**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«Московский энергетический институт»**

**Кафедра математического и компьютерного моделирования**

**«Математическое обеспечение ЭВМ»**

Лабораторная работа №2

**"Построение лексического блока"**

Вариант №11

Выполнил: Симаков А.М.

Группа: А-16-20

Преподаватель: Князев А.В.

**Москва 2021г**

1. **Задание.**

Общее:

Оператор присваивания:

<ид.>:=<ар.выр.>;

Условный оператор:

if(<лог.выр.>) then <совок. операторов> [else < совок.

операторов>] endif

Оператор цикла:

ДЛЯ <ид.>:=<F> ДО <F> [ШАГ <F>] ЦИКЛ <совок. опе-

раторов> КОНЕЦ

Арифметическое выражение:

<E>::=<T><E-список>

<E-список>::= +<T><E-список>

<E-список>::= ε

<T>::=<F><T-список>

<T-список>::= \*<F><T-список>

<T-список>::= ε

<F>::=<Id>

<F>::=<Int>

Логическое выражение:

<лог.выр.>::=<F><лог.опер.><F>

<лог.опер.>::= =

<лог.опер.>::= !=

Пример программы:

a:=16\*3+1;

b:=11+2\*a;

c:=3\*a+2;

if(b!=c) then a:=4\*b; b:=12; else a:=2\*b+3; endif

k:=0; s:=0;

ДЛЯ i:=1 ДО 10 ШАГ 2 ЦИКЛ

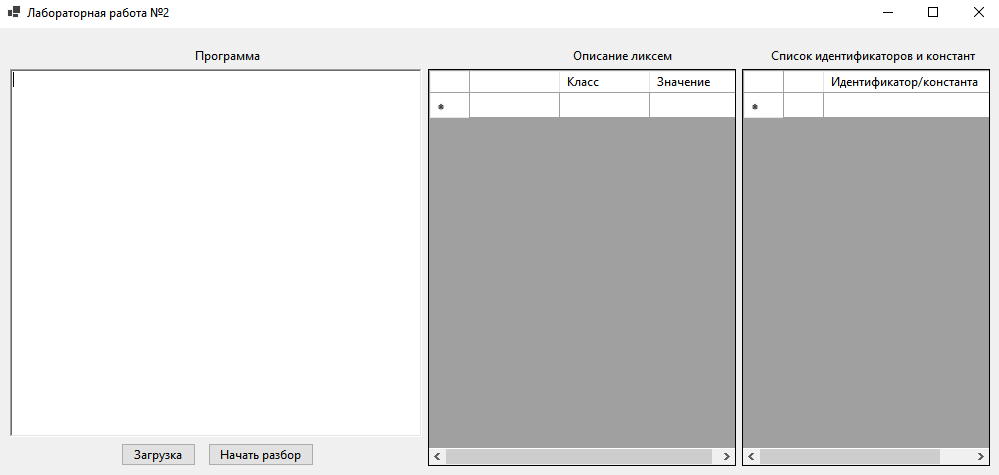
k:=k+1;

s:=s+k;

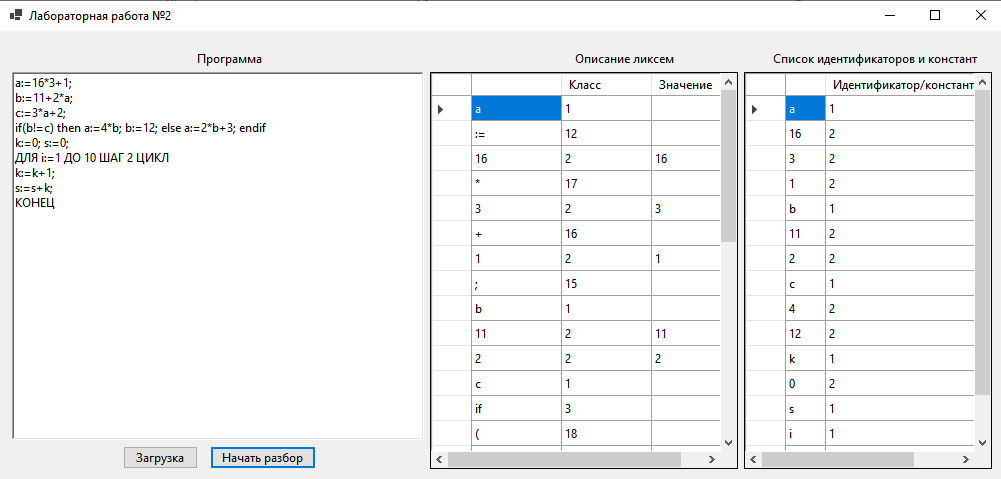
КОНЕЦ

1. **Описание работы программы**

Выводится форма с одним текстовым полем и двумя пустыми таблицами.



Вводим код в текстовое поле и нажимаем «Лексический анализ». Код можно  
загрузить из файла. После чего в правой таблице отобразятся все найденные константы и идентификаторы, а в центральной – все лексемы данного кода, их класс и значением.

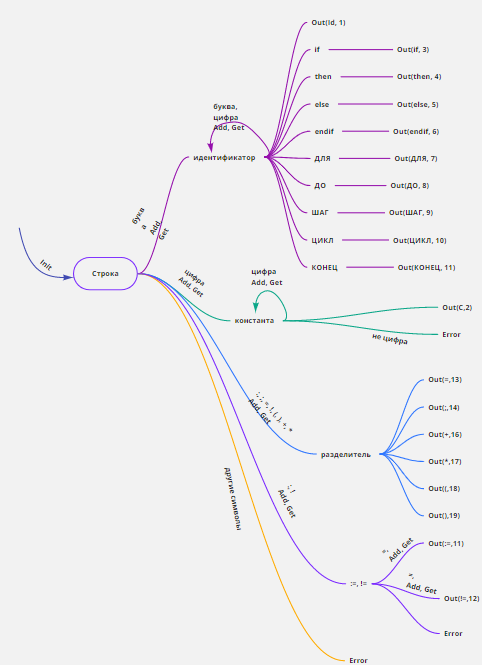


1. **Грамматика для лексем**

<идентификатор> = <идентификатор>цифра / <идентификатор>буква / буква  
<целое> = <целое>цифра / цифра  
<разделитель> = + / \* / ( / ) / ; / := / !=

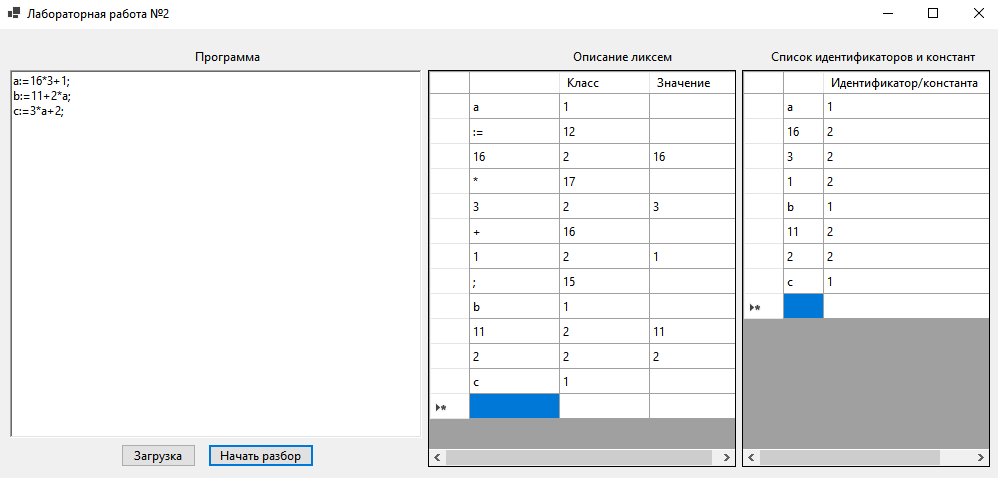
Буква – терминальный символ, представляющий собой любую букву латинского и русского алфавита,   
цифра – терминальный символ, представляющий собой любую цифру

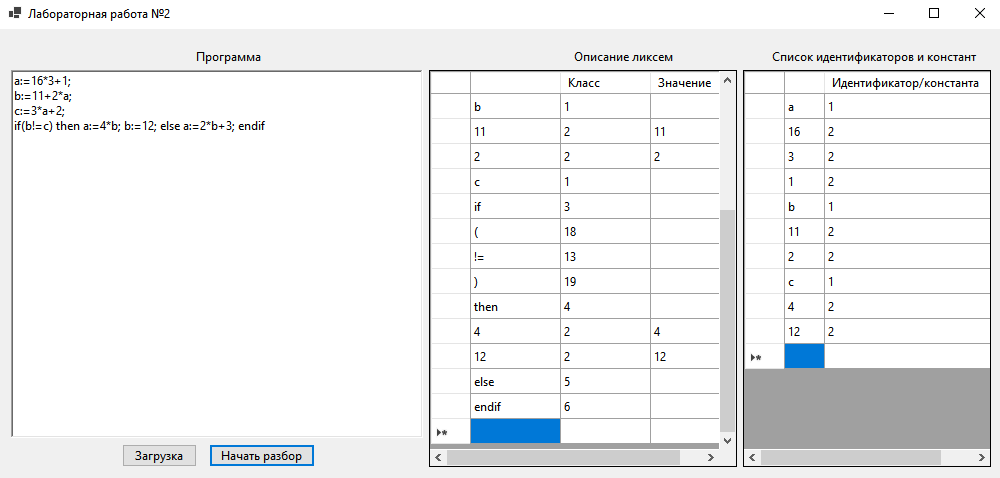
1. **Граф лексического блока**

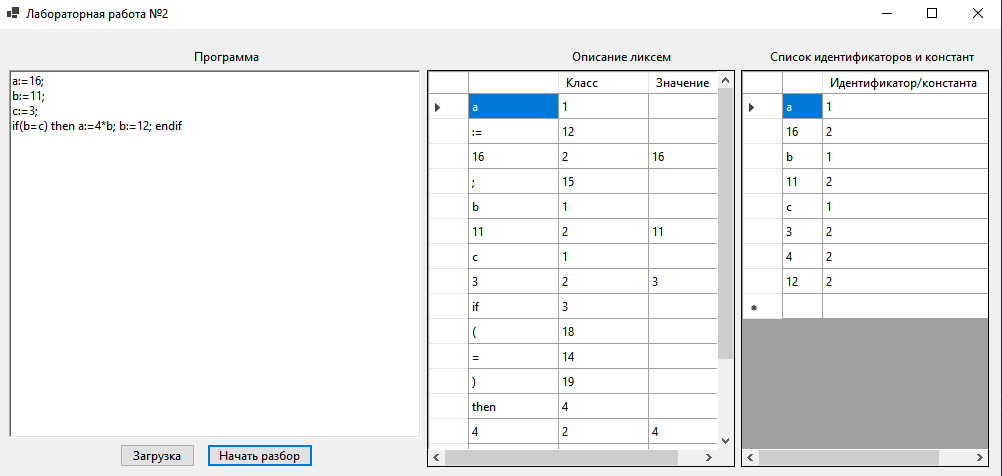


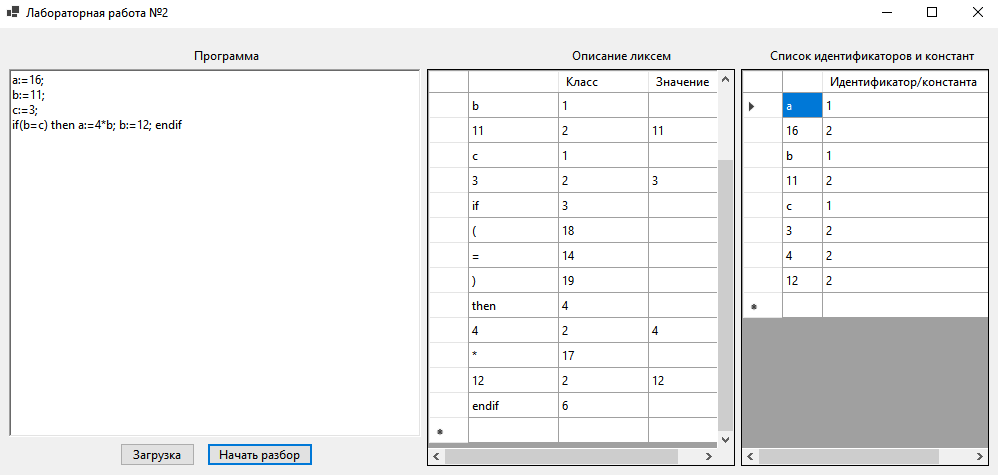
1. **Алгоритмы выполнения основных операций на псевдокоде**  
    Начальное состояние -1  
    **Пока** состояние = -1:  
    Проверка на ошибки  
    Анализируем текущий символ  
    **Если** символ – не (пробел или перенос строки или табуляция) или (разделитель)  
    **то** **если** символ – идентификатор  
    **то** ставим текущее состояние 1  
    добавляем символ в стек  
    **иначе** **если** стек – целое  
    **то** ставим текущее состояние 1  
    добавляем символ в стек  
    **иначе** ставим текущее состояние 3  
    Сдвигаемся на одну позицию  
     
    текущий символ на текущей позиции  
    **если** состояние - 3  
    **то** Возвращаем лексему и ее класс  
    Ставим текущее состояние <none>  
    Сдвигаемся на одну позицию  
      
    Сдвигаемся на одну позицию  
    текущий символ на текущей позиции  
    Анализируем текущий символ  
    **Пока** символ – не разделитель  
    Проверка на ошибки  
    Добавляем символ в стек  
    Сдвигаемся на одну позицию  
    текущий символ на текущей позиции  
    Анализируем текущий символ  
     
    **Если** стек – ключевое слово  
    **то** возвращаем стек и соответствующий класс  
    Ставим текущее состояние -1  
     
    Возвращаем стек и соответствующий класс   
    Ставим текущее состояние -1

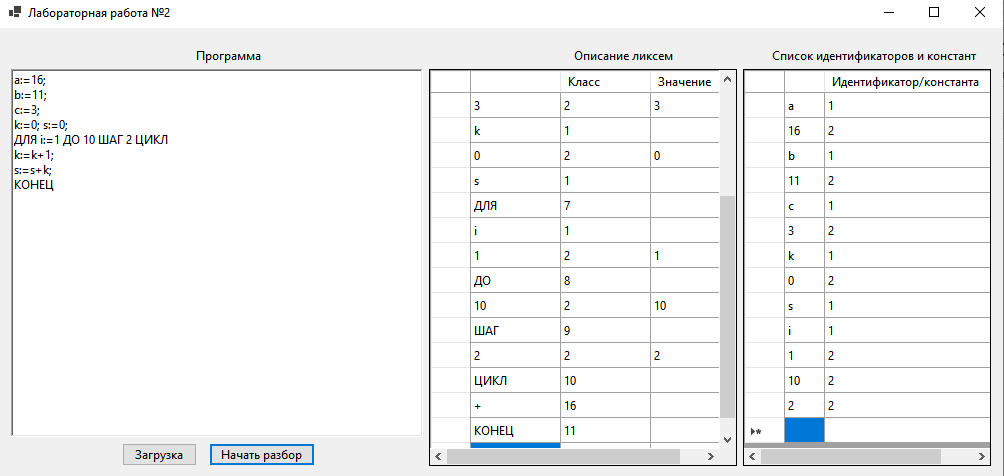
1. **Тесты**

Арифметические операции 

Условный оператор   
 





Цикл  
****

1. **Листинг программы**

Form1.cs

using System;

using System.Windows.Forms;

using System.IO;

namespace L2Math

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

private bool NotInMass(string ch, string[,] m, int n)

{

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

{

if (String.Compare(ch, m[i, 0]) == 0)

return false;

}

return true;

}

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

{

string pro = richTextBox1.Text;

int rows1 = 0;

int rows2 = 0;

string[ , ] mr1 = new string[200, 200];

string[ , ] mr2 = new string[100, 100];

Lexer lexer = new Lexer(pro);

Lexem lexem = lexer.GetLex();

while (lexem != null)

{

if (lexem.GetLexemClass() == "error")

{

MessageBox.Show("Обнаружена ошибка");

break;

}

if (NotInMass(lexem.GetLexem(), mr1, rows1))

{

mr1[rows1, 0] = lexem.GetLexem();

mr1[rows1, 1] = lexem.GetLexemClass();

rows1 += 1;

if (Convert.ToInt32(lexem.GetLexemClass()) == 1 || Convert.ToInt32(lexem.GetLexemClass()) == 2)

{

mr2[rows2, 0] = lexem.GetLexem();

mr2[rows2, 1] = lexem.GetLexemClass();

rows2 += 1;

}

}

lexem = lexer.GetLex();

}

dataGridView1.RowCount = 1;

dataGridView2.RowCount = 1;

int k = 0;

while (k != rows1)

{

dataGridView1.RowCount += 1;

k += 1;

}

k = 0;

while (k != rows2)

{

dataGridView2.RowCount += 1;

k += 1;

}

k = 0;

while (k != rows1)

{

dataGridView1[0, k].Value = mr1[k, 0];

dataGridView1[1, k].Value = mr1[k, 1];

if (mr1[k, 1] == "2")

dataGridView1[2, k].Value = mr1[k, 0];

k += 1;

}

k = 0;

while (k != rows2)

{

dataGridView2[0, k].Value = mr2[k, 0];

dataGridView2[1, k].Value = mr2[k, 1];

k += 1;

}

}

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

{

if (File.Exists("test.txt"))

{

string[] readText = File.ReadAllLines("test.txt");

foreach (var line in readText)

{

richTextBox1.SelectedText = line;

richTextBox1.SelectedText = "\n";

}

}

}

}

}

Lexer.cs

using System;

namespace L2Math

{

class Lexer

{

private int position = 0;

private string text;

private string buffer = "";

private int state = -1;

public Lexer(string str)

{

text = str;

}

private bool control(char ch)

{

if (ch == ' ' || ch == '\n' || ch == '\t')

return true;

return false;

}

private bool IsKeyword(string str)

{

if (String.Compare(str, "if") == 0 || String.Compare(str, "then") == 0 || String.Compare(str, "else") == 0 || String.Compare(str, "endif") == 0 || String.Compare(str, "ДЛЯ") == 0 || String.Compare(str, "ДО") == 0 || String.Compare(str, "ШАГ") == 0 || String.Compare(str, "ЦИКЛ") == 0 || String.Compare(str, "КОНЕЦ") == 0)

return true;

return false;

}

private bool IsIdentifier(char ch)

{

if (ch >= 'a' && ch <= 'z' || ch >= 'A' && ch <= 'Z' || ch >= 'а' && ch <= 'я' || ch >= 'А' && ch <= 'Я')

return true;

return false;

}

private bool IsInteger(char ch)

{

if (ch >= '0' && ch <= '9')

return true;

return false;

}

private bool IsSeperate(char ch)

{

if (ch == '=' || ch == ';' || ch == '+' || ch == '\*' || ch == '!' || ch == ':' || ch == '(' || ch == ')')

return true;

return false;

}

private int keywords(string str)

{

switch (str)

{

case "if":

return 3;

case "then":

return 4;

case "else":

return 5;

case "endif":

return 6;

case "ДЛЯ":

return 7;

case "ДО":

return 8;

case "ШАГ":

return 9;

case "ЦИКЛ":

return 10;

case "КОНЕЦ":

return 11;

}

return -1;

}

private int seperates(char ch)

{

switch (ch)

{

case '=':

return 14;

case ';':

return 15;

case '+':

return 16;

case '\*':

return 17;

case '(':

return 18;

case ')':

return 19;

}

return -1;

}

private int ConvertState(int state, char ch)

{

if (state == 3)

{

if (ch == ':')

{

position += 1;

ch = text[position];

if (ch == '=')

return 12;

else

return -1;

}

else

{

if (ch == '!')

{

position += 1;

ch = text[position];

if (ch == '=')

return 13;

else

return -1;

}

else

return seperates(ch);

}

}

return state;

}

private int LexAnalysis(char ch)

{

if (control(ch))

return -1;

if (IsSeperate(ch))

return 3;

if (IsIdentifier(ch))

return 1;

if (IsInteger(ch))

return 2;

return -2;

}

public Lexem GetLex()

{

if (position >= text.Length)

return null;

char ch;

int analysis;

while (state == -1)

{

if (position >= text.Length)

return null;

ch = text[position];

analysis = LexAnalysis(ch);

if (analysis == -2)

{

Lexem lexem0 = new Lexem("\0", "error");

return lexem0;

}

if (analysis != -1)

{

if (analysis == 1)

{

state = 1;

buffer = Convert.ToString(ch);

}

else if (analysis == 2)

{

state = 2;

buffer = Convert.ToString(ch);

}

else state = 3;

}

else

position += 1;

}

ch = text[position];

if (state == 3)

{

string conv = Convert.ToString(ConvertState(state, ch));

Lexem lexem1;

if (conv == "12")

{

lexem1 = new Lexem(":=", conv);

state = -1;

position += 1;

return lexem1;

}

if (conv == "13")

{

lexem1 = new Lexem("!=", conv);

state = -1;

position += 1;

return lexem1;

}

Lexem lexem0 = new Lexem(Convert.ToString(ch), Convert.ToString(ConvertState(state, ch)));

state = -1;

position += 1;

return lexem0;

}

position += 1;

if (position >= text.Length - 1)

{

Lexem lexem0 = new Lexem(buffer, Convert.ToString(ConvertState(state, ch)));

position += 1;

state = -1;

return lexem0;

}

ch = text[position];

analysis = LexAnalysis(ch);

while (analysis != -1 && analysis < 3 && position < text.Length)

{

if (state == 2 && analysis == 1 || analysis == -2)

{

Lexem lexem0 = new Lexem("\0", "error");

return lexem0;

}

buffer = String.Concat(buffer, ch);

position += 1;

if (position == text.Length)

break;

ch = text[position];

analysis = LexAnalysis(ch);

}

if (IsKeyword(buffer))

{

Lexem lexem0 = new Lexem(buffer, Convert.ToString(keywords(buffer)));

state = -1;

return lexem0;

}

Lexem lexem = new Lexem(buffer, Convert.ToString(ConvertState(state, ch)));

state = -1;

return lexem;

}

}

class Lexem

{

private string lexem;

private string lexem\_class;

public Lexem(string lexem, string lexem\_class)

{

this.lexem = lexem;

this.lexem\_class = lexem\_class;

}

public string GetLexem()

{

return lexem;

}

public string GetLexemClass()

{

return lexem\_class;

}

}

}