Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЕТ

к лабораторной работе №2

на тему

**ЛЕКСИЧЕСКИЙ АНАЛИЗ**

Студент В. П. Бычко

Преподаватель Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc160203922)

[2 Краткие теоретические сведения 4](#_Toc160203923)

[3 Результаты выполнения лабораторной работы 6](#_Toc160203924)

[Выводы 9](#_Toc160203925)

[Cписок использованных источников 10](#_Toc160203926)

[Приложение А (обязательное) Листинг кода 11](#_Toc160203927)

## 1 ЦЕЛЬ РАБОТЫ

Целью выполнения данной лабораторной работы является разработка лексического анализатора подмножества языка программирования, определенного в лабораторной работе 1. Также необходимо обнаружить ошибку при определении неверной последовательности символов и сообщить о ней. Всего необходимо показать нахождение четырех лексических ошибок.

## 2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Лексический анализатор представляет собой первую фазу компилятора, его основная задача состоит в чтении входных символов исходной программы, их группировании в лексемы и вывод последовательностей токенов для всех лексем исходной программы.

Лексема – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своём составе других структурных единиц языка.[1]

Поток токенов пересылается синтаксическому анализатору для разбора. Обычно при работе лексический анализатор взаимодействует также с таблицей символов. Когда лексический анализатор встречается с лексемой, составляющей идентификатор, эту лексему требуется внести в таблицу символов. В некоторых случаях лексический анализатор может получать из таблицы символов некоторую информацию об идентификаторах, которая может помочь ему верно определить передаваемый синтаксическому анализатору токен. Обычно взаимодействие реализуется как вызов лексического анализатора синтаксическим анализатором.

Этот вызов, представленный как команда getNextToken, заставляет лексический анализатор читать символы из входного потока, пока он не сможет идентифицировать очередную лексему и вернуть синтаксическому анализатору корректный токен.[2]

Поскольку лексический анализатор является частью компилятора, которая читает исходный текст, он может заодно выполнять и некоторые другие действия, помимо идентификации лексем. Одной из таких задач является отбрасывание комментариев и пробельных символов (пробел, символы табуляции и новой строки, а также, возможно, некоторые другие символы, использующиеся для отделения токенов друг от друга во входном потоке). Еще одной задачей является синхронизация сообщений об ошибках, генерируемых компилятором, с исходной программой. Например, лексический анализатор может отслеживать количество символов новой строки, чтобы каждое сообщение об ошибке сопровождалось номером строки, в которой она обнаружена. В некоторых компиляторах лексический анализатор создает копию исходной программы с сообщениями об ошибках, вставленными в соответствующие места исходного текста.

## 3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

В ходе выполнения лабораторной работы был реализован лексический анализатор языка Swift. На рисунке 3.1 представлен результат запуска программы.

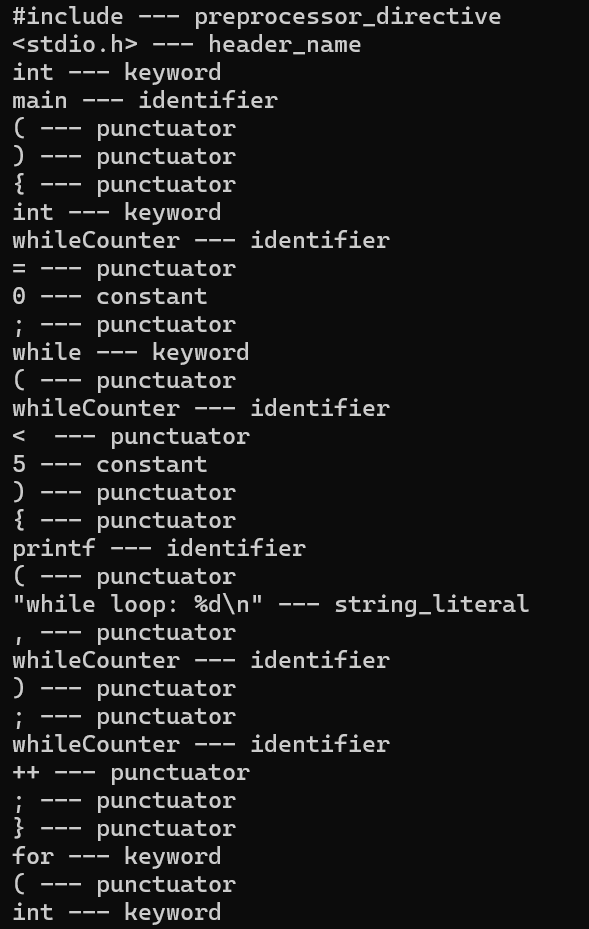


Рисунок 3.1 – Выполнение программы

В исходном коде намеренно допущена ошибка в числовой константе. На рисунке 3.2 изображено, что лексический анализатор отработал корректно и вывел сообщение об ошибке



Рисунок 3.2 – Лексическая ошибка в числовой константе

В исходном коде намеренно допущена ошибка: введен пустой символьный литерал. На рисунке 3.3 изображено, что лексический анализатор отработал корректно и вывел сообщение об ошибке.



Рисунок 3.3 – Пустой символьный литерал в тексте программы

В исходном коде намеренно допущена ошибка: введена слишком большая символьная константа. На рисунке 3.4 изображено, что лексический анализатор отработал корректно и вывел сообщение об ошибке.



Рисунок 3.4 – Слишком большая символьная константа

В исходном коде намеренно допущена ошибка в написании суффиксов числовой константы. На рисунке 3.5 изображено, что лексический анализатор отработал корректно и вывел сообщение об ошибке



Рисунок 3.5 – Неправильная комбинация целочисленных суффиксов

Ниже представлен корректный код программы

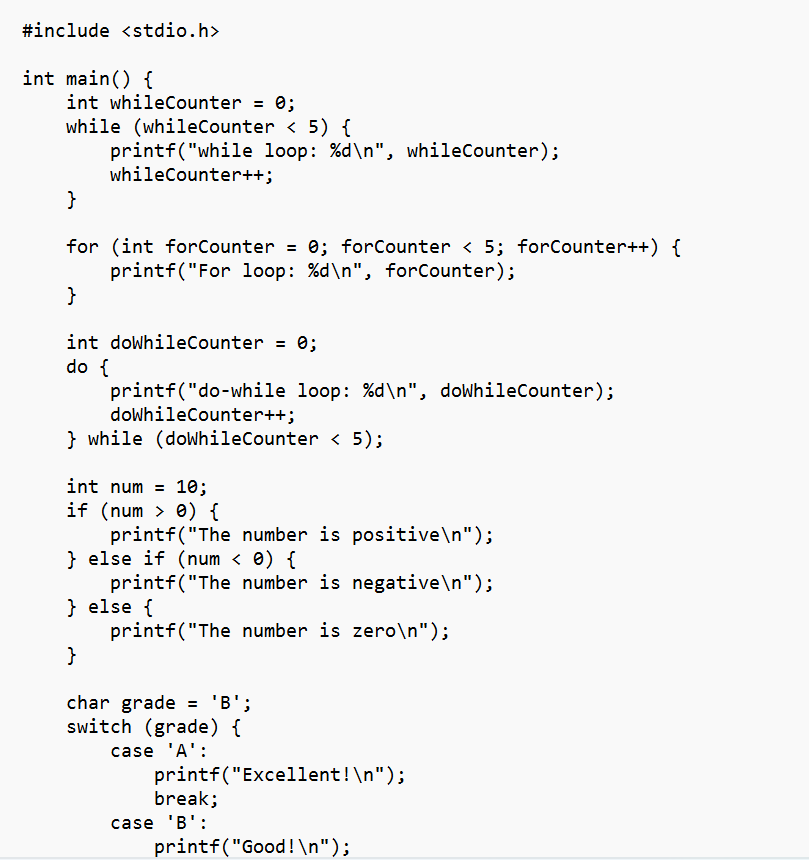


Рисунок 3.6 – Анализируемый корректный код

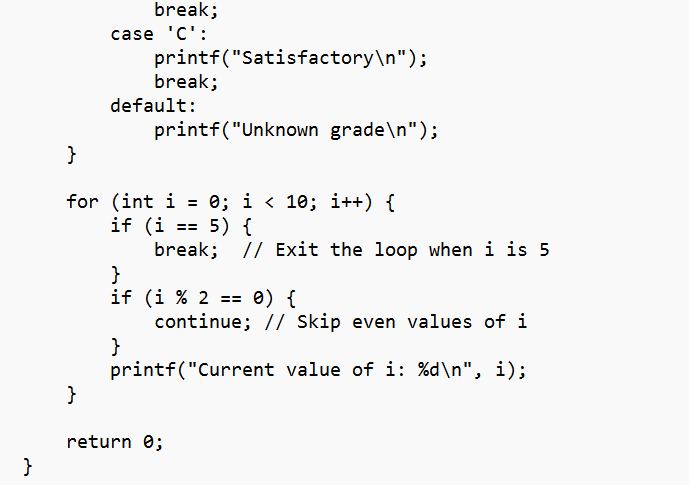


Рисунок 3.6 – Лист 2

# ВЫВОДЫ

В ходе выполнения данной лабораторной работы был разработан лексический анализатор подмножества языка программирования С. Также были обнаружены ошибки при определении неверной последовательности символов. Был продемонстрирован результат работы программы при обнаружении некорректных лексем.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Лексический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/LabWork2.pdf – Дата доступа: 15.02.2024

[2] Введение в теорию компиляторов [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/515420/

# ПРИЛОЖЕНИЕ А (обязательное) Листинг кода

**Reader.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace CLexer

{

public class Reader

{

private StreamReader reader;

char current;

char next;

bool realEnd = false;

private ReaderState state = ReaderState.InitialState;

public Reader(string FilePath)

{

string path = Path.Combine(Directory.GetParent(Directory.GetCurrentDirectory())!.Parent!.Parent!.FullName, FilePath);

reader = new StreamReader(path);

char[] nextSymbol = new char[1];

reader.Read(nextSymbol, 0, 1);

current = nextSymbol[0];

}

public char NextCharacter()

{

char[] nextSymbol = new char[1];

while (!realEnd)

{

if (reader.Read(nextSymbol, 0, 1) == 0)

realEnd = true;

next = nextSymbol[0];

switch (state)

{

#region Initial State

case ReaderState.InitialState:

{

switch (current)

{

case '/':

{

if (next != '/' && next != '\*')

{

return DoReturn();

}

state = ReaderState.PossibleComment;

break;

}

case '\"':

{

state = ReaderState.StringLiteral;

var val = current; current = next; return val;

}

default:

{

return DoReturn();

}

}

break;

}

#endregion

#region Possible Comment

case ReaderState.PossibleComment:

{

switch (current)

{

case '/':

{

state = ReaderState.InitialState;

reader.ReadLine();

break;

}

case '\*':

{

state = ReaderState.MultiLineComment;

break;

}

default:

{

state = ReaderState.InitialState;

return DoReturn();

}

}

break;

}

#endregion

#region Multi line comment

case ReaderState.MultiLineComment:

{

switch (current)

{

case '\*':

{

if (next == '/')

state = ReaderState.InitialState;

// надо чтобы убрать символ комментария

next = ' ';

break;

}

default:

{

break;

}

}

break;

}

#endregion

#region string or char literal

case ReaderState.StringLiteral:

{

switch (current)

{

case '\\':

{

state = ReaderState.EscapeCharacter;

return DoReturn();

}

case '\"':

{

state = ReaderState.InitialState;

return DoReturn();

}

case '\r':

{

state = ReaderState.PossibleEndLine;

return DoReturn();

}

default:

{

return DoReturn();

}

}

}

#endregion

#region escaping character

case ReaderState.EscapeCharacter:

{

state = ReaderState.StringLiteral;

return DoReturn();

}

#endregion

#region Possible end of line

case ReaderState.PossibleEndLine:

{

if (current == '\n')

{

throw new Exception("End of line in string literal");

}

else

{

state = ReaderState.StringLiteral;

return DoReturn();

}

}

#endregion

}

current = next;

}

if (state != ReaderState.InitialState)

throw new Exception("End of file at bad place");

else

return '\0';

}

private enum ReaderState

{

PossibleComment,

MultiLineComment,

InitialState,

StringLiteral,

PossibleEndLine,

WhiteSpace,

EscapeCharacter

}

private char DoReturn()

{

var val = current;

current = next;

return val;

}

}

}