  |  |  |  |  |  |  |  |  | З(20, 2) С |  |  |  |  |  |  |  |
| = |  |  |  |  |  |  |  |  |  |  |  |  |  | В С |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| ID |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | В С |  |  |  |  |  |  |  |
| int |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | В С |  |  |  |  |  |  |
| long |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | В С |  |  |  |  |  |