**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ**

**«Московский энергетический институт»**

**Кафедра математического и компьютерного моделирования**

**«Математическое обеспечение ЭВМ»**

Лабораторная работа №4

**"Разработка интерпретатора"**

Вариант №11

Выполнил: Симаков А.М.

Группа: А-16-20

Преподаватель: Князев А.В.

**Москва 2021г**

**Постановка задачи**

*Общее:*

Преобразовать заданную грамматику в L-атрибутную транслирующую

грамматику в форме простого присваивания.

2. Разработать объектный язык.

3. Разработать атрибутный автомат для трансляции предложений данной

грамматики.

4. Разработать программу, реализующую МП-транслятор. Программа

разрабатывается как приложение с графическим интерфейсом на языке C#

в среде Visual Studio.

5. Транслятор разрабатывается как программа, реализующая алгоритмы,

изложенные в теоретическом курсе. Не должны использоваться

вспомогательные средства: коллекции, регулярные выражения, функции

разбора строк и т.д.

6. Разработать программу-интерпретатор для созданного объектного языка.

*Индивидуальное:*

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

**LL(1) – грамматика**

1. <Program> ::= <S>
2. <Program> ::= ɛ
3. <S>::=I:=<E>;<Program>
4. <S>::= if (<Log>)then<S><SList>endif <Program>
5. <S>::= ДЛЯ I:=<F>ДО<F><Step>ЦИКЛ<S> КОНЕЦ
6. <E>::=<T><EList>
7. <EList >::= +<T><EList>
8. <EList >::= ε
9. <T>::=<F><TList>
10. <TList>::= \*<F><TList>
11. <TList>::= ε
12. <F>::= I
13. <F>::= C
14. <SList>:= else<S>
15. <SList>:= ε
16. <Log>.::=<F><LopOp>
17. <LogOp>::= =<F>
18. <LogOp>::= !=<F>
19. <Step>::=ШАГ<F>
20. <Step>::= ε

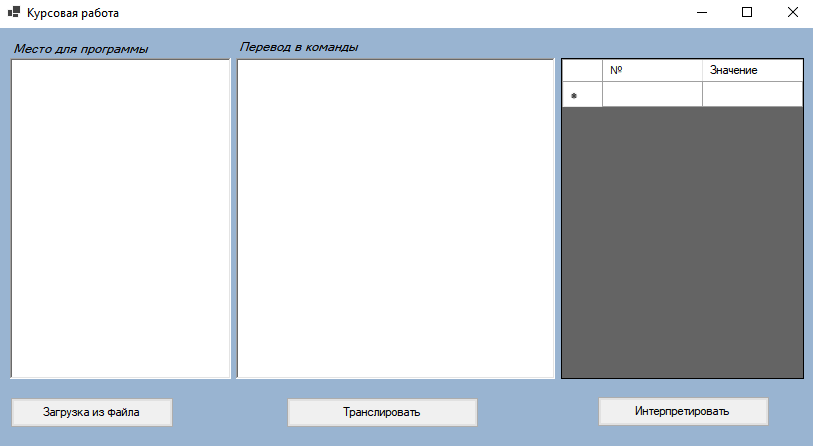
**Объектный язык**

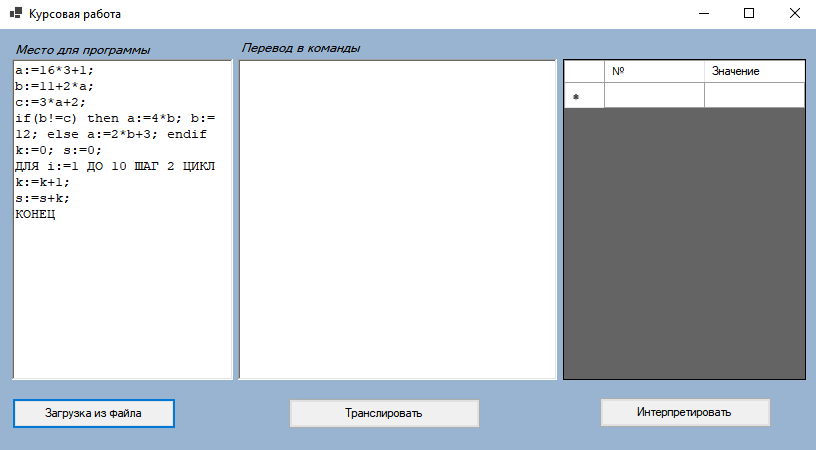
1. Присвоить(v1, v2) – копирует значение из ячейки v1 в v2
2. Сложить(v1, v2, r) – записывает результат сложения значений ячеек v1 и v2 в ячейку r
3. Умножить(v1, v2, r) – записывает умножения значения ячеек v1 и v2 в ячейку r
4. Равно(v1, v2, r) – записывает результат сравнения значений ячеек v1 и v2 в ячейку r, если значение первой ячейки больше второй, то записывается true и наоборот
5. Не равно(v1, v2, r) – записывает результат сравнения значений ячеек v1 и v2 в ячейку r, если значение первой ячейки меньше второй, то записывается true и наоборот
6. Условный переход (if) (v1, num) – если значение ячейки v1 = false, то продолжаем обработку команд с num-той команды
7. Безусловный переход (if) (v1, num) – если значение ячейки v1 = true, то продолжаем обработку команд с num-той команды
8. Инкрементировать(v1 {+step}) – увеличивает значение ячейки по адресу v1 на величину step. Если шаг не задан, то step по умолчанию 1.
9. Проверка условий цикла(v1, v2, num) – если значение ячейки v1 превышает значения ячейки v2, то продолжаем обработку команд с num-той команды
10. Безусловный переход (цикл) (num) – продолжаем обработку команд с num-той команды

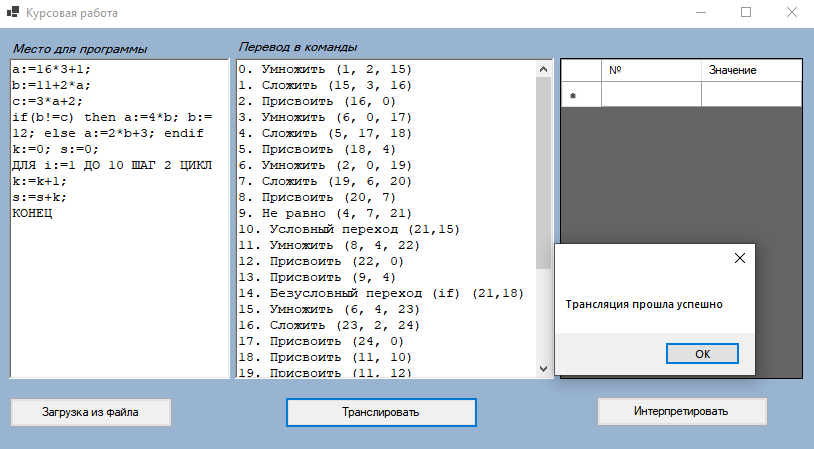
**Описание интерфейса программы**

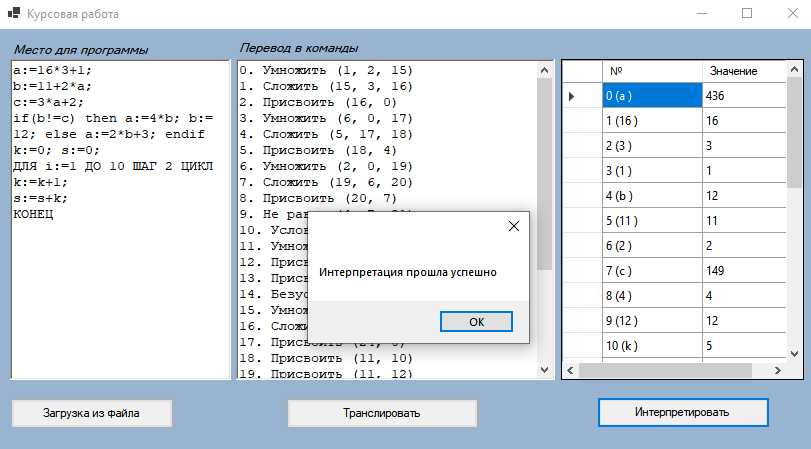
Выводиться форма с двумя текствовыми полнями, таблицей и тремя кнопками.  
В первое текстовое поле вводим код программы или загружаем его из файла. Нажимаем «Транслировать». Во втором текстовом поле выведуться команды, сгенерированные транслятором. В каждой команде пишется адрес ячейки, к которой будет обращаться интепретатор для ее реализации. Данные ячейки со значениями оотображаются в таблице.

Нажимаем «Интерперетировать», чтобы началась интерпретация.

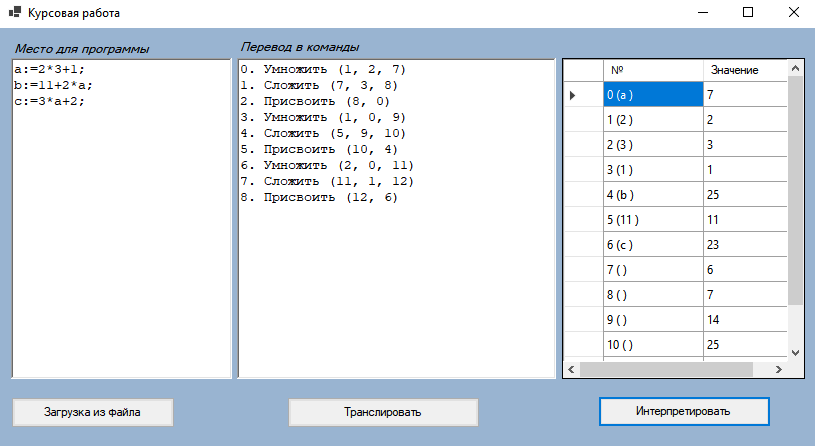




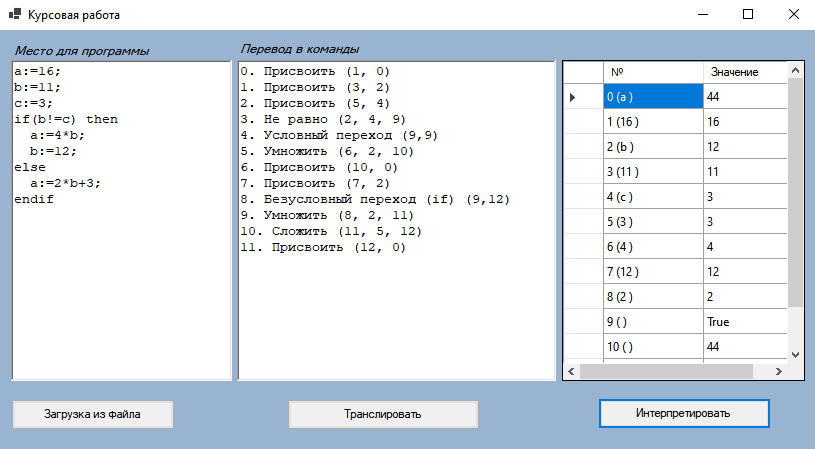


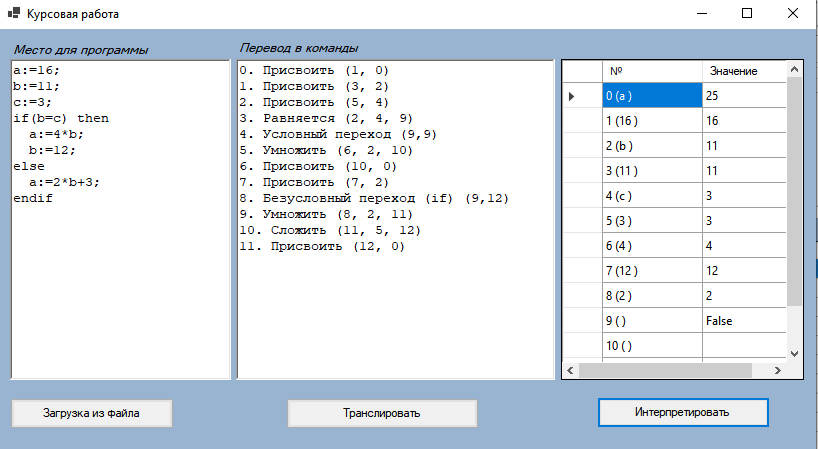


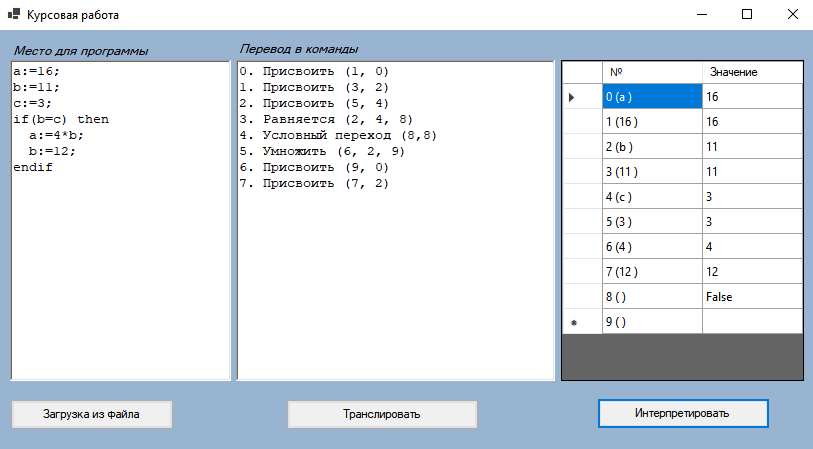
**Тесты**

Присваивание  


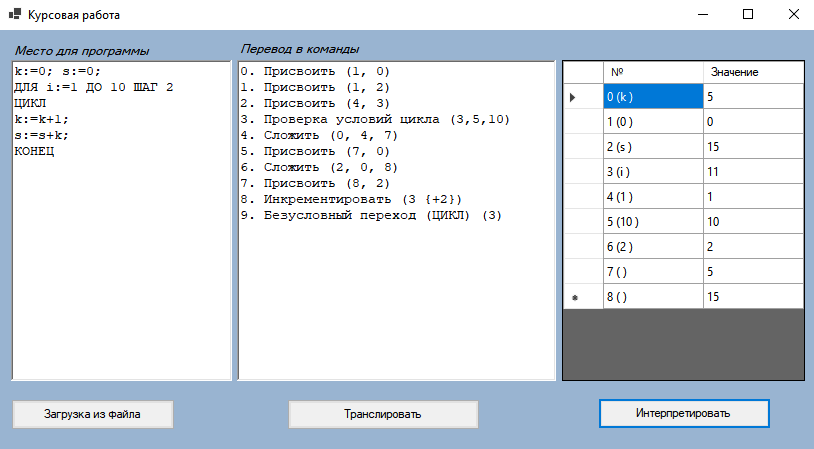
Условный оператор

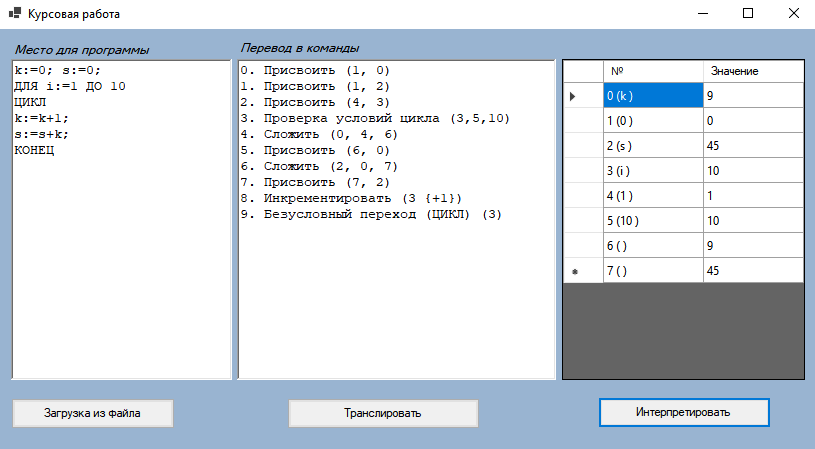






Цикл





**Листинг программы**

using System;

using System.Drawing;

using System.IO;

using System.Windows.Forms;

namespace Lab4

{

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

Stack<Data> DataList = new Stack<Data>();

Stack<Mark> MarkList;

Stack<Commands> CommandList;

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

{

DataList.Clear();

richTextBox2.Clear();

string pro = richTextBox1.Text;

Lexer lexer = new Lexer(pro);

Lexem lexem = lexer.GetLex();

while (lexem != null)

{

if (lexem.GetLexemClass() == "1")

{

if (!IsInStack(DataList, lexem.GetLexem()))

{

Data date = new Data(lexem.GetLexem());

DataList.Push(date);

}

}

if (lexem.GetLexemClass() == "2")

{

if (!IsInStack(DataList, lexem.GetLexem()))

{

Data date = new Data(lexem.GetLexem(), lexem.GetLexem());

DataList.Push(date);

}

}

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

{

MessageBox.Show("Обнаружена лексическая ошибка");

return;

}

lexem = lexer.GetLex();

}

lexer = new Lexer(pro);

SyntaxAnalysis synt = new SyntaxAnalysis();

DataCommandsMarks DAC = synt.Analysis(lexer, DataList);

if (DAC != null)

{

DataList = DAC.Datalist();

MarkList = DAC.Marklist();

CommandList = DAC.Commandlist();

string t;

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

{

int com = CommandList.Ind(i).GetCommandNum();

switch (com)

{

case 1:

t = "Присвоить (";

t = String.Concat(i, ". ", t, CommandList.Ind(i).GetAdressV1(), ", ", CommandList.Ind(i).GetAdressV2(), ")");

richTextBox2.SelectedText = t + "\n";

break;

case 2:

t = "Сложить (";

t = String.Concat(i, ". ", t, CommandList.Ind(i).GetAdressV1(), ", ", CommandList.Ind(i).GetAdressV2(), ", ", CommandList.Ind(i).GetAdressRes(), ")");

richTextBox2.SelectedText = t + "\n";

break;

case 3:

t = "Умножить (";

t = String.Concat(i, ". ", t, CommandList.Ind(i).GetAdressV1(), ", ", CommandList.Ind(i).GetAdressV2(), ", ", CommandList.Ind(i).GetAdressRes(), ")");

richTextBox2.SelectedText = t + "\n";

break;

case 4:

t = "Равняется (";

t = String.Concat(i, ". ", t, CommandList.Ind(i).GetAdressV1(), ", ", CommandList.Ind(i).GetAdressV2(), ", ", CommandList.Ind(i).GetAdressRes(), ")");

richTextBox2.SelectedText = t + "\n";

break;

case 5:

t = "Не равно (";

t = String.Concat(i, ". ", t, CommandList.Ind(i).GetAdressV1(), ", ", CommandList.Ind(i).GetAdressV2(), ", ", CommandList.Ind(i).GetAdressRes(), ")");

richTextBox2.SelectedText = t + "\n";

break;

case 6:

t = "Условный переход (";

t = String.Concat(i, ". ", t, SearchMarker(MarkList, CommandList.Ind(i).GetJumperNum()).GetAdress(), ",", SearchMarker(MarkList, CommandList.Ind(i).GetJumperNum()).GetLabel(), ")");

richTextBox2.SelectedText = t + "\n";

break;

case 7:

t = "Безусловный переход (if) (";

t = String.Concat(i, ". ", t, SearchMarker(MarkList, CommandList.Ind(i).GetJumperNum()).GetAdress(), ",", SearchMarker(MarkList, CommandList.Ind(i).GetJumperNum()).GetLabel(), ")");

richTextBox2.SelectedText = t + "\n";

break;

case 8:

t = "Инкрементировать (";

t = String.Concat(i, ". ", t, CommandList.Ind(i).GetAdressV1(), " {+", CommandList.Ind(i).GetStep(), "})");

richTextBox2.SelectedText = t + "\n";

break;

case 9:

t = "Проверка условий цикла (";

t = String.Concat(i, ". ", t, SearchMarker(MarkList, CommandList.Ind(i).GetJumperNum()).GetAdressi(), ",", SearchMarker(MarkList, CommandList.Ind(i).GetJumperNum()).Getend(), ",", SearchMarker(MarkList, CommandList.Ind(i).GetJumperNum()).GetLabel(), ")");

richTextBox2.SelectedText = t + "\n";

break;

case 10:

t = "Безусловный переход (ЦИКЛ) (";

t = String.Concat(i, ". ", t, SearchMarker(MarkList, CommandList.Ind(i).GetJumperNum()).GetLabel(), ")");

richTextBox2.SelectedText = t + "\n";

break;

}

}

MessageBox.Show("Трансляция прошла успешно");

}

else

{

MessageBox.Show("Обнаружена синтаксическая ошибка");

return;

}

lexer = new Lexer(pro);

lexem = lexer.GetLex();

}

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

{

if (File.Exists("txt1.txt"))

{

string[] readText = File.ReadAllLines("txt1.txt");

for (int i = 0; i < readText.Length - 1; i++)

{

richTextBox1.AppendText(readText[i] + "\n");

}

richTextBox1.AppendText(readText[readText.Length - 1]);

}

else

{

MessageBox.Show("Не удалось открыть файл", "Файл не найден", MessageBoxButtons.OK);

}

}

private bool IsInStack(Stack<Data> DL, string data)

{

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

if (DL.Ind(i).GetName() == data)

return true;

return false;

}

private Mark SearchMarker(Stack<Mark> ML, int num)

{

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

if (ML.Ind(i).GetNum() == num)

return ML.Ind(i);

return default;

}

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

{

Interpretator Interpr = new Interpretator(CommandList, MarkList, DataList);

DataList = Interpr.Calculate();

dataGridView1.RowCount = DataList.Length();

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

{

dataGridView1[0, i].Value = String.Concat(i, " (", DataList.Ind(i).GetName(), " )");

dataGridView1[1, i].Value = String.Concat(DataList.Ind(i).GetValue());

}

MessageBox.Show("Интерпретация прошла успешно");

}

}

}

using System;

namespace Lab4

{

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;

}

}

}

namespace Lab4

{

class Data

{

string value;

string name;

public Data() { }

public Data(string n)

{

name = n;

}

public Data(string n, string v)

{

name = n;

value = v;

}

public void SetValue(string v)

{

value = v;

}

public string GetName()

{

return name;

}

public string GetValue()

{

return value;

}

}

}

namespace Lab4

{

class Mark

{

int AdressResultofEvent;

int GoToLabel;

int JumperNum;

int adressi;

int end;

public Mark(int Num)

{

JumperNum = Num;

}

public Mark(int Num, int AdressresofEvent)

{

AdressResultofEvent = AdressresofEvent;

JumperNum = Num;

}

public void SetForCycle(int i, int end)

{

adressi = i;

this.end = end;

}

public int GetAdressi()

{

return adressi;

}

public int Getend()

{

return end;

}

public void SetLabel(int label)

{

GoToLabel = label;

}

public int GetLabel()

{

return GoToLabel;

}

public int GetNum()

{

return JumperNum;

}

public int GetAdress()

{

return AdressResultofEvent;

}

}

}

namespace Lab4

{

class Commands

{

int adressV1;

int adressV2;

int adressRes;

int commandNum;

int JumperNum;

int step;

public Commands(string command, int num)

{

JumperNum = num;

switch (command)

{

case "{ThenJumper}":

commandNum = 6;

break;

case "{ElseJumper}":

commandNum = 7;

break;

case "{EndJumper}":

commandNum = 9;

break;

case "{BeginJumper}":

commandNum = 10;

break;

}

}

public Commands(string command, int v1, int V2\_or\_step)

{

adressV1 = v1;

if (command == "{Assign}")

{

adressV2 = V2\_or\_step;

commandNum = 1;

}

if (command == "{Incr}")

{

step = V2\_or\_step;

commandNum = 8;

}

}

public Commands(string command, int v2, int v1, int res)

{

adressV1 = v1;

adressV2 = v2;

adressRes = res;

switch (command)

{

case "{Plus}":

commandNum = 2;

break;

case "{Multyply}":

commandNum = 3;

break;

case "{Equals}":

commandNum = 4;

break;

case "{NotEquals}":

commandNum = 5;

break;

}

}

public int GetCommandNum()

{

return commandNum;

}

public int GetStep()

{

return step;

}

public int GetAdressV1()

{

return adressV1;

}

public int GetAdressV2()

{

return adressV2;

}

public int GetAdressRes()

{

return adressRes;

}

public int GetJumperNum()

{

return JumperNum;

}

}

}

using System;

namespace Lab4

{

class SyntaxAnalysis

{

Stack<string> stack = new Stack<string>();

Stack<int> reserve = new Stack<int>();

bool InCycle = false;

Stack<int> ListofAdresses = new Stack<int>();

Stack<int> Steps = new Stack<int>();

int stepflag = 0;

public DataCommandsMarks Analysis(Lexer lexer, Stack<Data> DataList)

{

DataCommandsMarks DCM;

Stack<Commands> CommandList = new Stack<Commands>();

Stack<Mark> MarkList = new Stack<Mark>();

Stack<Mark> ReserveThenList = new Stack<Mark>();

Stack<Mark> ReserveElseList = new Stack<Mark>();

Stack<int> ListofJumperNums = new Stack<int>();

Stack<Mark> ReserveCheckList = new Stack<Mark>();

Stack<int> boolEvents = new Stack<int>();

Commands command;

Data date;

Mark mark;

int NumofJumper = 0;

stack.Push("Program");

int action;

string l\_class;

string state;

Lexem lexem = lexer.GetLex();

if (lexem == null)

{

DCM = new DataCommandsMarks(DataList, CommandList, MarkList);

return DCM;

}

for (; ; )

{

if (stack.IsEmpty())

return null;

state = stack.Top();

l\_class = lexem.GetLexemClass();

switch (state)

{

case "S":

action = S(l\_class);

break;

case "Program":

action = Program(l\_class);

break;

case "Id":

action = Id(lexem, DataList);

break;

case "E":

action = E(l\_class);

break;

case "EList":

action = EList(l\_class);

break;

case "T":

action = T(l\_class);

break;

case "TList":

action = TList(l\_class);

break;

case "F":

action = F(lexem, DataList);

break;

case "SList":

action = SList(l\_class);

break;

case "Log":

action = Log(l\_class);

break;

case "LogOp":

action = LList(l\_class);

break;

case "Step":

InCycle = false;

action = Step(l\_class);

break;

case "{Assign}":

command = new Commands("{Assign}", reserve.Pop(), reserve.Top());

CommandList.Push(command);

stack.Pop();

action = 2;

if (InCycle)

ListofAdresses.Push(reserve.Pop());

break;

case "{Plus}":

date = new Data();

command = new Commands("{Plus}", reserve.Pop(), reserve.Pop(), DataList.Length());

CommandList.Push(command);

reserve.Push(DataList.Length());

DataList.Push(date);

stack.Pop();

action = 1;

break;

case "{Multyply}":

date = new Data();

command = new Commands("{Multyply}", reserve.Pop(), reserve.Pop(), DataList.Length());

CommandList.Push(command);

reserve.Push(DataList.Length());

DataList.Push(date);

stack.Pop();

action = 1;

break;

case "{Equals}":

date = new Data();

command = new Commands("{Equals}", reserve.Pop(), reserve.Pop(), DataList.Length());

CommandList.Push(command);

boolEvents.Push(DataList.Length());

DataList.Push(date);

stack.Pop();

action = 1;

break;

case "{NotEquals}":

date = new Data();

command = new Commands("{NotEquals}", reserve.Pop(), reserve.Pop(), DataList.Length());

CommandList.Push(command);

boolEvents.Push(DataList.Length());

DataList.Push(date);

stack.Pop();

action = 1;

break;

case "{ThenJumper}":

command = new Commands("{ThenJumper}", NumofJumper);

CommandList.Push(command);

mark = new Mark(NumofJumper, boolEvents.Top());

ReserveThenList.Push(mark);

stack.Pop();

action = 1;

NumofJumper += 1;

break;

case "{ElseJumper}":

command = new Commands("{ElseJumper}", NumofJumper);

CommandList.Push(command);

mark = new Mark(NumofJumper, boolEvents.Pop());

ReserveElseList.Push(mark);

stack.Pop();

action = 1;

NumofJumper += 1;

break;

case "{ThenLabel}":

mark = ReserveThenList.Pop();

mark.SetLabel(CommandList.Length());

MarkList.Push(mark);

stack.Pop();

action = 1;

break;

case "{ElseLabel}":

mark = ReserveElseList.Pop();

mark.SetLabel(CommandList.Length());

MarkList.Push(mark);

stack.Pop();

action = 1;

break;

case "{BeginLabel}":

mark = new Mark(NumofJumper);

mark.SetLabel(CommandList.Length());

mark.SetForCycle(ListofAdresses.Ind(ListofAdresses.Length() - 2), ListofAdresses.Top());

MarkList.Push(mark);

ListofJumperNums.Push(NumofJumper);

NumofJumper += 1;

stack.Pop();

action = 1;

break;

case "{EndJumper}":

command = new Commands("{EndJumper}", NumofJumper);

CommandList.Push(command);

mark = new Mark(NumofJumper);

mark.SetForCycle(ListofAdresses.Ind(ListofAdresses.Length() - 2), ListofAdresses.Top());

ReserveCheckList.Push(mark);

stack.Pop();

NumofJumper += 1;

action = 1;

ListofAdresses.Push(reserve.Pop());

InCycle = false;

break;

case "{BeginJumper}":

command = new Commands("{BeginJumper}", ListofJumperNums.Pop());

CommandList.Push(command);

stack.Pop();

action = 1;

break;

case "{EndLabel}":

mark = ReserveCheckList.Pop();

mark.SetLabel(CommandList.Length());

MarkList.Push(mark);

stack.Pop();

action = 1;

break;

case "{Incr}":

ListofAdresses.Pop();

command = new Commands("{Incr}", ListofAdresses.Ind(ListofAdresses.Length() - 2), Steps.Pop());

ListofAdresses.Pop();

CommandList.Push(command);

stack.Pop();

action = 1;

NumofJumper += 1;

break;

default:

if (state == lexem.GetLexem())

{

stack.Pop();

action = 2;

if (lexem.GetLexemClass() == "9")

stepflag = 0;

}

else

action = -1;

break;

}

if (action == -1)

return null;

if (action == 2)

{

lexem = lexer.GetLex();

if (lexem == null)

if (stack.Top() == "Program" && stack.Length() == 1)

{

DCM = new DataCommandsMarks(DataList, CommandList, MarkList);

return DCM;

}

else return null;

}

}

}

private int S(string lclass)

{

switch (lclass)

{

//#1

case "1":

stack.Pop();

stack.Push("Program");

stack.Push("{Assign}");

stack.Push("E");

stack.Push(":=");

stack.Push("Id");

return 1;

//#2

case "3":

stack.Pop();

stack.Push("Program");

stack.Push("endif");

stack.Push("SList");

stack.Push("S");

stack.Push("then");

stack.Push("{ThenJumper}");

stack.Push(")");

stack.Push("Log");

stack.Push("(");

return 2;

//#3

case "7":

stack.Pop();

stack.Push("Program");

stack.Push("КОНЕЦ");

stack.Push("{EndLabel}");

stack.Push("{BeginJumper}");

stack.Push("{Incr}");

stack.Push("S");

stack.Push("ЦИКЛ");

stack.Push("{EndJumper}");

stack.Push("{BeginLabel}");

stack.Push("Step");

stack.Push("F");

stack.Push("{Assign}");

stack.Push("F");

stack.Push(":=");

stack.Push("Id");

InCycle = true;

return 2;

default:

return -1;

}

}

private int Step(string lclass)

{

//#6

if (lclass == "9")

{

stack.Pop();

stack.Push("F");

stepflag = 1;

return 2;

}

if (lclass == "10")

{

stack.Pop();

stepflag = 2;

return 1;

}

return -1;

}

private int Program(string lclass)

{

//#4

if (lclass == "1" || lclass == "3" || lclass == "7")

{

stack.Pop();

stack.Push("S");

return 1;

}

//#5

if (lclass == "5" || lclass == "6" || lclass == "11")

{

stack.Pop();

return 1;

}

return -1;

}

private int Id(Lexem lexem, Stack<Data> DataList)

{

//#6

if (lexem.GetLexemClass() == "1")

{

reserve.Push(Search(DataList, lexem.GetLexem()));

stack.Pop();

return 2;

}

return -1;

}

private int E(string lclass)

{

//#7

if (lclass == "1" || lclass == "2" || lclass == "18")

{

stack.Pop();

stack.Push("EList");

stack.Push("T");

return 1;

}

return -1;

}

private int EList(string lclass)

{

//#8

if (lclass == "16")

{

stack.Pop();

stack.Push("EList");

stack.Push("{Plus}");

stack.Push("T");

return 2;

}

//#5

if ( lclass == "19")

{

stack.Pop();

return 1;

}

//#6

if (lclass == "15")

{

stack.Pop();

return 1;

}

return -1;

}

private int T(string lclass)

{

//#9

if (lclass == "1" || lclass == "2" || lclass == "18")

{

stack.Pop();

stack.Push("TList");

stack.Push("F");

return 1;

}

return -1;

}

private int TList(string lclass)

{

//#10

if (lclass == "17")

{

stack.Pop();

stack.Push("TList");

stack.Push("{Multyply}");

stack.Push("F");

return 2;

}

//#5

if (lclass == "16" || lclass == "15" || lclass == "19")

{

stack.Pop();

return 1;

}

return -1;

}

private int F(Lexem lexem, Stack<Data> DataList)

{

//#6

if (lexem.GetLexemClass() == "1" || lexem.GetLexemClass() == "2")

{

reserve.Push(Search(DataList, lexem.GetLexem()));

if (InCycle)

ListofAdresses.Push(Search(DataList, lexem.GetLexem()));

if (stepflag == 1)

{

Steps.Push(Convert.ToInt32(lexem.GetLexem()));

stepflag = 0;

}

if (stepflag == 2)

Steps.Push(1);

stack.Pop();

return 2;

}

return -1;

}

private int SList(string lclass)

{

if (lclass == "5")

{

stack.Pop();

stack.Push("{ElseLabel}");

stack.Push("Program");

stack.Push("S");

stack.Push("{ThenLabel}");

stack.Push("{ElseJumper}");

return 2;

}

if (lclass == "6")

{

stack.Pop();

stack.Push("{ThenLabel}");

return 1;

}

return -1;

}

private int Log(string lclass)

{

//#12

if (lclass == "1" || lclass == "2")

{

stack.Pop();

stack.Push("LogOp");

stack.Push("F");

return 1;

}

return -1;

}

private int LList(string lclass)

{

//#6

if (lclass == "14")

{

stack.Pop();

stack.Push("{Equals}");

stack.Push("F");

return 2;

}

if (lclass == "13")

{

stack.Pop();

stack.Push("{NotEquals}");

stack.Push("F");

return 2;

}

return -1;

}

private int Search(Stack<Data> st, string id)

{

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

if (st.Ind(i).GetName() == id)

return i;

return default;

}

}

class DataCommandsMarks

{

Stack<Data> DataList;

Stack<Commands> CommandList;

Stack<Mark> MarkList;

public DataCommandsMarks(Stack<Data> DL, Stack<Commands> CL, Stack<Mark> ML)

{

DataList = DL;

CommandList = CL;

MarkList = ML;

}

public Stack<Data> Datalist()

{

return DataList;

}

public Stack<Commands> Commandlist()

{

return CommandList;

}

public Stack<Mark> Marklist()

{

return MarkList;

}

}

}

using System;

namespace Lab4

{

class Interpretator

{

Stack<Commands> CommandList;

Stack<Mark> MarkList;

Stack<Data> DataList;

public Interpretator(Stack<Commands> CL, Stack<Mark> ML, Stack<Data> DL)

{

CommandList = CL;

MarkList = ML;

DataList = DL;

}

public Stack<Data> Calculate()

{

int v1;

int v2;

int i = 0;

while (i < CommandList.Length())

{

Commands command = CommandList.Ind(i);

v1 = Convert.ToInt32(DataList.Ind(command.GetAdressV1()).GetValue());

v2 = Convert.ToInt32(DataList.Ind(command.GetAdressV2()).GetValue());

switch (command.GetCommandNum())

{

case 1:

DataList.Ind(command.GetAdressV2()).SetValue(DataList.Ind(command.GetAdressV1()).GetValue());

break;

case 2:

DataList.Ind(command.GetAdressRes()).SetValue(Convert.ToString(v1 + v2));

break;

case 3:

DataList.Ind(command.GetAdressRes()).SetValue(Convert.ToString(v1\*v2));

break;

case 4:

DataList.Ind(command.GetAdressRes()).SetValue(Convert.ToString(v1 == v2));

break;

case 5:

DataList.Ind(command.GetAdressRes()).SetValue(Convert.ToString(v1 != v2));

break;

case 6:

if (!Convert.ToBoolean(DataList.Ind(SearchMark(command.GetJumperNum()).GetAdress()).GetValue()))

i = SearchMark(command.GetJumperNum()).GetLabel() - 1;

break;

case 7:

if (Convert.ToBoolean(DataList.Ind(SearchMark(command.GetJumperNum()).GetAdress()).GetValue()))

i = SearchMark(command.GetJumperNum()).GetLabel() - 1;

break;

case 8:

int k = Convert.ToInt32(DataList.Ind(command.GetAdressV1()).GetValue());

int step = command.GetStep();

DataList.Ind(command.GetAdressV1()).SetValue(Convert.ToString(k + step));

break;

case 9:

v1 = Convert.ToInt32(DataList.Ind(SearchMark(command.GetJumperNum()).GetAdressi()).GetValue());

v2 = Convert.ToInt32(DataList.Ind(SearchMark(command.GetJumperNum()).Getend()).GetValue());

if (v1 >= v2)

i = SearchMark(command.GetJumperNum()).GetLabel() - 1;

break;

case 10:

i = SearchMark(command.GetJumperNum()).GetLabel() - 1;

break;

}

i++;

}

return DataList;

}

private Mark SearchMark(int JumperNum)

{

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

{

if (MarkList.Ind(i).GetNum() == JumperNum)

return MarkList.Ind(i);

}

return null;

}

}

}