Минобрнауки России

ФГБОУ ВПО «НИУ МЭИ» АВТИ

Кафедра математического и компьютерного моделирования

**Лабораторная работа №3**

**Математическое обеспечение ЭВМ**

**«Построение синтаксического анализатора»**

**Работу выполнил:**

Солонин Егор А-14-19

Вариант 14

**Работу принял:**

Князев А. В.

Москва 2021

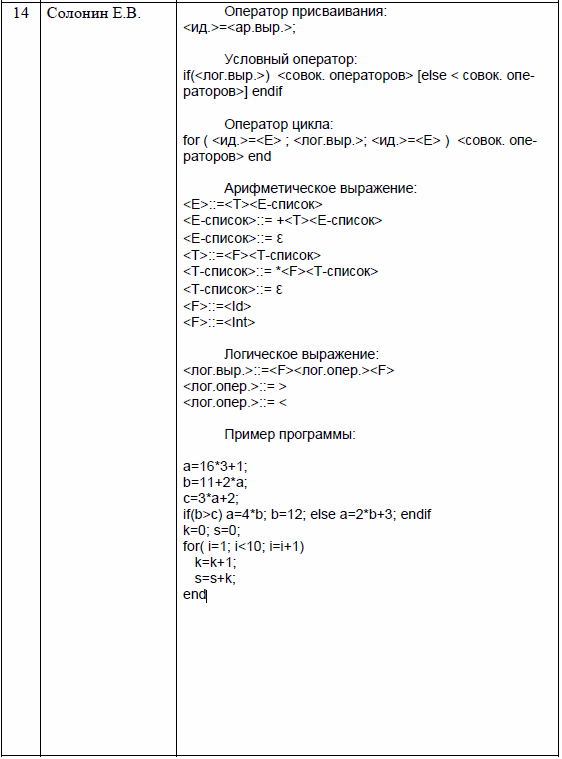
**Задание:**

1. Преобразовать заданную грамматику в LL(1)-грамматику.

2. Разработать МП-автомат для нисходящего грамматического разбора предложений данного языка.

3. Разработать функцию, реализующую МП-автомат.

4. Разработать программу, иллюстрирующую работу созданного автомата для данного языка. Программа разрабатывается как приложение с графическим интерфейсом на языке C# в среде Visual Studio. Не должны использоваться коллекции. Не должны использоваться регулярные выражения и другие средства разбора строк.

*Индивидуальный вариант (№14):*

**LL(1)-грамматика:**

1. <программа>::= <оператор><спис. операторов>
2. <спис. операторов>::= <оператор><спис. операторов>
3. <программа>::=
4. <оператор>::= <идент.>=<E>;
5. <оператор>::=for(<идент.>=<E>; <лог.выр.>; <идент.>=<E>) <спис. операторов> end
6. <оператор>::= if(<лог.выр.>) <спис. операторов> <иначе> endif
7. <иначе>::= else <спис. операторов>
8. <иначе>::=
9. <лог.выр.>::= <F><лог.опер.><F>
10. <лог.опер.>::= >
11. <лог.опер.>::= <
12. <E>::= <T><E-список>
13. <E-список>::= +<T><E-список>
14. <E-список>::=
15. <T>::= <F><T-список>
16. <T-список>::=  \*<F><T-список>
17. <T-список>::=
18. <F>::= <идент.>
19. <F>::= <целое>

**Множества выбора:**

Выбор(1) = {<идент.>, if, for}

Выбор(2) = {<идент.>, if, for}

Выбор(3)= {“-|”}

Выбор(4)= {<идент.>}

Выбор(5)= { for  }

Выбор(6)= {if}

Выбор(7)= {else}

Выбор(8)= {“-|”}

Выбор(9)= {<идент.>, <целое>}

Выбор(10)= {>}

Выбор(11)= {<}

Выбор(12)= {<идент.>, <целое>}

Выбор(13)= {+}

Выбор(14)= {;}

Выбор(15)= {<идент.>, <целое>}

Выбор(16)= {\*}

Выбор(17)= {+,;}

Выбор(18)= {<идент.>}

Выбор(19)= {<целое>}

**Управляющая таблица МП-автомата:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Магаз. символы | <идент> | <целое> | for | if | else | endif | end | + | \* | ( | ) | > | < | = | ; | -| |
| <программа> | 1 | O | 1 | 1 | O | O | O | O | O | O | O | O | O | O | O | O |
| <спис. операторов> | 2 | O | 2 | 2 | 3 | 3 | 3 | O | O | O | O | O | O | O | O | 3 |
| <оператор> | 4 | O | 5 | 6 | O | O | O | O | O | O | O | O | O | O | O | O |
| <else> | 8 | O | 8 | 8 | 7 | 8 | 8 | O | O | O | O | O | O | O | O | 8 |
| <лог.выр.> | 9 | 9 | O | O | O | O | O | O | O | O | O | O | O | O | O | O |
| <лог. опер.> | O | O | O | O | O | O | O | O | O | O | O | 10 | 11 | O | O | O |
| <E> | 12 | 12 | O | O | O | O | O | O | O | O | O | O | O | O | O | O |
| <E список> | O | O | O | O | O | O | O | 13 | O | O | O | O | O | O | 14 | O |
| <T> | 15 | 15 | O | O | O | O | O | O | O | O | O | O | O | O | O | O |
| <T список> | O | O | O | O | O | O | O | 16 | 16 | O | O | O | O | O | 17 | O |
| <F> | 18 | 19 | O | O | O | O | O | O | O | O | O | O | O | O | O | O |
| <идент.> | # | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O |
| <endif> | O | O | O | O | O | # | O | O | O | O | O | O | O | O | O | O |
| <end> | O | O | O | O | O | O | # | O | O | O | O | O | O | O | O | O |
| = | O | O | O | O | O | O | O | O | O | O | O | O | O | # | O | O |
| ( | O | O | O | O | O | O | O | O | O | # | O | O | O | O | O | O |
| ) | O | O | O | O | O | O | O | O | O | O | # | O | O | O | O | O |
| ; | O | O | O | O | O | O | O | O | O | O | O | O | O | O | # | O |
| if | O | O | O | O | # | O | O | O | O | O | O | O | O | O | O | O |
| for | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O | O |
| ˅ | О | О | О | О | О | O | O | О | О | О | О | О | О | O | O | Д |

Начальное содержимое магазина: <программа>˅

1. Заменить(<оператор><спис. операторов>), Держать
2. Заменить(<оператор><спис. операторов>), Держать
3. Вытолкнуть, Держать
4. Заменить(=<E>;), Сдвиг
5. Заменить(for( <идент.>=<E>; <лог.выр.>; <идент.>=<E>) <спис. операторов> end), Сдвиг
6. Заменить((<лог. выр.>) <спис. операторов><else>), Сдвиг
7. Заменить(<спис. операторов>), Сдвиг
8. Вытолкнуть, Держать
9. Заменить(<F><лог.опер.><F>), Держать
10. Вытолкнуть, Сдвиг
11. Вытолкнуть, Сдвиг
12. Заменить(<T><E-список>), Держать
13. Заменить(<T><E-список>), Сдвиг
14. Вытолкнуть, Держать
15. Заменить(<F><T-список>), Держать
16. Заменить(<F><T-список>), Сдвиг
17. Вытолкнуть, Держать
18. Вытолкнуть, Сдвиг
19. Вытолкнуть, Сдвиг

# - Вытолкнуть, Сдвиг

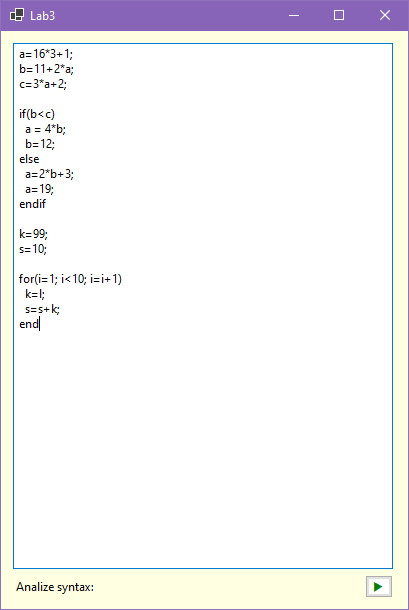
Д – Допустить

О – Отвергнуть

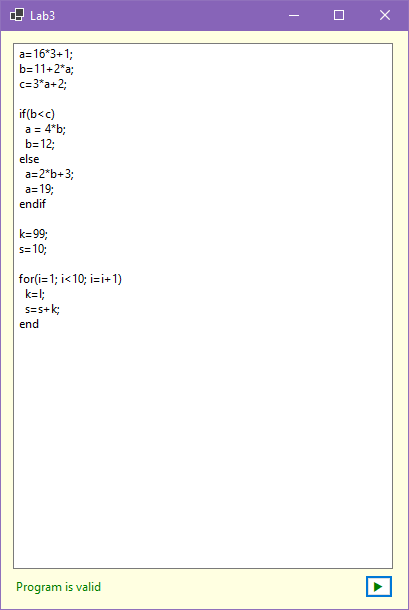
**Алгоритм работы синтаксического анализатора:**

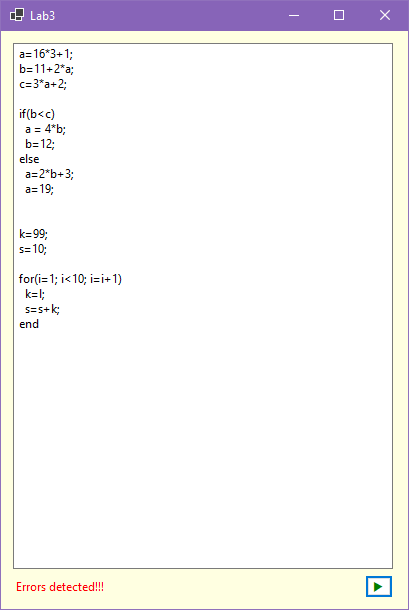
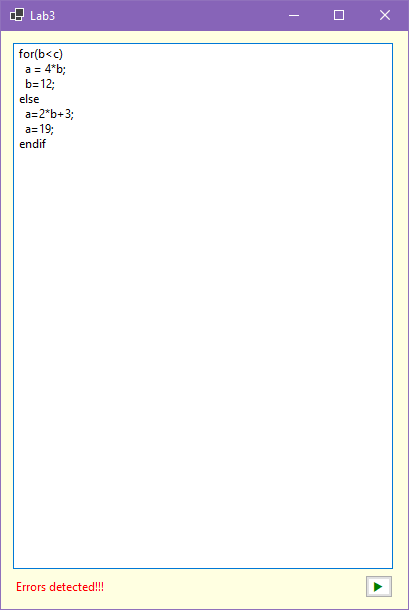
1. Заводим автомат с магазинной памятью *PDA* на основе стека
2. Изначально добавляем <.> (признак окончания программы) и <program> в стек
3. Пока стек не пуст и пока не получен отрицательный результат работы анализатора:
   1. Получить очередной тип лексемы с помощью функции из класса лексического блока (лабораторная работа 2)
   2. Получить значение с вершины стека
   3. Обработать каждый возможный случай на основе управляющей таблицы

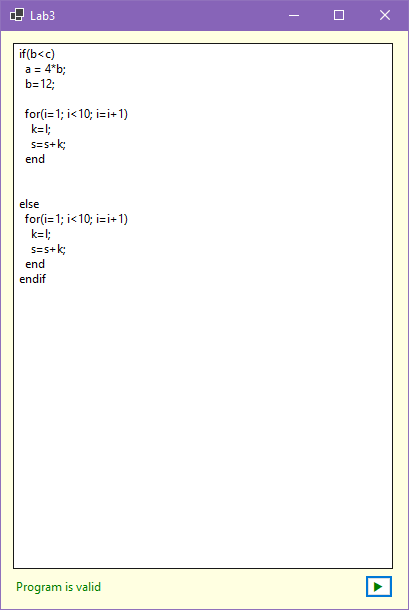
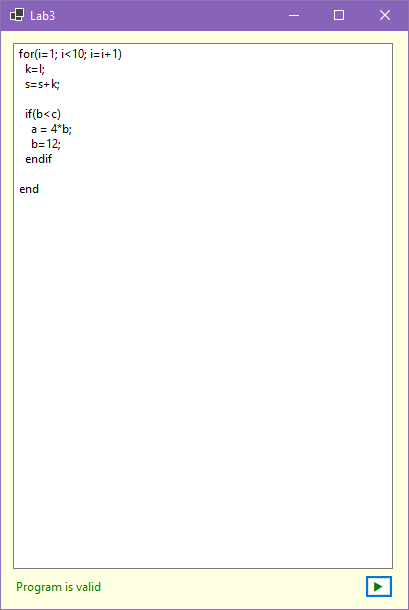
**Описание интерфейса программы:**

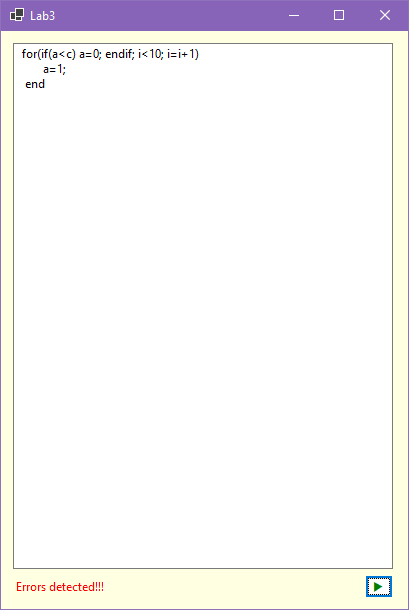
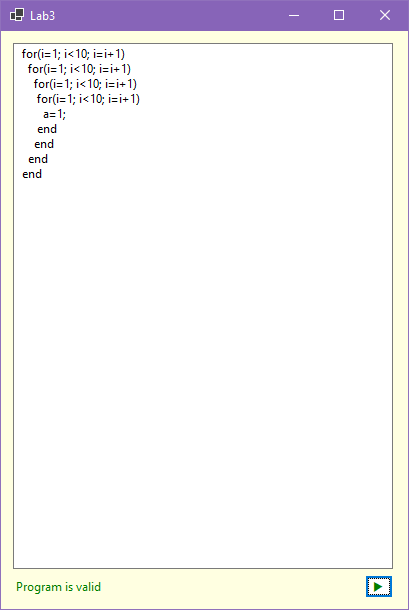
В текстовое поле заносится код. После нажатия на кнопку пользователю сообщается о корректности синтаксиса в представленном коде

**Тесты работы программы:**

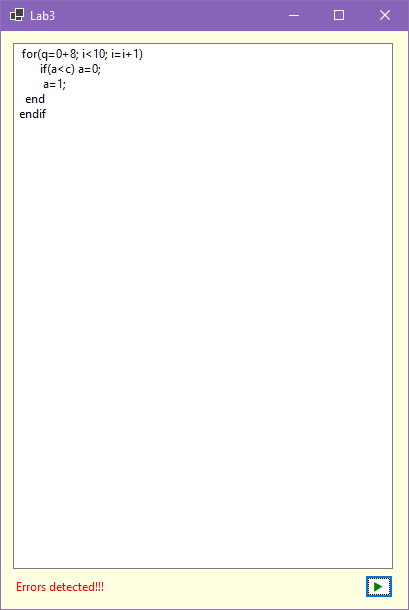
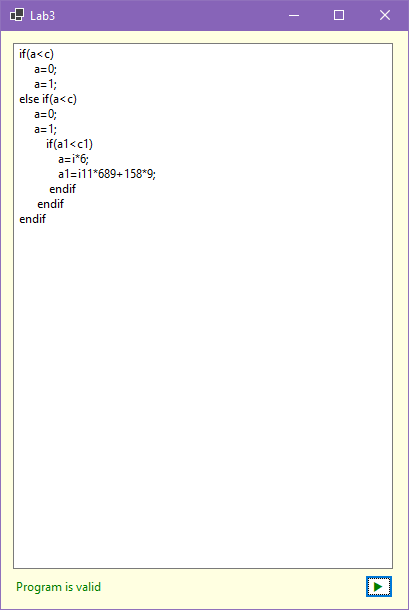
*Тест 1 (корректный код)*

*Тест 2 (ошибка – отсутствует endif) Тест 3 (ошибка – for вместо if)*

*Тест 5 (проверка - for внутри if/else) Тест 6 (if внутри for)*

*Тест 7 (вложенные циклы)* *Тест 8 (блок if внутри шапки цикла for)*

*Тест 9 (ошибка – отсутствует endif) Тест 10 (вложенные if)*



**Листинг программы:**

public partial class Form1 : Form

{

public Form1()

{

InitializeComponent();

}

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

{

resultMsg.Text = "";

Thread.Sleep(500);

bool result = SyntaxAnalyzer.programIsValid(data.Text);

resultMsg.Text = result ? "Program is valid" : "Errors detected!!!";

resultMsg.ForeColor = result ? Color.Green : Color.Red;

}

}

class SyntaxAnalyzer

{

public static bool programIsValid(string data)

{

bool result = true;

bool keep = false;

bool bra\_opened = false;

bool inside\_for = false;

LexBlock.index = 0;

string LexemType = "";

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

PDA.Push("<.>");

PDA.Push("<program>");

while (result && !PDA.IsEmpty())

{

if (!keep)

{

string Lexem = LexBlock.GetLexem(data);

LexemType = LexBlock.GetLexemType(Lexem).ToString();

if (LexemType == "INVALID")

{

result = false;

continue;

}

}

switch (PDA.Peek())

{

case "<program>":

if ((LexemType == "ID") || (LexemType == "KW\_IF") || (LexemType == "KW\_FOR"))

{

PDA.Pop();

PDA.Push("<operators list>");

PDA.Push("<operator>");

}

else result = false;

keep = true;

break;

case ("<operators list>"):

if ((LexemType == "ID") || (LexemType == "KW\_IF") || (LexemType == "KW\_FOR"))

{

PDA.Pop();

PDA.Push("<operators list>");

PDA.Push("<operator>");

}

else if ((LexemType == "KW\_ENDIF") ||

(LexemType == "END") ||

(LexemType == "KW\_FOREND") ||

(LexemType == "KW\_ELSE"))

{

PDA.Pop();

}

else result = false;

keep = true;

break;

case ("<operator>"):

if ((LexemType == "ID"))

{

PDA.Pop();

PDA.Push("<;>");

PDA.Push("<E>");

PDA.Push("<=>");

}

else if ((LexemType == "KW\_FOR"))

{

PDA.Pop();

PDA.Push("<end>");

PDA.Push("<operators list>");

PDA.Push("<)>");

PDA.Push("<E>");

PDA.Push("<=>");

PDA.Push("<ID>");

PDA.Push("<;>");

PDA.Push("<logical expression>");

PDA.Push("<;>");

PDA.Push("<E>");

PDA.Push("<=>");

PDA.Push("<ID>");

PDA.Push("<(>");

inside\_for = true;

}

else if ((LexemType == "KW\_IF"))

{

PDA.Pop();

PDA.Push("<endif>");

PDA.Push("<else block>");

PDA.Push("<operators list>");

PDA.Push("<)>");

PDA.Push("<logical expression>");

PDA.Push("<(>");

}

else result = false;

keep = false;

break;

case ("<else block>"):

if (LexemType == "KW\_ELSE")

{

PDA.Pop();

PDA.Push("<operators list>");

keep = false;

}

else if ((LexemType == "ID") ||

(LexemType == "KW\_IF") ||

(LexemType == "KW\_FOR") ||

(LexemType == "KW\_ENDIF") ||

(LexemType == "END") ||

(LexemType == "KW\_FOREND"))

{

PDA.Pop();

keep = true;

}

else result = false;

break;

case ("<logical expression>"):

if ((LexemType == "ID") || (LexemType == "INT"))

{

PDA.Pop();

PDA.Push("<F>");

PDA.Push("<logical operator>");

PDA.Push("<F>");

keep = true;

}

else result = false;

break;

case ("<logical operator>"):

if ((LexemType == "OP\_LESS") || (LexemType == "OP\_MORE"))

PDA.Pop();

else result = false;

keep = false;

break;

case ("<E>"):

if ((LexemType == "ID") || (LexemType == "INT"))

{

PDA.Pop();

PDA.Push("<E list>");

PDA.Push("<T>");

}

else result = false;

keep = true;

break;

case ("<E list>"):

if ((LexemType == "OP\_PLS"))

{

PDA.Pop();

PDA.Push("<E list>");

PDA.Push("<T>");

keep = false;

}

else if((LexemType == "DELIM\_SEMI") || (LexemType == "BRA\_CLS" && bra\_opened && inside\_for))

{

PDA.Pop();

keep = true;

}

else result = false;

break;

case ("<T>"):

if ((LexemType == "ID") || (LexemType == "INT"))

{

PDA.Pop();

PDA.Push("<T list>");

PDA.Push("<F>");

}

else result = false;

keep = true;

break;

case ("<T list>"):

if ((LexemType == "OP\_PLS") || (LexemType == "OP\_MULT"))

{

PDA.Pop();

PDA.Push("<T list>");

PDA.Push("<F>");

keep = false;

}

else if ((LexemType == "DELIM\_SEMI") || (LexemType == "BRA\_CLS" && bra\_opened && inside\_for))

{

PDA.Pop();

keep = true;

}

else result = false;

break;

case ("<F>"):

if ((LexemType == "ID") || (LexemType == "INT"))

PDA.Pop();

else result = false;

keep = false;

break;

case ("<=>"):

Pop\_Shift(ref LexemType, "OP\_SET", ref PDA, ref result, ref keep);

break;

case ("<;>"):

Pop\_Shift(ref LexemType, "DELIM\_SEMI", ref PDA, ref result, ref keep);

break;

case ("<(>"):

bra\_opened = true;

Pop\_Shift(ref LexemType, "BRA\_OPN", ref PDA, ref result, ref keep);

break;

case ("<)>"):

bra\_opened = false;

Pop\_Shift(ref LexemType, "BRA\_CLS", ref PDA, ref result, ref keep);

break;

case ("<end>"):

inside\_for = false;

Pop\_Shift(ref LexemType, "KW\_FOREND", ref PDA, ref result, ref keep);

break;

case ("<endif>"):

Pop\_Shift(ref LexemType, "KW\_ENDIF", ref PDA, ref result, ref keep);

break;

case ("<.>"):

Pop\_Shift(ref LexemType, "END", ref PDA, ref result, ref keep);

break;

case ("<ID>"):

Pop\_Shift(ref LexemType, "ID", ref PDA, ref result, ref keep);

break;

default:

result = false;

break;

}

}

return result;

}

private static void Pop\_Shift(ref string LexemType, string GotLexType, ref Stack<string> PDA, ref bool result, ref bool keep)

{

if ((LexemType == GotLexType))

PDA.Pop();

else result = false;

keep = false;

}

}

class Node<T>

{

public Node(T data)

{

Data = data;

}

public T Data { get; set; }

public Node<T> Next { get; set; }

}

class Stack<T>

{

Node<T> head;

int count;

private InvalidOperationException err411 = new InvalidOperationException("Error 411: Stack is empty");

public bool IsEmpty() => count == 0;

public int Count() => count;

public void Push(T item)

{

Node<T> node = new Node<T>(item);

node.Next = head;

head = node;

count++;

}

public T Pop()

{

if (IsEmpty())

throw err411;

Node<T> temp = head;

head = head.Next;

count--;

return temp.Data;

}

public T Peek() => IsEmpty() ? throw err411 : head.Data;

public Stack<T> Reverse()

{

Stack<T> copy = new Stack<T>();

Node<T> node = head;

for (; node != null; node = node.Next)

copy.Push(node.Data);

return copy;

}

public T[] ToArray()

{

T[] arr = new T[count];

int i = 0;

Node<T> node = head;

for(; node != null; node = node.Next, i++)

arr[i] = node.Data;

return arr;

}

}