Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ Н.Г.ЧЕРНЫШЕВСКОГО»

Кафедра информатики и программирования

ОТЧЕТ ПО ПРАКТИКЕ ДИСЦИПЛИНЫ «ТЕОРИЯ ФОРМАЛЬНЫХ ЯЗЫКОВ И ТРАНСЛЯЦИЙ»

Лабораторная работа №1

студента 4 курса 442 группы

направления 02.03.03 – Математическое обеспечение и администрирование

информационных систем (МОиАИС)

факультета компьютерных наук и информационных технологий (КНиИТ)

Мартышина Ивана Алексеевича

Саратов 2021

**Лабораторная работа 1**

**2.Задание на лабораторную работу**

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.

**Вариант 17 Конструкция 7 сложность 2**

**if** <лоогическое выражение> **then**

<операторы>

[ **elseif** <логическое выражение> **then**

<операторы> ]

[ **else**

<операторы> ]

**end**

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, letter

Description automatically generated

**3. Описание цепочек анализируемого языка**

Цепочка для идентификаторов:

S -> aA|bA|…|zA

A -> aA|bA|…|zA

A -> 0A|1A|…|9A

A -> \_|\_

Цепочка для констант:

S -> 0A|1A|…|9A

A -> 0A|1A|…|9A

A -> \_|\_

Цепочки для специальных символов:

1) S -> <A|>D

A -> >D| \_|\_

D -> \_|\_

2) S -> =B

B -> =E| \_|\_

E -> \_|\_

3) S -> +C|-C|/C|\*C

C -> \_|\_

**4. Таблица терминальных символов с подробным описанием**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Индекс** | **Лексема** | **Категория** | **Тип** | **Комментарий** |
| 0 | if | Ключевое слово | if | Определяет начало блока условного опратора «если» |
| 1 | else if | Ключевое слово | else if | Определяет начало блока условного опратора «иначе если» |
| 2 | else | Ключевое слово | else | Определяет начало блока условного опратора «иначе» |
| 3 | then | Ключевое слово | then | Переход к блоку с операндами |
| 4 | end | Ключевое слово | end | Определяет конец блока условного оператора |
| 5 | and | Ключевое слово | and | Логическое «И» |
| 6 | or | Ключевое слово | or | Логическое «ИЛИ» |
| 7 | < | Специальный символ | relation | Операция сравнения «меньше» |
| 8 | <= | Специальный символ | relation | Операция сравнения «меньше или равно» |
| 9 | <> | Специальный символ | relation | Операция сравнения «не равно» |
| 10 | == | Специальный символ | relation | Операция сравнения «равно» |
| 11 | = | Специальный символ | assignment | Операция присваивания |
| 12 | + | Специальный символ | arithmeticOp | Операция сложения |
| 13 | - | Специальный символ | arithmeticOp | Операция вычитания |
| 14 | / | Специальный символ | arithmeticOp | Операция деления |
| 15 | \* | Специальный символ | arithmeticOp | Операция умножения |

**5. Описание грамматики ввиде графа**

**Diagram

Description automatically generated with medium confidence**

**6. Описание основных алгоритмов и структрур данных, используемых в программе:**

**а) Описание структур данных, используемых для представления последовательности лексем, таблиц идентификаторов и констант;**

Класс Lexem необходим для работы с лексемами, он содержит информацию что за лексема, к какому классу относиться и само значение:

public class Lexeme

{

public LexemeType Type { get; set; }

public LexemeClass Class { get; set; }

public string Value { get; set; }

}

Для определения типов и категорий лексем используються следующие enum типы.

public enum LexemeType { Do, While, Loop, Not, And, Or, Output, Relation, ArithmeticOperation, Assignment, Undefined }

public enum LexemeClass { Keyword, Identifier, Constant, SpecialSymbols, Undefined }

**б) Описание процедур и функций, отвечающих за работу с введунными структурами данных;**

Класс Analyzer, имеет метод, который выполняет анализ полученный на вход строки из файла и начинает посимвольный ее разбор и состовляет класс Lexeme для каждой найденой лексемы, исходя из заданной ему грамматике в ходе прохода по строке он строит результирующую таблицу.

**в) описание алгоритма анализа автоматного языка.**

Алгоритм анализа автоматного языка работает по принципу перебора переданной ему на вход строки в которой содержиться «исходный код», во время прохода по строке начинаеться ее анализ, происходит отбор лексем, определяеться ее тип и к какой каткгории она относиться. Сам процесс определения типов и категории лексем проиходит с помощью графа, который строиться на основе цепочки, которые он получает в ходе обхода строки, после того как лексема была прогнана через граф, определяеться к какой категории и типу она относиться и после чего заноситься в таблицу.

**7. Описание интерфейса пользователя программы**

Необходимо занести в файл input.txt исходный код, который вы хотите передать в анализатор, после чего на экране будет отображена таблица с результатами проведенного анализа.

Пример:

**Graphical user interface, application, table

Description automatically generated**

**8. Листинг программы**

public class Lexeme

{

public LexemeType Type { get; set; }

public LexemeClass Class { get; set; }

public string Value { get; set; }

}

public enum LexemeType { If, Then, Elseif, Else, End, And, Or, Relation, ArithmeticOperation, Assignment, Undefined }

public enum LexemeClass { Keyword, Identifier, Constant, SpecialSymbols, Undefined }

public enum State { Start, Identifier, Constant, Error, Final, Comparison, ReverseComparison, ArithmeticOperation, Assignment }

public class Analyzer

{

public List<Lexeme> Lexemes { get; private set; }

public Analyzer()

{

Lexemes = new List<Lexeme>();

}

public bool Run(string text)

{

Lexemes = new List<Lexeme>();

State state = State.Start, prevState;

bool isAbleToAdd;

text += " ";

StringBuilder lexBufNext = new StringBuilder();

StringBuilder lexBufCur = new StringBuilder();

int textIndex = 0;

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

{

prevState = state;

isAbleToAdd = true;

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

{

state = State.Final;

break;

}

if (textIndex == text.Length)

{

break;

}

char symbol = text[textIndex];

switch (state)

{

case State.Start:

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

else if (char.IsDigit(symbol)) state = State.Constant;

else if (char.IsLetter(symbol)) state = State.Identifier;

else if (symbol == '>') state = State.Comparison;

else if (symbol == '<') state = State.ReverseComparison;

else if (symbol == '+' || symbol == '-' || symbol == '/' || symbol == '\*') state = State.ArithmeticOperation;

else if (symbol == '=') state = State.Assignment;

else state = State.Error;

isAbleToAdd = false;

if (!char.IsWhiteSpace(symbol))

lexBufCur.Append(symbol);

break;

case State.Comparison:

if (char.IsWhiteSpace(symbol))

{

state = State.Start;

}

else if (char.IsLetter(symbol))

{

state = State.Identifier;

lexBufNext.Append(symbol);

}

else if (char.IsDigit(symbol))

{

state = State.Constant;

lexBufNext.Append(symbol);

}

else

{

state = State.Error;

isAbleToAdd = false;

}

break;

case State.ReverseComparison:

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

else if (symbol == '>')

{

state = State.Start;

lexBufCur.Append(symbol);

}

else if (symbol == '=')

{

state = State.Start;

lexBufCur.Append(symbol);

}

else if (char.IsLetter(symbol))

{

state = State.Identifier;

lexBufNext.Append(symbol);

}

else if (char.IsDigit(symbol))

{

state = State.Constant;

lexBufNext.Append(symbol);

}

else

{

state = State.Error;

isAbleToAdd = false;

}

break;

case State.Assignment:

if (symbol == '=')

{

state = State.ReverseComparison;

lexBufCur.Append(symbol);

}

else if (char.IsWhiteSpace(symbol))

{

state = State.Start;

lexBufCur.Append(symbol);

}

else

{

state = State.Error;

isAbleToAdd = false;

}

break;

case State.Constant:

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

else if (char.IsDigit(symbol))

{

state = State.Constant;

lexBufCur.Append(symbol);

}

else if (symbol == '<')

{

state = State.ReverseComparison;

lexBufNext.Append(symbol);

}

else if (symbol == '>' || symbol == '=')

{

state = State.Comparison;

lexBufNext.Append(symbol);

}

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

{

state = State.ArithmeticOperation;

lexBufNext.Append(symbol);

}

else

{

state = State.Error;

isAbleToAdd = false;

}

break;

case State.Identifier:

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

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

{

state = State.Identifier;

isAbleToAdd = false;

lexBufCur.Append(symbol);

}

else if (symbol == '<')

{

state = State.ReverseComparison;

lexBufNext.Append(symbol);

}

else if (symbol == '>' || symbol == '=')

{

state = State.Comparison;

lexBufNext.Append(symbol);

}

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

{

state = State.ArithmeticOperation;

lexBufNext.Append(symbol);

}

else if (symbol == ':')

{

state = State.Assignment;

lexBufNext.Append(symbol);

}

else

{

state = State.Error;

isAbleToAdd = false;

}

break;

case State.ArithmeticOperation:

if (char.IsWhiteSpace(symbol))

{

state = State.Start;

}

else if (char.IsLetter(symbol))

{

state = State.Identifier;

lexBufNext.Append(symbol);

}

else if (char.IsDigit(symbol))

{

state = State.Constant;

lexBufNext.Append(symbol);

}

else

{

state = State.Error;

isAbleToAdd = false;

}

break;

}

if (isAbleToAdd)

{

AddLexeme(prevState, lexBufCur.ToString());

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

lexBufNext.Clear();

}

textIndex++;

}

return state == State.Final;

}

private void AddLexeme(State prevState, string value)

{

LexemeType lexType = LexemeType.Undefined;

LexemeClass lexClass = LexemeClass.Undefined;

if (prevState == State.ArithmeticOperation)

{

lexType = LexemeType.ArithmeticOperation;

lexClass = LexemeClass.SpecialSymbols;

}

else if (prevState == State.Assignment)

{

lexType = LexemeType.Assignment;

lexClass = LexemeClass.SpecialSymbols;

}

else if (prevState == State.Constant)

{

lexType = LexemeType.Undefined;

lexClass = LexemeClass.Constant;

}

else if (prevState == State.ReverseComparison)

{

lexType = LexemeType.Relation;

lexClass = LexemeClass.SpecialSymbols;

}

else if (prevState == State.Comparison)

{

lexType = LexemeType.Relation;

lexClass = LexemeClass.SpecialSymbols;

}

else if (prevState == State.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() == "then") lexType = LexemeType.Then;

else if (value.ToLower() == "elseif") lexType = LexemeType.Elseif;

else if (value.ToLower() == "else") lexType = LexemeType.Else;

else if (value.ToLower() == "end") lexType = LexemeType.End;

else

{

lexType = LexemeType.Undefined;

isKeyword = false;

}

if (isKeyword) lexClass = LexemeClass.Keyword;

else lexClass = LexemeClass.Identifier;

}

var lexeme = new Lexeme

{

Class = lexClass,

Type = lexType,

Value = value,

};

Lexemes.Add(lexeme);

}

}