МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Ульяновский государственный технический университет**»**

Кафедра «Вычислительная техника**»**

**Лабораторная работа №1**

по дисциплине «Системное программное обеспеченье»

«Лексический анализ»

Выполнил студент

гр. ИВТАСбд-31

Сутулов К. О.

Ульяновск, 2022

# Задание

Используя платформу .NET, необходимо реализовать лексический анализатор для языка программирования “C”. Лексический анализатор должен поддерживать операторы do, while, арифметические операции, операцию присвоения, инкремент, объявление переменных.

# Ход работы

Суть лексического анализатора заключается в разбиении программы на лексемы – последовательности символов, имеющие смысловое значение в программе. Для хранения лексем была реализована структура Lexem, в которой содержатся тип, идентификатор и значение лексемы:

public class Lexeme : BaseModel

{

public enum LexemeType

{

ParsingError = -1,

DataType = 0,

Variable = 1,

Keysymbol = 2,

Identifier = 3,

Constant = 4,

Operator = 5,

Keyword = 6,

}

public LexemeType Type;

public int? Id;

public string? Value;

public Lexeme() {}

public Lexeme(LexemeType type, int id, string value, Dictionary<string, string>? attributes = null)

{

Id = id;

Type = type;

Value = value;

if (attributes != null)

Attributes = attributes;

}

public override string ToString()

{

return base.ToJSON<Lexeme>();

}

}

Также для удобства все лексемы, формирующие объявление идентификаторов, будем заносить в отдельную структуру Variable, где будет храниться id идентификатора, его имя, а также дополнительные атрибуты:

namespace lab1.Models

{

public class Variable : BaseModel

{

public string? Type;

public string? Name;

public string? Id;

public Variable() {}

public Variable(string? type, string? name, string? id, Dictionary<string, string>? attributes = null)

{

Id = id;

Type = type;

Name = name;

if (attributes != null)

base.Attributes = attributes;

}

public override string ToString()

{

return base.ToJSON<Variable>();

}

}

}

Далее разрабатывается непосредственно сам лексический анализатор. На вход ему подается строка, содержащая саму программу. Сам лексический анализатор содержит в себе так называемые таблицы символов, которые содержат все возможные лексемы языка, за исключением идентификаторов, которые определяются программистом. Аттрибуты структуры выглядят следующим образом:

public class LexicalAnalyzer

{

public string Program;

private List<string> \_dataTypes = new List<string>() {"int", "short", "char", "long", "bool", "void", "string", "float", "double" };

private List<string> \_operators = new List<string>() { "=", "+", "-", "\*", "/", "==", "!=", "<", ">", ">=", "<=", "%", "+=", "-=", "++", "--", "%", "!", "&", "&&", "||", "|" };

private List<string> \_keysymbols = new List<string>() { ";", "{", "}", "(", ")", "[", "]", ".", "," };

private List<string> \_keywords = new List<string>() { "do", "while", "for", "if", "else", "return", "break" };

private List<string> \_strsymbols = new List<string>() { "\"", "\'" };

public LexicalAnalyzer(string program) { Program = program; }

}

Далее реализуется метод токенизации входной программы. Была составлена недетерминированная однопроходная модель разбора, алгоритм которой состоит в следующем: последовательно записываем в буфер по одному символу программы, и после каждого добавления выполняем проверку на то, что содержимое буфера является либо константой, либо входит в таблицу символов, либо является корректным объявлением идентификатора. При отсутствии соответствия выбрасывается ошибка о неопознанном символе. Также был реализован выброс базовых ошибок компиляции, таких как забытая кавычка в строке или имя идентификатора, начинающееся с цифры. Результатом метода является структура TokenizedProgram, которая содержит в себе массив лексем и таблицу идентификаторов:

public TokenizedProgram TokenizeProgram()

{

TokenizedProgram tokenizedProgram = new TokenizedProgram();

List<string> allSymbols = \_dataTypes.Concat(\_dataTypes).Concat(\_operators).Concat(\_keywords).Concat(\_keysymbols).Concat(\_strsymbols).ToList();

string buffer = String.Empty;

int iter = 0;

int varId = 0;

int identificatorId = 0;

int constantId = 0;

int lineCounter = 1;

while (iter < Program.Length)

{

char currentChar = Program[iter];

char nextChar = (iter + 1) < Program.Length ? Program[iter + 1] : '\n';

iter++;

if (char.IsWhiteSpace(currentChar))

{

if (currentChar == '\n') lineCounter++;

buffer = string.Empty;

continue;

};

buffer += currentChar;

bool prevWasVarDeclaration = tokenizedProgram.Lexemes.Count > 0 && \_dataTypes.Contains(tokenizedProgram.Lexemes.Last().Value ?? "");

bool bufferIsNumber = double.TryParse(buffer, out \_);

bool isUnknownSymbol = !char.IsLetterOrDigit(currentChar) && currentChar != '\_' && !allSymbols.Contains(currentChar.ToString());

var nextCharIsSpaceOrKeysymbol = () => char.IsWhiteSpace(nextChar) || allSymbols.Contains(nextChar.ToString());

if (currentChar == '\"') // read string

{

bool isBadStr = false;

while (nextChar != '\"')

{

iter++;

buffer += nextChar;

if (nextChar == '\n')

{

tokenizedProgram.Lexemes.Add(

new Lexeme(Lexeme.LexemeType.ParsingError, -5, $"Error at line {lineCounter} <Missing quote>")

);

isBadStr = true;

break;

}

nextChar = Program[iter];

}

if (isBadStr) break;

buffer += nextChar;

tokenizedProgram.Lexemes.Add(new Lexeme(Lexeme.LexemeType.Constant, constantId++, buffer, new Dictionary<string, string>() { { "Type", "string" } }));

buffer = string.Empty;

iter++;

}

else if (currentChar == '\'') // read char

{

buffer += nextChar;

if (nextChar == '\\')

{

iter++;

nextChar = Program[iter];

buffer += nextChar;

}

iter++;

nextChar = Program[iter];

buffer += nextChar;

if (nextChar != '\'')

{

tokenizedProgram.Lexemes.Add(

new Lexeme(Lexeme.LexemeType.ParsingError, -6, $"Error at line {lineCounter} <Bad char value>")

);

break;

}

iter++;

tokenizedProgram.Lexemes.Add(new Lexeme(Lexeme.LexemeType.Constant, constantId++, buffer, new Dictionary<string, string>() { { "Type", "char" } }));

buffer = string.Empty;

}

else if (buffer == "true" || buffer == "false")

{

tokenizedProgram.Lexemes.Add(new Lexeme(Lexeme.LexemeType.Constant, constantId++, buffer, new Dictionary<string, string>() { { "Type", "bool" } }));

buffer = string.Empty;

}

else if (\_dataTypes.Contains(buffer)) // read type

{

tokenizedProgram.Lexemes.Add(

new Lexeme(Lexeme.LexemeType.DataType, \_dataTypes.IndexOf(buffer), buffer)

);

buffer = string.Empty;

}

else if (\_operators.Contains(buffer + nextChar)) //read double operator

{

tokenizedProgram.Lexemes.Add(

new Lexeme(Lexeme.LexemeType.Operator, \_operators.IndexOf(buffer + nextChar), buffer + nextChar)

);

buffer = string.Empty;

iter++;

}

else if (\_operators.Contains(buffer)) // read operator

{

tokenizedProgram.Lexemes.Add(new Lexeme(Lexeme.LexemeType.Operator, \_operators.IndexOf(buffer), buffer));

buffer = string.Empty;

}

else if (\_keysymbols.Contains(buffer) && !(buffer == "." && double.TryParse(nextChar.ToString(), out \_))) // read keysymbol

{

tokenizedProgram.Lexemes.Add(new Lexeme(Lexeme.LexemeType.Keysymbol, \_keysymbols.IndexOf(buffer), buffer));

buffer = string.Empty;

}

else if (\_keywords.Contains(buffer) && !(buffer + nextChar == "dou")) // read keyword

{

tokenizedProgram.Lexemes.Add(new Lexeme(Lexeme.LexemeType.Keyword, \_keywords.IndexOf(buffer), buffer));

buffer = string.Empty;

}

else if (bufferIsNumber && nextCharIsSpaceOrKeysymbol()) // read number

{

bool isReal = false;

int fractionalCnt = 0;

if (nextChar == '.')

{

isReal = true;

buffer += nextChar;

iter++;

nextChar = Program[iter];

while (!nextCharIsSpaceOrKeysymbol())

{

fractionalCnt++;

buffer += nextChar;

iter++;

nextChar = Program[iter];

}

}

if (prevWasVarDeclaration)

{

tokenizedProgram.Lexemes.Add(

new Lexeme(Lexeme.LexemeType.ParsingError, -4, $"Error at line {lineCounter} <Identificator \"{buffer}\" cannot be a number>")

);

break;

}

if (isReal && (!double.TryParse(buffer, NumberStyles.Float, CultureInfo.InvariantCulture, out \_) || fractionalCnt == 0))

{

tokenizedProgram.Lexemes.Add(

new Lexeme(Lexeme.LexemeType.ParsingError, -7, $"Error at line {lineCounter} <Bad float number \"{buffer}\">")

);

break;

}

if (!isReal && buffer.Length > 1 && buffer[0] == '0')

{

tokenizedProgram.Lexemes.Add(

new Lexeme(Lexeme.LexemeType.ParsingError, -7, $"Error at line {lineCounter} <Bad int number \"{buffer}\">")

);

break;

}

tokenizedProgram.Lexemes.Add(new Lexeme(Lexeme.LexemeType.Constant, constantId++, buffer, new Dictionary<string, string>() { { "Type", isReal ? "float" : "int" } }));

buffer = string.Empty;

}

else if (char.IsLetterOrDigit(currentChar) && nextCharIsSpaceOrKeysymbol()) // read identifier

{

if (char.IsDigit(buffer[0]))

{

tokenizedProgram.Lexemes.Add(

new Lexeme(Lexeme.LexemeType.ParsingError, -3, $"Error at line {lineCounter} <Identificator \"{ buffer }\" starts with number>")

);

break;

}

//if (!tokenizedProgram.Identificators.Exists(v => v.Name == buffer) && !prevWasVarDeclaration)

//{

// tokenizedProgram.Lexemes.Add(

// new Lexeme(Lexeme.LexemeType.ParsingError, -1, $"Error at line {lineCounter} <Unexpected identificator \"{buffer}\">")

// );

// break;

//}

if (prevWasVarDeclaration) {

tokenizedProgram.Identificators.Add(

new Variable(tokenizedProgram.Lexemes.Last().Value, buffer, varId.ToString())

);

varId++;

}

tokenizedProgram.Lexemes.Add(new Lexeme(Lexeme.LexemeType.Identifier, identificatorId++, buffer));

buffer = string.Empty;

}

else if (isUnknownSymbol) // throw error

{

tokenizedProgram.Lexemes.Add(new Lexeme(Lexeme.LexemeType.ParsingError, -2, $"Error at line {lineCounter} <Cannot resolve symbol \"{ buffer }\">"));

break;

}

}

return tokenizedProgram;

}

# Тестирование программы

Для тестирования программы был составлен следующий код:

internal class Program

{

static void Main(string[] args)

{

string program = File.ReadAllText("../../../Input/program1.txt");

LexicalAnalyzer analizer = new LexicalAnalyzer(program);

var tokenized = analizer.TokenizeProgram();

Console.WriteLine("============== Токены ===========================\n");

tokenized.Lexemes.ForEach(v => Console.WriteLine($"Тип: \"{v.Type}\", ID: \"{v.Id}\", Значение: {v.Value}"));

Console.WriteLine("\n============== Идентификаторы ======================\n");

tokenized.Identificators.ForEach(v => Console.WriteLine($"Тип: \"{v.Type}\", ID: \"{v.Id}\" Имя: \"{v.Name}\""));

}

}

Во входной файл program1.txt была введена программа на языке С, соответствующая всем условиям технического задания. Результат работы программы приведен далее:

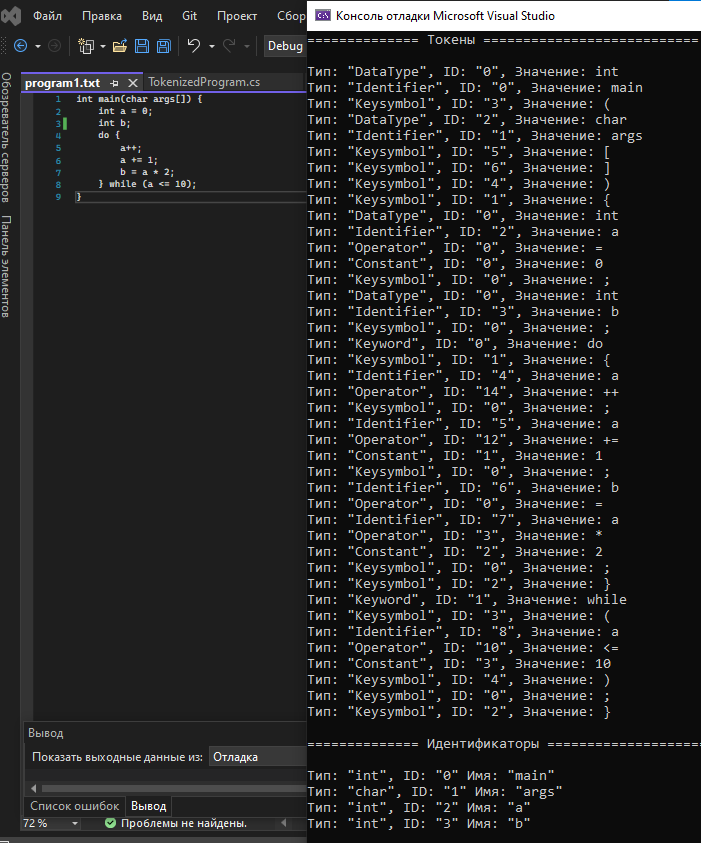


Рис. 1. Результат работы программы

Выходные данные программы, как и ожидалось, соответствуют ожидаемому результату.

# Вывод

В процессе выполнения лабораторной работы с использованием платформы .NET была составлена программная реализация лексического анализатора языка “С”. Были определены таблицы символов компилируемого языка, разработан алгоритм разбора и проведено тестирование кода.