МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ**

**Н. Г. ЧЕРНЫШЕВСКОГО»**

Факультет компьютерных наук и информационных технологий

**Лабораторная работа 1**

Выполнил:

студент 441 группы

Орлов Даниил Андреевич

2022

Вариант: 14

Номер конструкции – 6

Уровень сложности – 2

Чувствительность к регистру – Нет

Задание: Составить автоматную грамматику и на ее основе реализовать лексический анализатор языка, цепочки которого имеют вид, указанный в задании. Лексический анализатор должен преобразовывать исходный текст в последовательность лексем. По результатам работы анализатора должны формироваться таблицы идентификаторов и констант

Представление базовых элементов языка в виде графа

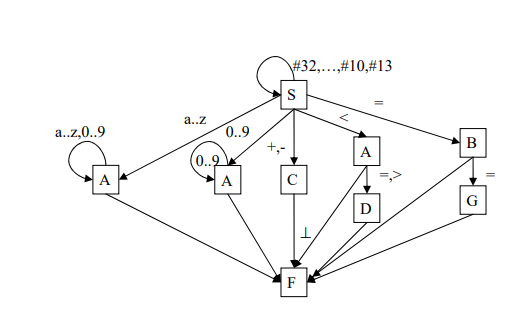
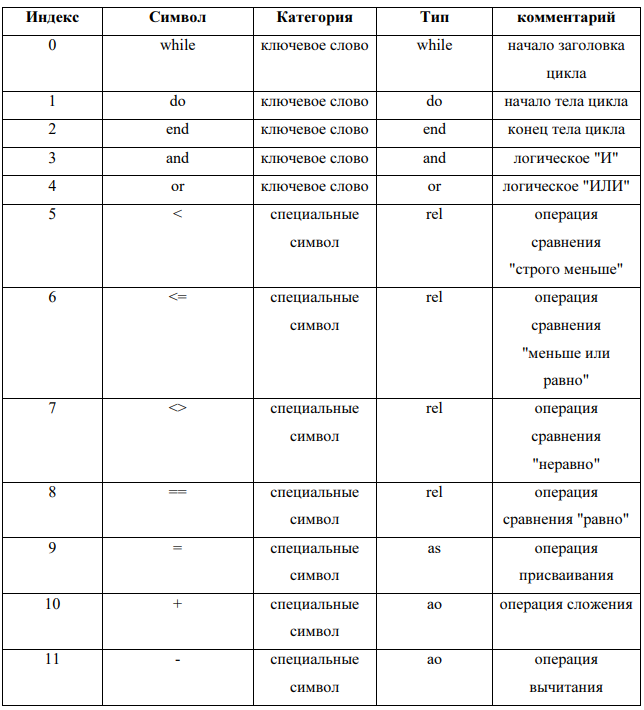


Таблица терминальных символов



4. Описание основных алгоритмов и структур данных, используемых в программе:

а) описание структур данных, используемых для представления

последовательности лексем, таблиц идентификаторов и констант;

В программе используется механизм enum, который описывает LexemeType :

public enum LexemeType

{

If,

Then,

Else,

End,

And,

Or,

ArithmeticOperation,

Relation,

Assigment,

Undefined

}

Здесь определены все нужные для работы виды Лексем

Более общее LexemeClass – делит лексемы на разные типы

public enum LexemeClass

{

Keyword,

Identifier,

Constant,

SpecialSymbols,

Undefined

}

б) описание процедур и функций, отвечающих за работу с введенными

структурами данных;

Метод StartFindLexems реализует логику работы алгоритма. Посимвольно проверяет входное слово(код), при этом храня предыдущее состояние, в зависимости от которого, та или иная констуркция будет присвоена своему типу лексемы языка. Метод также сохраняет всю полученную информацию в список класса Lexeme.

public class Lexeme

{

public LexemeType Type { get; set; }

public LexemeClass Class { get; set; }

public string Value { get; set; }

public override string ToString()

{

return $"Type: {Type}, Class: {Class}, Value: {Value}";

}

}

в) описание алгоритма анализа автоматного языка.

Метод получает на вход текст, который требуется проанализировать. В последствии при помощи операторов ветвления данный текст посимвольно анализируется, при этом помня состояние в котором он находится. Это позволяет получить список элементов базового языка. Более простые лексемы определяются сразу, а более сложные(константы, ключевые слова, ==, >= и др.) обрабатывает метод AddDiffLexeme, который ищет совпадение с уже имеющимся списком всех возможных допустимых вариантов.

Описание интерфейса пользователя программы.

Программа получает на вход текст, который будет проанализирован, и выводит список полученных лексем на консоль. Если анализ прошёл успешно будет сказано True, иначе False.

Контрольный пример и результаты тестирования.

Пример 1:

if a >= b and a < 0

then

if b <= a or b >= 10

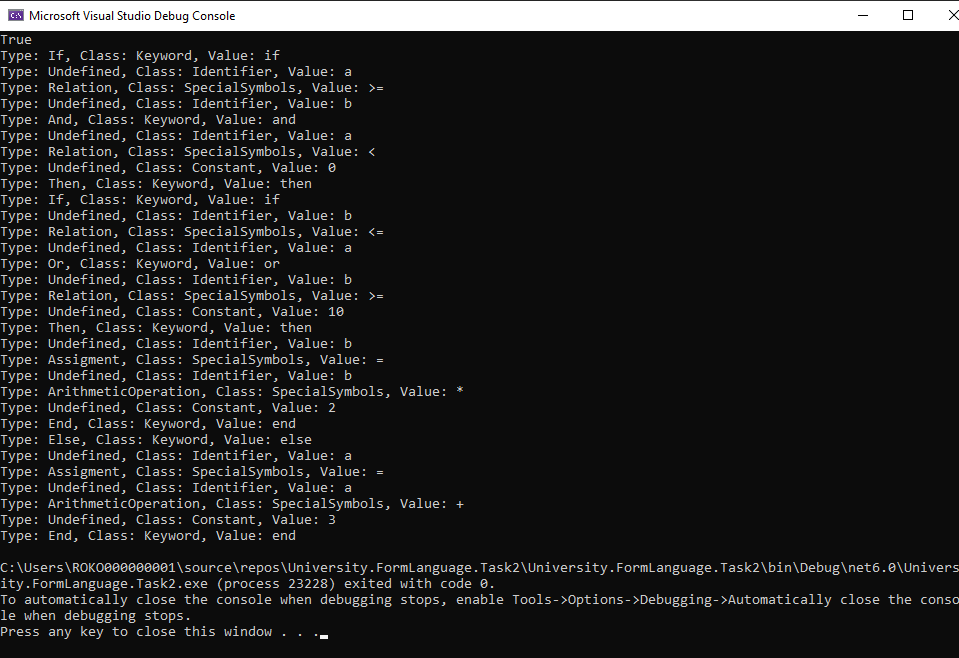
then b = b \* 2

end

else

a = a + 3

end

Результат:

Пример 2:

if1 a >= b and a == 0

then

b = b + 2

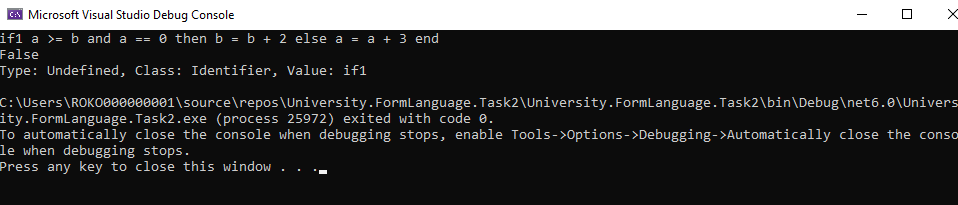
else

a = a + 3

end

Результат:

С ошибкой



Пример 3:

if a >= b and a < 0

then

шf b <= a or b >= 10

then

b = b \* 2

end

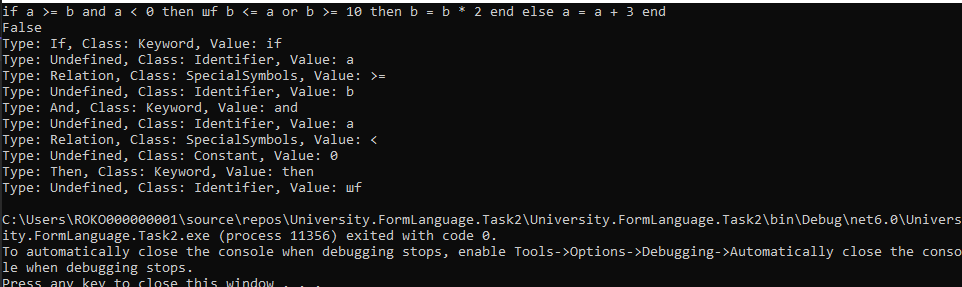
else

a = a + 3

end

Результат

С ошибкой



Листинг программы

**Program.cs**

// See https://aka.ms/new-console-template for more information

using University.FormLanguage.Task2;

LexemeAnalizer analizer = new LexemeAnalizer();

if (analizer.StartFindLexems("if a >= b and a < 0 then if b <= a or b >= 10 then b = b \* 2 end else a = a + 3 end"))

{

Console.WriteLine("True");

}

else

{

Console.WriteLine("False");

}

List<Lexeme> lexems = analizer.Lexems;

foreach (var item in lexems)

{

Console.WriteLine(item.ToString());

}

**LexemeAnalizer.cs**

using System.Text;

namespace University.FormLanguage.Task2

{

public class LexemeAnalizer

{

public List<Lexeme> Lexems { get; set; } = new List<Lexeme>();

public bool StartFindLexems(string text)

{

SystemState state = SystemState.Start;

SystemState prevSystemState;

bool isAbleToAdd;

text += " ";

StringBuilder lexBufNext = new StringBuilder();

StringBuilder lexBufCur = new StringBuilder();

int textIndex = 0;

while (state != SystemState.Error && state != SystemState.Final)

{

prevSystemState = state;

isAbleToAdd = true;

if (textIndex == text.Length && state != SystemState.Error)

{

state = SystemState.Final;

break;

}

if (textIndex == text.Length)

{

break;

}

char symbol = text[textIndex];

switch (state)

{

case SystemState.Start:

if (char.IsWhiteSpace(symbol))

{

state = SystemState.Start;

}

else if (char.IsDigit(symbol))

{

state = SystemState.Constant;

}

else if (char.IsLetter(symbol))

{

state = SystemState.Identifier;

}

else if (symbol == '>')

{

state = SystemState.Comparison;

}

else if (symbol == '<')

{

state = SystemState.ReverseComparison;

}

else if (symbol == '+' || symbol == '-' || symbol == '/' || symbol == '\*')

{

state = SystemState.ArithmeticOperation;

}

else if (symbol == '=')

{

state = SystemState.Assigment;

}

else

{

state = SystemState.Error;

}

isAbleToAdd = false;

if (!char.IsWhiteSpace(symbol))

{

lexBufCur.Append(symbol);

}

break;

case SystemState.Comparison:

if (char.IsWhiteSpace(symbol))

{

state = SystemState.Start;

}

else if (symbol == '=')

{

state = SystemState.Start;

lexBufCur.Append(symbol);

}

else if (char.IsLetter(symbol))

{

state = SystemState.Identifier;

lexBufNext.Append(symbol);

}

else if (char.IsDigit(symbol))

{

state = SystemState.Constant;

lexBufNext.Append(symbol);

}

else

{

state = SystemState.Error;

isAbleToAdd = false;

}

break;

case SystemState.ReverseComparison:

if (char.IsWhiteSpace(symbol))

{

state = SystemState.Start;

}

else if (symbol == '>')

{

state = SystemState.Start;

lexBufCur.Append(symbol);

}

else if (symbol == '=')

{

state = SystemState.Start;

lexBufCur.Append(symbol);

}

else if (char.IsLetter(symbol))

{

state = SystemState.Identifier;

lexBufNext.Append(symbol);

}

else if (char.IsDigit(symbol))

{

state = SystemState.Constant;

lexBufNext.Append(symbol);

}

else

{

state = SystemState.Error;

isAbleToAdd = false;

}

break;

case SystemState.Assigment:

if (symbol == '=')

{

state = SystemState.Comparison;

lexBufCur.Append(symbol);

}

else

{

state = SystemState.Start;

lexBufNext.Append(symbol);

}

break;

case SystemState.Constant:

if (char.IsWhiteSpace(symbol))

{

state = SystemState.Start;

}

else if (char.IsDigit(symbol))

{

isAbleToAdd = false;

state = SystemState.Constant;

lexBufCur.Append(symbol);

}

else if (symbol == '<')

{

state = SystemState.ReverseComparison;

lexBufNext.Append(symbol);

}

else if (symbol == '>')

{

state = SystemState.Comparison;

lexBufNext.Append(symbol);

}

else if (symbol == '=')

{

state = SystemState.Assigment;

lexBufNext.Append(symbol);

}

else if (symbol == '+' || symbol == '-' || symbol == '/' || symbol == '\*')

{

state = SystemState.ArithmeticOperation;

lexBufNext.Append(symbol);

}

else

{

state = SystemState.Error;

isAbleToAdd = false;

}

break;

case SystemState.Identifier:

if (char.IsWhiteSpace(symbol)) state = SystemState.Start;

else if (char.IsDigit(symbol) || char.IsLetter(symbol))

{

state = SystemState.Identifier;

isAbleToAdd = false;

lexBufCur.Append(symbol);

}

else if (symbol == '<')

{

state = SystemState.ReverseComparison;

lexBufNext.Append(symbol);

}

else if (symbol == '>')

{

state = SystemState.Comparison;

lexBufNext.Append(symbol);

}

else if (symbol == '=')

{

state = SystemState.Assigment;

lexBufNext.Append(symbol);

}

else if (symbol == '+' || symbol == '-' || symbol == '/' || symbol == '\*')

{

state = SystemState.ArithmeticOperation;

lexBufNext.Append(symbol);

}

else if (symbol == ':')

{

state = SystemState.Assigment;

lexBufNext.Append(symbol);

}

else

{

state = SystemState.Error;

isAbleToAdd = false;

}

break;

case SystemState.ArithmeticOperation:

if (char.IsWhiteSpace(symbol))

{

state = SystemState.Start;

}

else if (char.IsLetter(symbol))

{

state = SystemState.Identifier;

lexBufNext.Append(symbol);

}

else if (char.IsDigit(symbol))

{

state = SystemState.Constant;

lexBufNext.Append(symbol);

}

else if (symbol == '-' || symbol == '+' || symbol == '/' || symbol == '\*')

{

state = SystemState.ArithmeticOperation;

lexBufNext.Append(symbol);

}

else

{

state = SystemState.Error;

isAbleToAdd = false;

}

break;

}

if (isAbleToAdd)

{

if (!AddDiffLexeme(prevSystemState, lexBufCur.ToString()))

{

return false;

}

lexBufCur = new StringBuilder(lexBufNext.ToString());

lexBufNext.Clear();

}

textIndex++;

}

return state == SystemState.Final;

}

private bool AddDiffLexeme(SystemState prevState, string value)

{

LexemeType lexType = LexemeType.Undefined;

LexemeClass lexClass = LexemeClass.Undefined;

if (prevState == SystemState.ArithmeticOperation)

{

lexType = LexemeType.ArithmeticOperation;

lexClass = LexemeClass.SpecialSymbols;

}

else if (prevState == SystemState.Assigment)

{

lexClass = LexemeClass.SpecialSymbols;

if (value == "==")

{

lexType = LexemeType.Relation;

}

else

{

lexType = LexemeType.Assigment;

}

}

else if (prevState == SystemState.Constant)

{

lexType = LexemeType.Undefined;

lexClass = LexemeClass.Constant;

}

else if (prevState == SystemState.ReverseComparison)

{

lexType = LexemeType.Relation;

lexClass = LexemeClass.SpecialSymbols;

}

else if (prevState == SystemState.Comparison)

{

lexType = LexemeType.Relation;

lexClass = LexemeClass.SpecialSymbols;

}

else if (prevState == SystemState.Identifier)

{

bool isKeyword = true;

if (value.ToLower() == "if")

{

lexType = LexemeType.If;

}

else if (value.ToLower() == "and")

{

lexType = LexemeType.And;

}

else if (value.ToLower() == "or")

{

lexType = LexemeType.Or;

}

else if (value.ToLower() == "else")

{

lexType = LexemeType.Else;

}

else if (value.ToLower() == "then")

{

lexType = LexemeType.Then;

}

else if (value.ToLower() == "end")

{

lexType = LexemeType.End;

}

else

{

lexType = LexemeType.Undefined;

isKeyword = false;

}

if (isKeyword)

{

lexClass = LexemeClass.Keyword;

}

else

{

lexClass = LexemeClass.Identifier;

}

}

Lexeme lexeme = new()

{

Class = lexClass,

Type = lexType,

Value = value.Trim(),

};

if (Lexems.Count != 0)

{

var lastLexem = Lexems.Last();

if (lastLexem.Class == lexeme.Class)

{

return false;

}

}

if (lexeme.Value.Length > 0)

{

Lexems.Add(lexeme);

}

return true;

}

}

}

**LexemEnums.cs**

namespace University.FormLanguage.Task2

{

public enum LexemeType

{

If,

Then,

Else,

End,

And,

Or,

ArithmeticOperation,

Relation,

Assigment,

Undefined

}

public enum LexemeClass

{

Keyword,

Identifier,

Constant,

SpecialSymbols,

Undefined

}

public enum SystemState

{

Start,

Identifier,

Constant,

Error,

Final,

Comparison,

ReverseComparison,

ArithmeticOperation,

Assigment

}

}