Минобрнауки России

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«САРАТОВСКИЙ национальный исследовательский  
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ**

**Н. Г. ЧЕРНЫШЕВСКОГО»**

Факультет компьютерных наук и информационных технологий

**Теория формальных языков и трансляций.**

Лабораторная работа №1. Построение лексического анализатора

Студента 4 курса 441 группы

направления 02.03.03 — Математическое обеспечение и администрирование

информационных систем

факультета КНиИТ

Алексеева Олега Сергеевича

Саратов 2024

# ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

В задании необходимо было:

* Разработать автоматный лексический анализатор для языка, цепочки которого состоят из базовых лексических элементов: ключевых слов, идентификаторов, констант, специальных символов (операции сравнения, арифметические операции, разделители и т.д.).
* Реализовать лексический анализатор, который преобразует исходный текст в последовательность лексем и формирует таблицы идентификаторов и констант.

**do until <логическое выражение> <операторы> loop**

<логическое выражение> -> <выражение сравнения> |

<унарная логическая операция><выражение сравнения> |

<логическое выражение><бинарная логическая операция><выражение сравнения>

<выражение сравнения> -> <операнд> | <операнд><операция сравнения><операнд>

<операция сравнения> -> < | > | = | <>

<унарная логическая операция> -> **not**

<бинарная логическая операция> -> **and**|**or**

<операнд> -> <идентификатор> -> <константа>

<операторы> -> <операторы>; <оператор> | <оператор>

<оператор> -> <идентификатор> = <арифметическое выражение> |

**output** <операнд>

<арифметическое выражение> -> <операнд> |

<операнд ><арифметическая операция><арифметическое выражение>

<арифметическая операция> -> + | - | / | \*

# ОПИСАНИЕ ЦЕПОЧЕК АНАЛИЗИРУЕМОГО ЯЗЫКА

Анализируемый язык состоит из следующих базовых лексических элементов:

* **Ключевые слова.** Примеры: do, until, loop, output, not, and, or.
* **Идентификаторы.** Последовательности, начинающиеся с буквы, за которой могут следовать буквы и цифры. Пример: a, b, myVar.
* **Константы.** Последовательности цифр, представляющие целые числа (например, 10, 20).
* **Специальные символы.** Операторы сравнения (<, >, <>, ==), арифметические операторы (+, -, \*, /), а также разделители (например, точка с запятой ;).

Цепочка программы может иметь следующий вид:

do until not a < 10 and b <> 20

a = a + 1;

output a

loop

В данной цепочке:

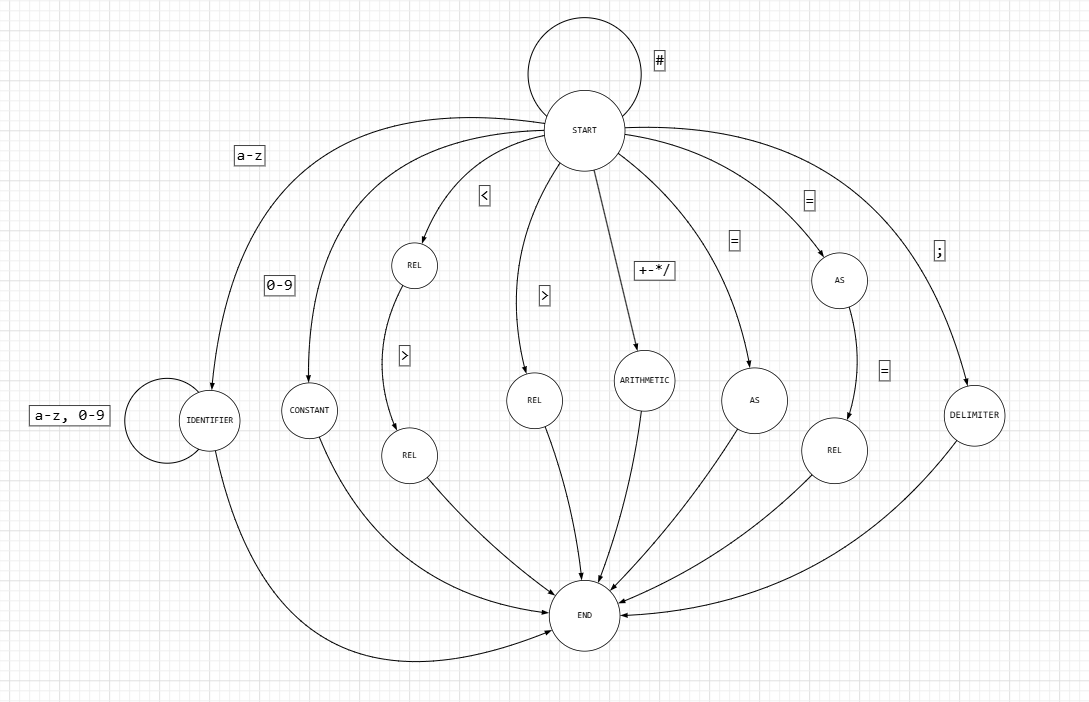
* Ключевыми словами являются do, until, loop, output, not, and.
* Идентификаторами – a и b.
* Константами – 10 и 20.
* Специальными символами – <, <>, =, ==, +, ;.

# ТАБЛИЦА ТЕРМИНАЛЬНЫХ СИМВОЛОВ

Ниже представлена таблица терминальных символов с подробным описанием:

| **Индекс** | **Символ** | **Категория** | **Тип лексемы** | **Комментарий** |
| --- | --- | --- | --- | --- |
| 0 | do | Ключевое слово | *Do* | Начало оператора цикла |
| 1 | until | Ключевое слово | *Until* | Условие цикла (предусловие) |
| 2 | loop | Ключевое слово | *Loop* | Завершение оператора цикла |
| 3 | output | Ключевое слово | *Output* | Команда вывода |
| 4 | not | Ключевое слово | *Not* | Унарная логическая операция |
| 5 | and | Ключевое слово | *And* | Бинарная логическая операция |
| 6 | or | Ключевое слово | *Or* | Бинарная логическая операция |
| 7 | < | Специальный символ | *Rel* | Оператор сравнения «меньше» |
| 8 | > | Специальный символ | *Rel* | Оператор сравнения «больше» |
| 9 | <> | Специальный символ | *Rel* | Оператор сравнения «не равно» |
| 10 | = | Специальный символ | As | Оператор присваивания |
| 11 | + | Специальный символ | *Plus* | Арифметическая операция (сложение) |
| 12 | - | Специальный символ | *Minus* | Арифметическая операция (вычитание) |
| 13 | \* | Специальный символ | *Multiply* | Арифметическая операция (умножение) |
| 14 | / | Специальный символ | *Divide* | Арифметическая операция (деление) |
| 15 | ; | Разделитель | *Delimiter* | Разделитель операторов (конец оператора) |
| 16 | == | Специальный символ | *Rel* | Оператор сравнения |

# ГРАФ СОСТОЯНИЙ

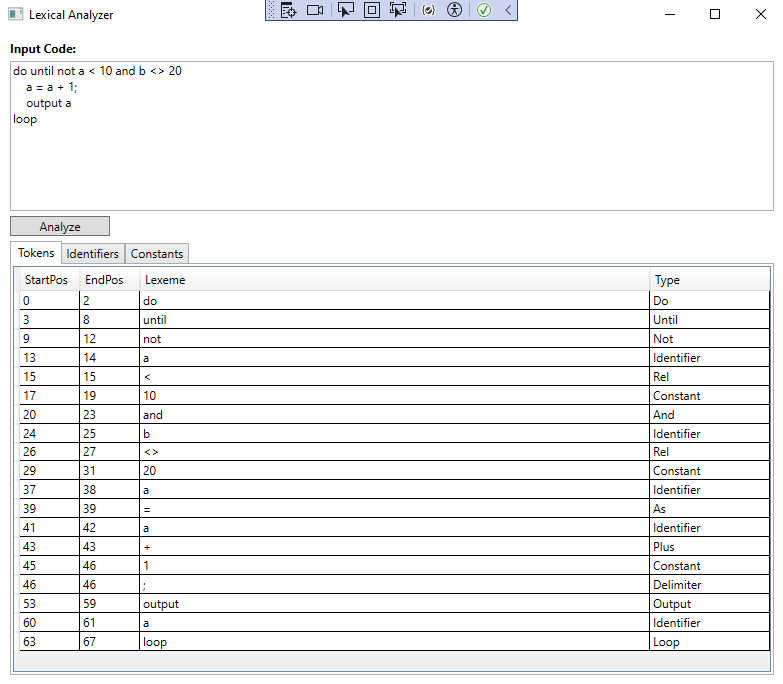
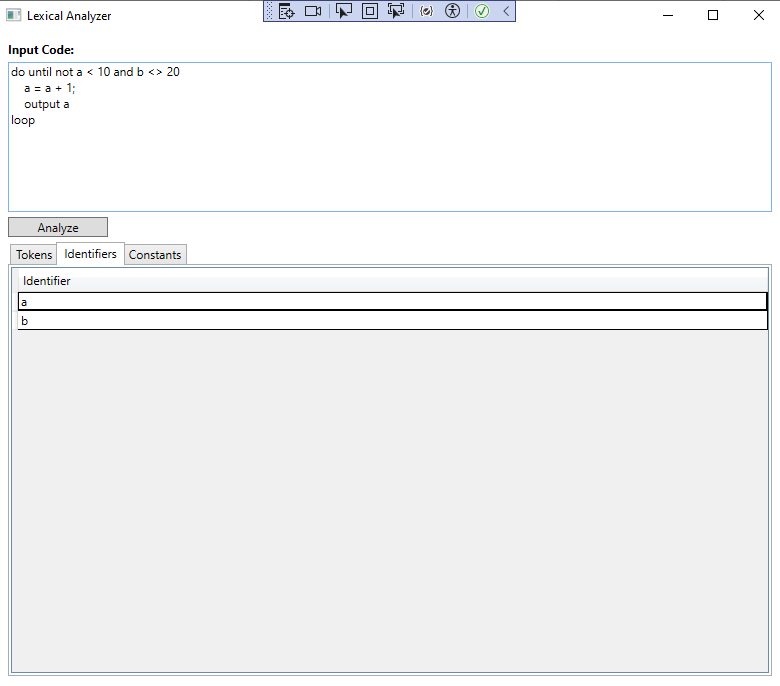
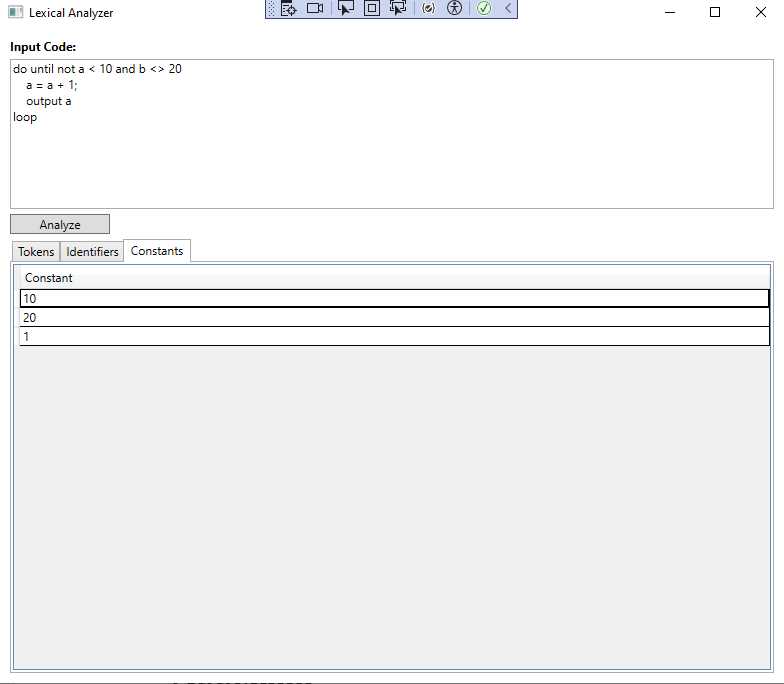


# ОПИСАНИЕ ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

Программа реализована в виде WPF‑приложения на C#. Основные элементы интерфейса:

* **TextBox.**  
  Предназначен для ввода исходного кода программы, подлежащего лексическому анализу.
* **"Analyze".**  
  При нажатии на кнопку запускается процесс лексического анализа введённого текста.
* **TabControl.**  
  На отдельных вкладках отображаются:
  + **Tokens.** Список полученных лексем с указанием позиции, текста лексемы и её типа.
  + **Identifiers.** Таблица уникальных идентификаторов, найденных в исходном тексте.
  + **Constants.** Таблица констант.

# КОНТРОЛЬНЫЙ ПРИМЕР И РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ

# ЛИСТИНГ ПРОГРАММЫ

**LexAnalyzer.cs**

using System;

using System.Collections;

using System.Collections.Generic;

using Lab1.Models;

namespace Lab1.Services

{

public class LexAnalyzer

{

public List<Token> Tokens { get; private set; } = new List<Token>();

public List<string> Identifiers { get; private set; } = new List<string>();

public List<string> Constants { get; private set; } = new List<string>();

private Dictionary<string, TokenType> keywords = new Dictionary<string, TokenType>()

{

{ "do", TokenType.Do },

{ "until", TokenType.Until },

{ "loop", TokenType.Loop },

{ "output", TokenType.Output },

{ "not", TokenType.Not },

{ "and", TokenType.And },

{ "or", TokenType.Or }

};

public bool Analyze(string input)

{

int pos = 0;

while (pos < input.Length)

{

char c = input[pos];

if (char.IsWhiteSpace(c))

{

pos++;

continue;

}

if (char.IsLetter(c))

{

int start = pos;

while (pos < input.Length && char.IsLetterOrDigit(input[pos]))

{

pos++;

}

string lexeme = input.Substring(start, pos - start);

string lowerLexeme = lexeme.ToLower();

if (keywords.ContainsKey(lowerLexeme))

{

Token token = new Token { Type = keywords[lowerLexeme], Lexeme = lexeme, StartPos = start, EndPos = start };

Tokens.Add(token);

}

else

{

Token token = new Token { Type = TokenType.Identifier, Lexeme = lexeme, StartPos = start, EndPos = start };

Tokens.Add(token);

if (!Identifiers.Contains(lexeme))

Identifiers.Add(lexeme);

}

continue;

}

else if (char.IsDigit(c))

{

int start = pos;

while (pos < input.Length && char.IsDigit(input[pos]))

{

pos++;

}

string lexeme = input.Substring(start, pos - start);

Token token = new Token { Type = TokenType.Constant, Lexeme = lexeme, StartPos = start, EndPos = start };

Tokens.Add(token);

if (!Constants.Contains(lexeme))

Constants.Add(lexeme);

continue;

}

else

{

if (c == '<')

{

if (pos + 1 < input.Length && input[pos + 1] == '>')

{

Token token = new Token { Type = TokenType.Rel, Lexeme = "<>", StartPos = pos + 1, EndPos = pos + 1 };

Tokens.Add(token);

pos += 2;

continue;

}

else

{

Token token = new Token { Type = TokenType.Rel, Lexeme = "<", StartPos = pos, EndPos = pos};

Tokens.Add(token);

pos++;

continue;

}

}

else if (c == '>')

{

Token token = new Token { Type = TokenType.Rel, Lexeme = ">", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else if (c == '=')

{

if (pos + 1 < input.Length && input[pos + 1] == '=')

{

Token token = new Token { Type = TokenType.Rel, Lexeme = "==", StartPos = pos + 1, EndPos = pos + 1 };

Tokens.Add(token);

pos += 2;

continue;

}

else

{

Token token = new Token { Type = TokenType.As, Lexeme = "=", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

}

else if (c == '+')

{

Token token = new Token { Type = TokenType.Plus, Lexeme = "+", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else if (c == '-')

{

Token token = new Token { Type = TokenType.Minus, Lexeme = "-", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else if (c == '\*')

{

Token token = new Token { Type = TokenType.Multiply, Lexeme = "\*", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else if (c == '/')

{

Token token = new Token { Type = TokenType.Divide, Lexeme = "/", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else if (c == ';')

{

Token token = new Token { Type = TokenType.Delimiter, Lexeme = ";", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else

{

Token token = new Token { Type = TokenType.Unknown, Lexeme = c.ToString(), StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

}

}

return true;

}

}

}

**Token.cs**

namespace Lab1.Models

{

public class Token

{

public TokenType Type { get; set; }

public string Lexeme { get; set; }

public int Position { get; set; }

public int StartPos { get; set; }

public int EndPos { get; set; }

}

}

**TokenType.cs**

namespace Lab1.Models

{

public enum TokenType

{

Do,

Until,

Loop,

Output,

Not,

And,

Or,

Identifier,

Constant,

Rel,

As,

Plus,

Minus,

Multiply,

Divide,

Delimiter,

Unknown

}

}