**

Колледж космического машиностроения и технологий

**КУРСОВОЙ ПРОЕКТ**

**По МДК.01.02 «Прикладное программирование»**

**Тема: «Разработка парсера учебного языка программирования»»**

Выполнил студент

Фатеев Даниил Иванович

Группа П1-17

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_(подпись)

\_\_\_\_\_\_\_\_\_\_\_ (Дата сдачи работы)

Королев, 2020

Оглавление

[Задание на выполнение курсового проекта 3](#_Toc44088567)

[Краткое описание программы: 3](#_Toc44088568)

[Полное описание задачи: 3](#_Toc44088569)

[Введение 4](#_Toc44088570)

[1.Теоретическая часть 5](#_Toc44088571)

[1.1 Структура компиляторов и интерпретаторов 5](#_Toc44088572)

[1.2 Лексер 5](#_Toc44088573)

[1.3 Программы для лексического анализа 6](#_Toc44088574)

[1.4 Парсер 7](#_Toc44088575)

[1.5 Программы для синтаксического анализа 8](#_Toc44088576)

[1.6 Генератор и объединяющий модуль 8](#_Toc44088577)

[2. Проектная часть 8](#_Toc44088578)

[2.1 Инструменты для разработки 8](#_Toc44088579)

[2.1.2 Описание приложения 9](#_Toc44088580)

[2.2 Описание функций программы 9](#_Toc44088581)

[2.2.1 Файлы 9](#_Toc44088582)

[2.2.2 Структура файлов 10](#_Toc44088583)

[2.3 Подробный разбор некоторых функций 12](#_Toc44088584)

[2.3.1 Функция next() 12](#_Toc44088585)

[2.3.2 Функция is\_space() 14](#_Toc44088586)

[2.4 Диаграммы прецендентов и классов программы 17](#_Toc44088587)

[2.4.1 Диаграммы прецендентов 17](#_Toc44088588)

[2.4.2 Диаграмма классов 17](#_Toc44088589)

[3.Организационная часть 18](#_Toc44088590)

[3.1 Назначение программы 18](#_Toc44088591)

[3.2 Установка программы 18](#_Toc44088592)

[3.3 Руководство пользователя 18](#_Toc44088593)

[Заключение 20](#_Toc44088594)

[Источники 21](#_Toc44088595)

Задание на выполнение курсового проекта

## **Краткое описание программы:**

Разработать текстовый анализатор (парсер) учебного языка программирования на ПК с использованием GCC и C++.

## **Полное описание задачи:**

Разработать для приложения модули:

1. Модуль токенов и их функций.

Модуль содержит в себе имена токенов языка (лексем)

и функции проверки для недействительных лексем языка (Русские буквы в имени переменной и т.д.).

1. Модуль Лексера и его функций.

Модуль содержит в себе функции обработки текста программы,

в случае ошибки вызывает проверку из первого модуля. Посимвольно обрабатывает текст программы и в конце возвращает список символов языка с определенными токенами.

1. Главный модуль.

Принимает на вход текст программы из текстового файла и передает его функции Лексера на посимвольную обработку.

**Входные данные:**

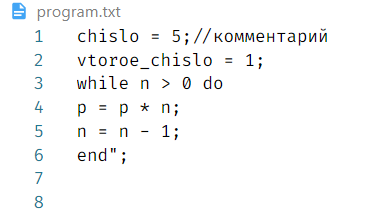
На вход подается текстовый файл программы на любом языке программирования. Пример такого файла:

Рисунок 0.1

**Выходные данные:**

После компилирования и запуска программы на экране консоли (CMD, PowerShell, bash, Terminal) будет выведен отформатированный список токенов программы и определенных токенами лексем:

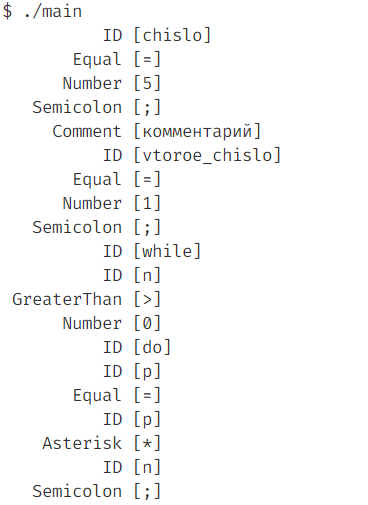


Рисунок 0.2

## **Введение**

В процессе разработки используется GCC компилятор, что позволяет запускать программу на любой системе где есть терминал и установлен компилятор языка C++. Для поддержания данной кроссплатформенности требуется исключить зависимости как от библиотек систем Linux, так и от библеотек системы Windows, что может потребовать либо:

Поиска переписанной на нужную платформу библиотеки.

Использование функций-замен там где это возможно.

В первой части рассмотрим предметную область и похожие программы. Во второй части будут разобраны инструменты, библиотеки, структура программной части и листинги модулей. В третьей части будет рассмотрено взаимодействие пользователя с программой. В заключении будут сделаны выводы о проекте и полученных при работе с ним знаниях.

# **1.Теоретическая часть**

## **1.1 Структура компиляторов и интерпретаторов**

Каждый компилятор или интерпретатор состоит из следующих модулей:

* Лексический анализатор (лексер)
* Синтаксический анализатор (парсер)
* Генератор или стековая машина
* Объединяющий модуль

Рассмотрим каждый из них поподробнее.

## **1.2 Лексер**

Данный модуль проводит лексический анализ текста программы. Лексический анализ представляет собой разбиение текста на токены, то есть единицы языка: переменные, названия функций (идентификаторы), операторы, числа. Таким образом, подав лексеру на вход строку с исходным кодом, мы получим на выходе список всех токенов, которые в ней содержатся. Обращения к исходному коду уже не будет происходить на следующих этапах, поэтому лексер должен выдать всю необходимую для них информацию.

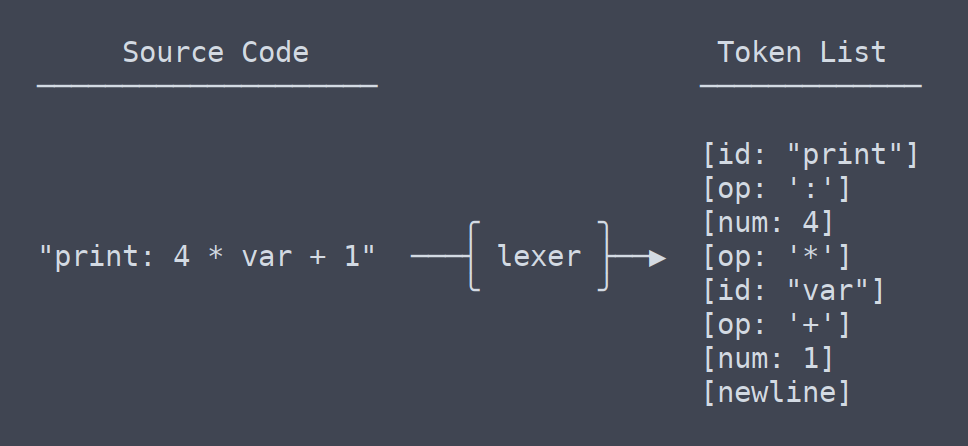


Рисунок 1.2. Схема Лексера

**1.3 Программы для лексического анализа**

Для создания Лексера существует ряд специальных программ – генераторы лексических анализаторов. Самыми популярными являются GNU Flex и Yacc, они получают на вход программу с описанием грамматики языка (Рис.1.3) и генерируют программу на С, которая проводит лексический анализ (Рис.1.4).

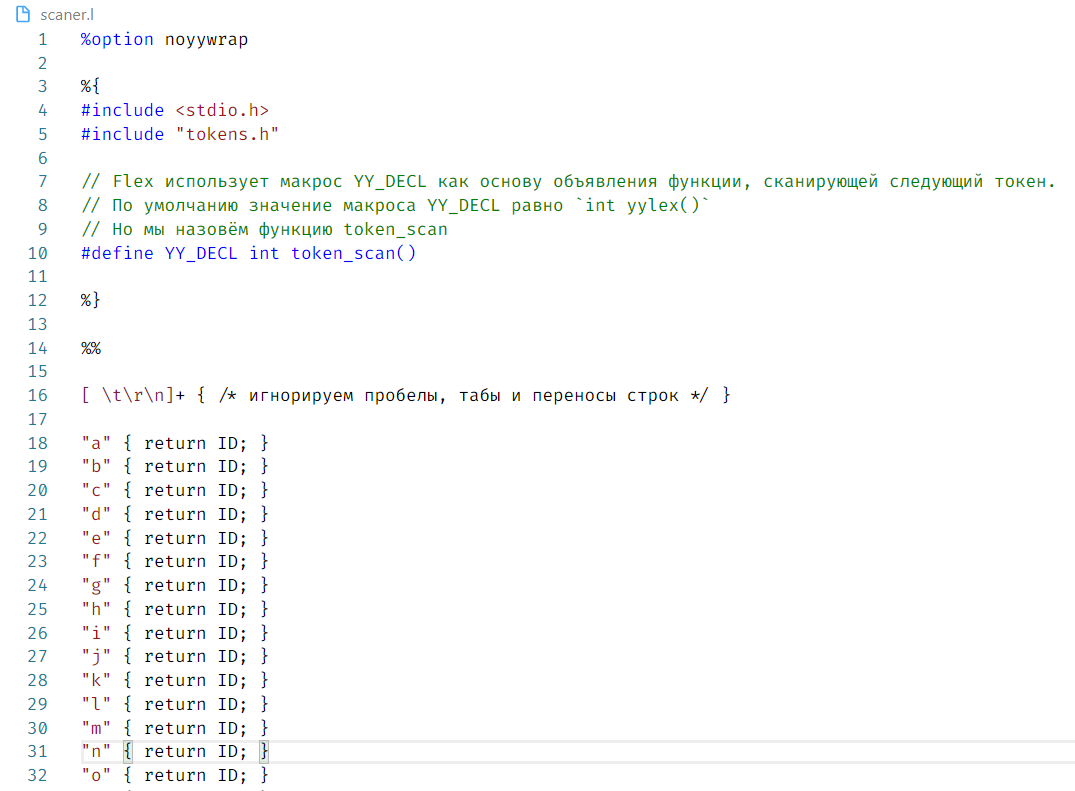


Рисунок 1.3. Файл грамматики



Рисунок 1.4. Сгенерированный лексический анализатор

**1.4 Парсер**

Синтаксический анализатор получает на вход список токенов из Лексера и генерирует АСД (Абстрактное Синтаксическое Дерево), которое позволяет структурно представить правила языка.

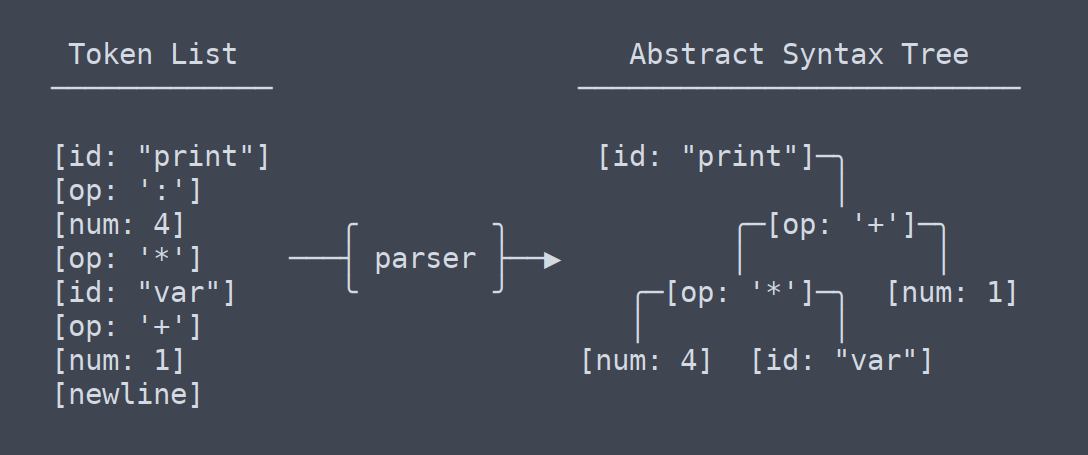


Рисунок 0.1

**1.5 Программы для синтаксического анализа**

Самым популярным генератором синтаксических анализаторов является Bison. По принципу работы во многом похож на Flex и Yacc, потому что также генерирует программу с синтаксическими правилами на языке С.

**1.6 Генератор и объединяющий модуль**

Если язык программирования интерпретируемый, то для получения значений программы нужно запустить главный модуль, который запустит все вышеперечисленные модули и выполнит программу. Если язык компилируемый, то без дополнительных средств не обойтись. Такие инструменты называют «Компиляторами компиляторов». Самый популярный среди разработчиков LLVM. Имеет множество надстроек и библиотек и может оказаться весьма сложным в освоении.

Мнение о программах: имеют огромный функционал, но будут полезны лишь в крупном проекте и энтузиастам. Имеют порты на Windows, но лучше работать с ними в Linux, или через MinGW и Cygwin.

**2. Проектная часть**

**2.1 Инструменты для разработки**

**VSCode** – бесплатный редактор кода от компании Microsoft. Позволяет работать с кодом, не отвлекаясь на посторонние окна. Имеет возможности тонкой настройки через settings.json. Позволяет устанавливать расширения: Git, IntelliSense, bash-терминал и Cygwin.

**Cygwin –** бесплатная UNIX среда с интерфейсом командной строки, является «портом» Linux на Windows. Состоит из библиотеки cygwin.dll, обеспечивающую совместимость API и соответствие стандартам POSIX и библиотеки приложений, обеспечивающих UNIX-среду. Единственный минус: не работает с Windows-библиотеками, а при использовании библиотек Linux появляется зависимость от cygwin.dll.

**GNU C++ (g++) –** бесплатный компилятор C++ от GNU. Очень простой и настраиваемый компилятор. Установлен в Cygwin.

**2.1.2 Описание приложения**

Данное приложение является простым, но хорошо настраиваемым лексическим анализатором. Состоит из:

1. Модуль токенов и их функций.

Модуль содержит в себе имена токенов языка (лексем)

и функции проверки для недействительных лексем языка (Русские буквы в имени переменной и т.д.).

1. Модуль Лексера и его функций.

Модуль содержит в себе функции обработки текста программы,

в случае ошибки вызывает проверку из первого модуля. Посимвольно обрабатывает текст программы и в конце возвращает список символов языка с определенными токенами.

1. Главный модуль.

Принимает на вход текст программы из текстового файла и передает его функции Лексера на посимвольную обработку.

После обработки выводит результат в командную строку и файл.

Пользователь может работать с приложением при помощи shell или другого терминала.

## 

## **2.2 Описание функций программы**

### **2.2.1 Файлы**

В таблице ниже приведены все файлы и их описание:

|  |  |
| --- | --- |
| Имя файла | Описание |
| main.cpp | Главный исполняемый файл |
| lexer.cpp | Файл, реализующий лексический анализ и методы из файла lexer.hpp |
| lexer.hpp | Заголовочный файл, содержащий в себе прототипы функций и конструкторов |
| tokens.cpp | Файл, реализующий функции, и конструкторы из файла tokens.hpp |
| tokens.hpp | Заголовочный файл содержащий в себе класс имен токенов, прототипы функций и конструкторов |

**2.2.2 Структура файлов**

Таблица 1 содержит описание классов в файлах. В таблице 2 описываются все функции программы, к каким классам они принадлежат:

|  |  |  |
| --- | --- | --- |
| Имя файла | Имя класса | Описание класса |
| tokens.hpp | Token | Содержит конструкторы и прототипы функций для определения токенов |
| tokens.hpp | Tokens | Вложенный класс-перечисление имен токенов |
| lexer.hpp | Lexer | Содержит конструкторы и прототипы функций для определения токенов. Использует элементы из классов Token и Tokens |

Таблица 1

|  |  |  |
| --- | --- | --- |
| Имя класса | Имя функции | Описание функции |
| Lexer | Lexer (const char\*cod) | Принимает массив символов и обрабатывает при помощи функций. |
| Lexer | next() | Сопоставляет символ с таблицей и возвращает имя токена |
| Lexer | identifier () | Проверка на идентификатор |
| Lexer | number () | Проверка на число |
| Lexer | Lexema () | Получает на вход результат из tokens() возвращает символ |
|  | bool is\_space(char c) | Проверка на символ |
|  | bool is\_digit(char c) | Проверка на число |
|  | bool is\_identifier\_char(char c) | Проверка на идентификатор |
| Token | tokens() | Возвращает значение токенов |
| Token | bool is (Tokens tokens) | Проверка токена |
| Token | bool is\_not(Tokens tokens) | Проверка токена |
| Token | bool is\_one\_of(Tokens t1, Tokens t2) | Проверка токена |
| Token | string\_view lexeme () | Получает значение имя токена |
| Token | void lexeme (string\_view lexeme) | Выводит имя токена |

Таблица 2

Остановимся на некоторых важных функциях.

## **2.3 Подробный разбор некоторых функций**

### **2.3.1 Функция next()**

Листинг функции next() приведен ниже. Данная функция при помощи операторов case проверяет и символ и возвращает нужное имя токена.

Листинг 1:

//Функция next() сопоставляет символ с таблицей

//и возвращает имя токена

Token Lexer::next() noexcept

{

    //Пока символ не пробел проверяем его

    while (is\_space(peek()))

        get();

    switch (peek())

    {

    case '\0':

        return Token(Token::Tokens::End, p\_cod, 1);

    default:

        return lexema(Token::Tokens::Unexpected);

    case 'a':

    case 'b':

    case 'c':

    case 'd':

    case 'e':

    case 'f':

    case 'g':

    case 'h':

    case 'i':

    case 'j':

    case 'k':

    case 'l':

    case 'm':

    case 'n':

    case 'o':

    case 'p':

    case 'q':

    case 'r':

    case 's':

    case 't':

    case 'u':

    case 'v':

    case 'w':

    case 'x':

    case 'y':

    case 'z':

    case 'A':

    case 'B':

    case 'C':

    case 'D':

    case 'E':

    case 'F':

    case 'G':

    case 'H':

    case 'I':

    case 'J':

    case 'K':

    case 'L':

    case 'M':

    case 'N':

    case 'O':

    case 'P':

    case 'Q':

    case 'R':

    case 'S':

    case 'T':

    case 'U':

    case 'V':

    case 'W':

    case 'X':

    case 'Y':

    case 'Z':

        return identifier();

    case '0':

    case '1':

    case '2':

    case '3':

    case '4':

    case '5':

    case '6':

    case '7':

    case '8':

    case '9':

        return number();

    case '(':

        return lexema(Token::Tokens::LeftParen);

    case ')':

        return lexema(Token::Tokens::RightParen);

    case '[':

        return lexema(Token::Tokens::LeftSquare);

    case ']':

        return lexema(Token::Tokens::RightSquare);

    case '{':

        return lexema(Token::Tokens::LeftCurly);

    case '}':

        return lexema(Token::Tokens::RightCurly);

    case '<':

        return lexema(Token::Tokens::LessThan);

    case '>':

        return lexema(Token::Tokens::GreaterThan);

    case '=':

        return lexema(Token::Tokens::Equal);

    case '+':

        return lexema(Token::Tokens::Plus);

    case '-':

        return lexema(Token::Tokens::Minus);

    case '\*':

        return lexema(Token::Tokens::Asterisk);

    case '/':

        return lexema(Token::Tokens::Slash);

    case '#':

        return lexema(Token::Tokens::Sharp);

    case '.':

        return lexema(Token::Tokens::Dot);

    case ',':

        return lexema(Token::Tokens::Comma);

    case ':':

        return lexema(Token::Tokens::Colon);

    case ';':

        return lexema(Token::Tokens::Semicolon);

    case '\'':

        return lexema(Token::Tokens::SingleQuote);

    case '"':

        return lexema(Token::Tokens::DoubleQuote);

    case '|':

        return lexema(Token::Tokens::Pipe);

    case '&':

        return lexema(Token::Tokens::Ampersand);

    case '@':

        return lexema(Token::Tokens::Dog);

    }

}

### **2.3.2 Функция is\_space()**

Эта функция вызывается функцией next() для проверки символа, является ли он пробелом, табуляцией, возвратом каретки или началом новой строки. Ниже приведен листинг и блок схема.

Листинг 2:

//Проверка на пробелы табы и переходы строки

bool is\_space(char c) noexcept

{

    switch (c)

    {

    case ' ':

    case '\t':

    case '\r':

    case '\n':

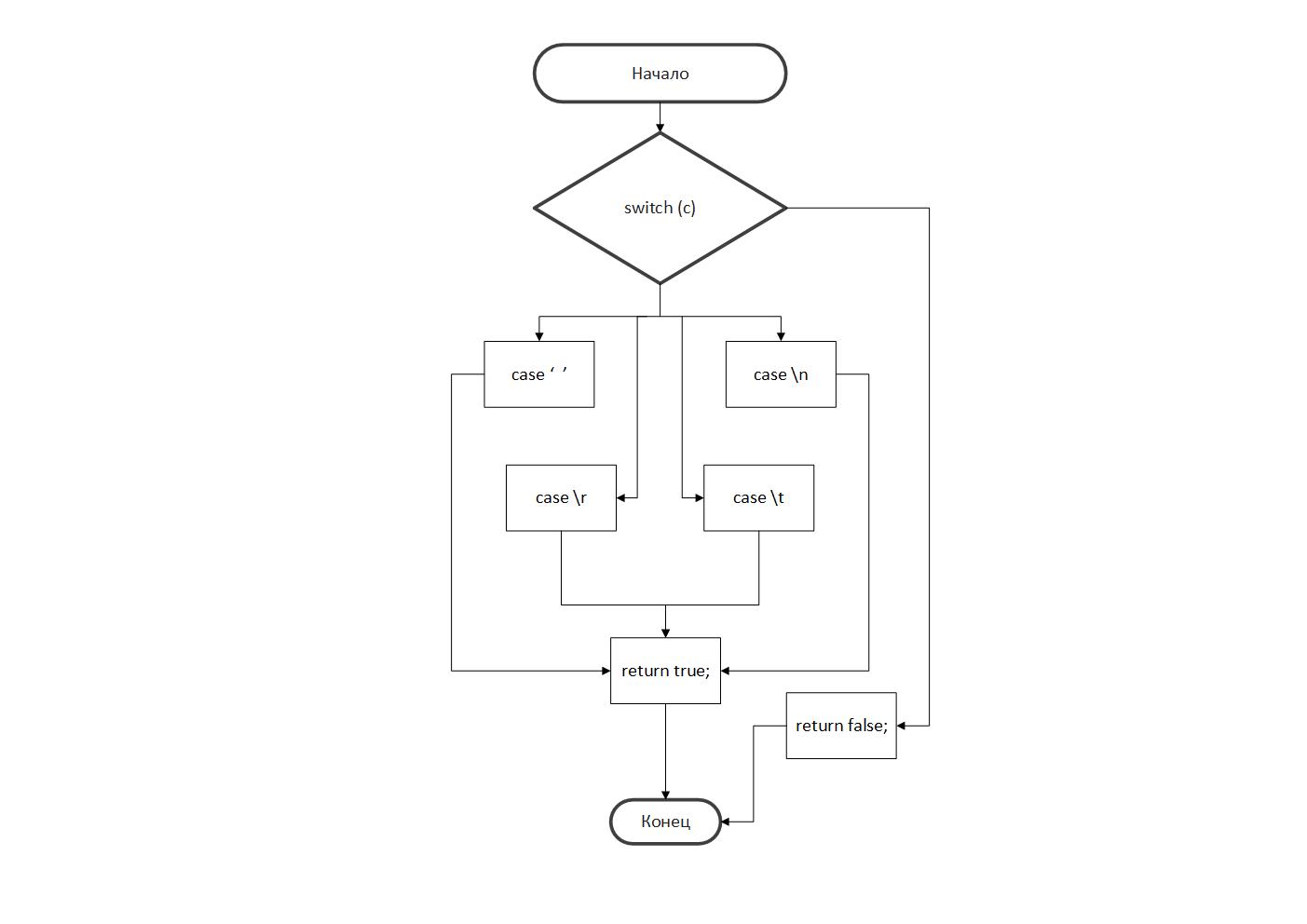
        return true;

    default:

        return false;

    }

}

****

Блок-схема модуля

## **2.4 Диаграммы прецендентов и классов программы**

### **2.4.1 Диаграммы прецендентов**

Существует несколько способов использования данной программы,

данные способы приведены на диаграммах прецендентов.

Диаграмма 1 иллюстрирует использование программы студентом:

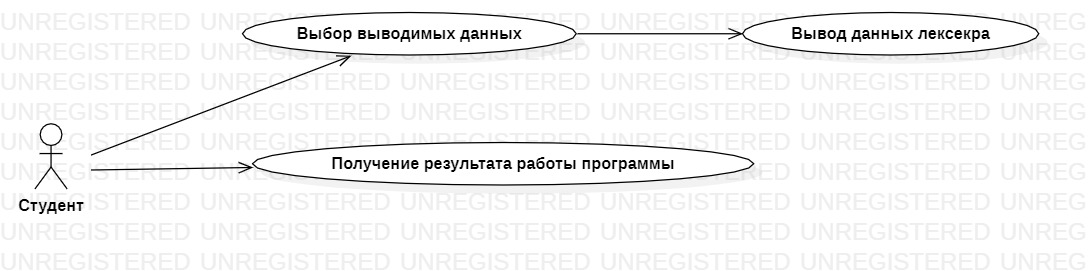


Диаграмма 2 иллюстрирует использование программы программистом:

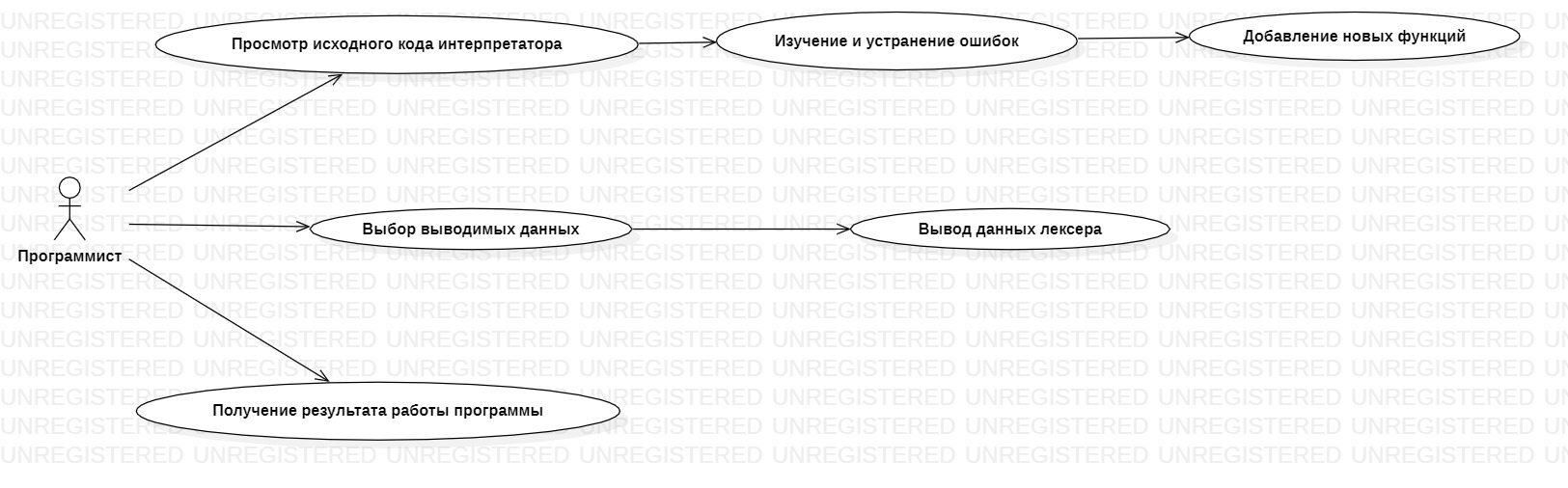
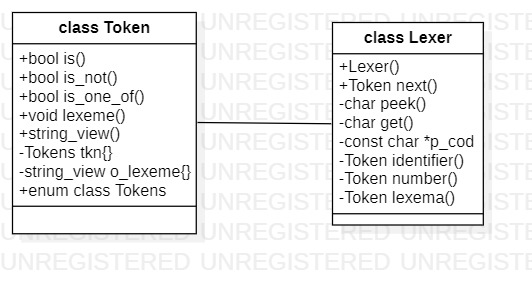


Диаграмма 3 иллюстрирует использование программы преподавателем:



### **2.4.2 Диаграмма классов**

Диаграмма классов наглядно показывает в каких зависимостях друг от друга находятся классы программы.



На Диаграмме 4 виден вид связи классов ассоциация, поскольку класс Lexer использует некоторые функции из класса Token.

**3.Организационная часть**

**3.1 Назначение программы**

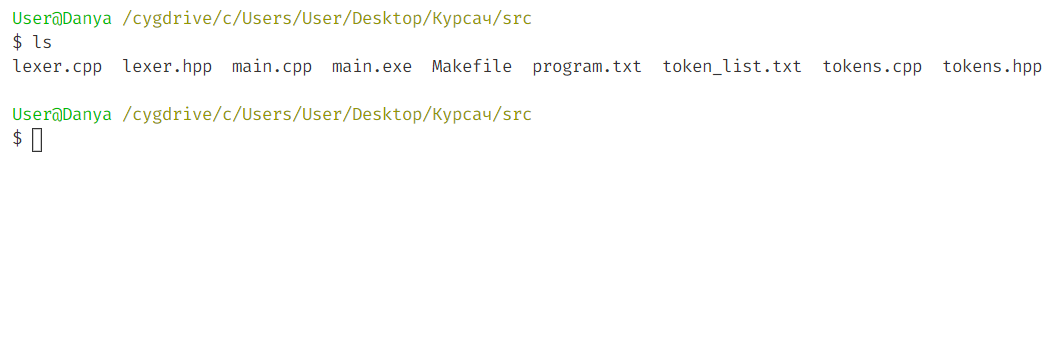
Программа представляет из себя готовый модуль лексера языка программирования. Этот модуль легкий быстрый. Легко модифицируем для еще большего количества символов. Очень прост в использовании. Можно использовать отдельно или в качестве модуля в программах распознавания текста.

**3.2 Установка программы**

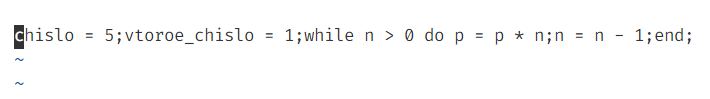
Для запуска программы не требуется установка. На компьютере должен находиться только компилятор языка С++ или терминал Linux.

## **3.3** **Руководство пользователя**

Данная программа удобна для использования в системе Linux, но ее также легко можно использовать в Windows.

1.Для правильной работы программы поместите все файлы в одну директорию: 

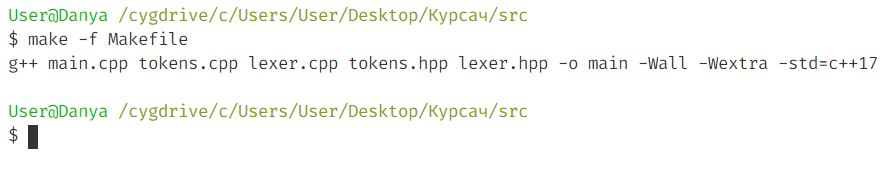
2. При помощи текстового редактора (в данном случае vim), отредактируйте файл program.txt записав программу в одну строку.



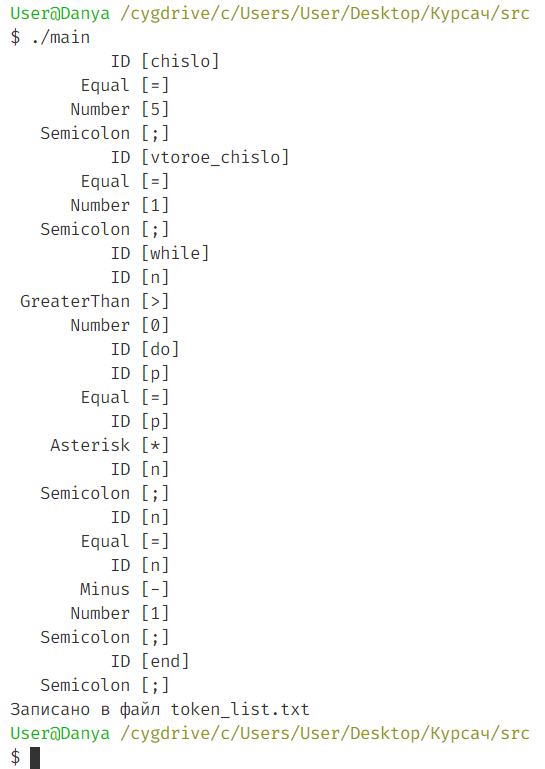
3.Сохраните и выйдете из редактора.



4.Запустите Makefile:



5.Запустите программу:



После запуска мы видим выведенный список токенов, который был параллельно записан в файл token\_list.txt.

# **Заключение**

В ходе курсовой работы был разработан лексический анализатор текста на ПК. Была изучена работа схожих программ. А также возможности стандарта С++ 17 (оптимизация и ускорение работы кода). Были умножены знания об ООП парадигме данного языка программирования и применены некоторые методы ускорения компиляции кода.

Области применения:

* Модуль компиляторов и интерпретаторов
* Модуль в программах распознавания текста

Доработка:

В будущем планируется совместить программу с модулем синтаксического анализатора для создания собственного языка программирования, изучить работу CMake и создать как графический, так и консольный интерфейс.

# **Источники**

<https://tproger.ru/translations/how-to-create-programming-language/>

<https://habr.com/ru/post/266589/>

<https://ru.wikipedia.org/wiki/C%2B%2B17>