



Государственное бюджетное образовательное учреждение высшего образования
Московской области

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

Колледж космического машиностроения и технологий

КУРСОВОЙ ПРОЕКТ

По МДК.01.02 «Прикладное программирование»

Тема: «Разработка лексера учебного языка программирования»

Выполнил студент

Фатеев Д.И.

Группа П1-17

_____ (подпись)

_____ (Дата сдачи работы)

Проверил преподаватель

Гусятинер Л.Б.

_____ (подпись)

Королев, 2020

Оглавление

1. Теоретическая часть	5
1.1 Структура компиляторов и интерпретаторов	5
1.2 Лексер	5
1.3 Программы для лексического анализа	6
1.4 Парсер	9
1.5 Программы для синтаксического анализа	9
1.6 Генератор и объединяющий модуль	9
2. Проектная часть	10
2.1 Инструменты для разработки	10
2.1.2 Описание приложения	10
2.2 Описание функций программы	11
2.2.1 Файлы	11
2.2.3 Обзор функций C++17 примененных в программе	13
2.3 Подробный разбор некоторых функций	14
2.3.1 Функция <code>main()</code>	14
2.3.2 Функция <code>next()</code>	15
2.3.3 Функция <code>is_space()</code>	18
2.3.5 Функция <code>is_digit()</code>	21
2.4 Диаграммы прецедентов и классов программы	21
2.4.2 Диаграмма классов	23
2.4.3 Сценарий работы программы	24
3. Эксплуатационная часть	25
3.1 Назначение программы	25
3.2 Установка программы	25

3.3 Руководство оператора	25
Заключение	28
Список литературы и интернет источников	29
Приложение 1.Файл main.cpp	30
Приложение 2.Файл lexer.hpp	31
Приложение 3.Файл lexer.cpp	32
Приложение 4.Файл tokens.cpp	39
Приложение 5.Файл tokens.hpp	41

Введение

Целью данного курсового проекта будет создание лексического анализатора (лексер). Лексический анализатор - важная часть любого компилятора и интерпретатора. Он должен быть быстрым и безошибочно определять символы, выдавая им правильное имя токена.

В процессе разработки используется GCC компилятор, что позволяет запускать программу на любой системе где есть терминал и установлен компилятор языка C++ стандарта 17. Для поддержания данной кроссплатформенности требуется исключить зависимости как от библиотек систем Linux, так и от библиотек системы Windows, что может потребовать либо:

- Поиска переписанной на нужную платформу библиотеки.
- Использования функций-замен там где это возможно.

В теоретической части будут подробно рассмотрены похожие программы и модули. В проектной части будут рассмотрены инструменты реализации, файлы программы, классы, некоторые функции, и диаграммы проектирования. В эксплуатационной части будет представлено руководство пользователя. В заключении будут сделаны выводы о проделанной работе. Также будут приведены источники и приложения с кодом.

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 подробнее.

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 (рис.3).

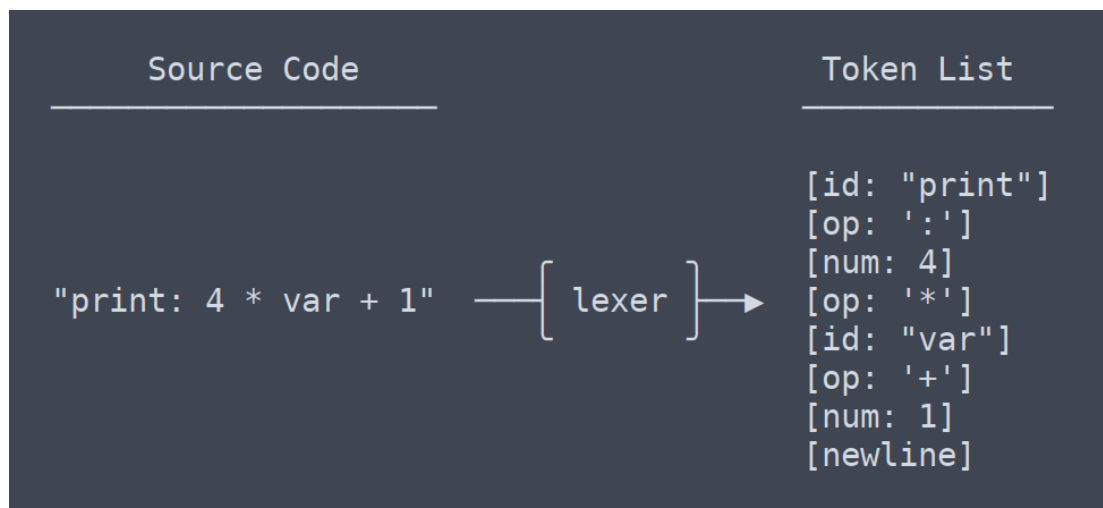


Рисунок 3 – схема работы лексера

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 (Листинг 1) и генерируют программу на C, которая проводит лексический анализ.

Листинг 1. Файл грамматики программы flex:

```
%option noyywrap

%{
#include <stdio.h>
#include "tokens.h"

// Flex использует макрос YY_DECL как основу объявления функции, с
// канирующей следующий токен.
// По умолчанию значение макроса YY_DECL равно
`int yylex ()`
// Но мы назовём функцию token_scan
#define YY_DECL int token_scan()

}%

%%

[ \t\r\n]+ { /* игнорируем пробелы, табы и переносы строк */ }

"a" { return ID; }
"b" { return ID; }
"c" { return ID; }
"d" { return ID; }
"e" { return ID; }
"f" { return ID; }
"g" { return ID; }
"h" { return ID; }
"i" { return ID; }
"j" { return ID; }
"k" { return ID; }
"l" { return ID; }
"m" { return ID; }
"n" { return ID; }
"o" { return ID; }
```

```
"p" { return ID; }
"q" { return ID; }
"r" { return ID; }
"s" { return ID; }
"t" { return ID; }
"u" { return ID; }
"v" { return ID; }
"w" { return ID; }
"x" { return ID; }
"y" { return ID; }
"z" { return ID; }
"A" { return ID; }
"B" { return ID; }
"C" { return ID; }
"D" { return ID; }
"E" { return ID; }
"F" { return ID; }
"G" { return ID; }
"H" { return ID; }
"I" { return ID; }
"J" { return ID; }
"K" { return ID; }
"L" { return ID; }
"M" { return ID; }
"N" { return ID; }
"O" { return ID; }
"P" { return ID; }
"Q" { return ID; }
"R" { return ID; }
"S" { return ID; }
"T" { return ID; }
"U" { return ID; }
"V" { return ID; }
"W" { return ID; }
"X" { return ID; }
"Y" { return ID; }
"Z" { return ID; }
"0" { return NUM; }
"1" { return NUM; }
"2" { return NUM; }
"3" { return NUM; }
"4" { return NUM; }
"5" { return NUM; }
"6" { return NUM; }
"7" { return NUM; }
"8" { return NUM; }
```

```
"9" { return NUM; }
 "(" { return SPECS; }
 ")" { return SPECS; }
 "[" { return SPECS; }
 "]" { return SPECS; }
 "{" { return SPECS; }
 "}" { return SPECS; }
 "." { return SPECS; }
 "," { return SPECS; }
 ":" { return SPECS; }
 ";" { return SPECS; }
 "_" { return SPECS; }
 "\"" { return SPECS; }
 "\'" { return SPECS; }
 "<" { return OP; }
 ">" { return OP; }
 "=" { return OP; }
 "+" { return OP; }
 "-" { return OP; }
 "/" { return OP; }
 "*" { return OP; }
 "|" { return OP; }
 "&" { return OP; }
 "while" { return KEYWORD; }
 "for" { return KEYWORD; }
 "do" { return KEYWORD; }
 "return" { return KEYWORD; }
 "end" { return KEYWORD; }
 "der" { return KEYWORD; }
 "var" { return KEYWORD; }
```


1.4 Парсер

Синтаксический анализатор получает на вход список токенов из лексера и генерирует АСД (Абстрактное Синтаксическое Дерево), которое позволяет структурно представить правила языка(рис.4).

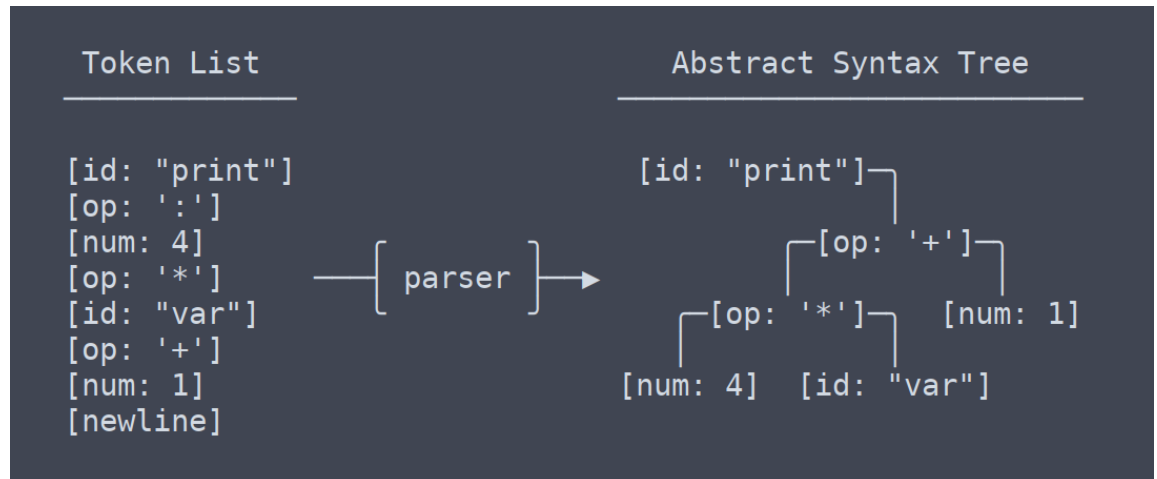


Рисунок 4 – схема работы парсера

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 C.

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, 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 в использовании 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.).

2. 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.

3. 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 1 приведены все файлы и их описание:

Таблица 1 - описание файлов

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 2 содержит описание классов в файлах. В таблице 3 описываются все функции программы, к каким классам они принадлежат.

Таблица 2 - свойства классов

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 3 - свойства функций классов

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

2.2.3 Обзор функций C++17 примененных в программе

Программа использует одно из нововведений C++17. В частности, новый тип `string_view`.

Из предыдущих стандартов для оптимизации программа использует функции `auto` и `constexpr`. Рассмотрим их подробнее:

string_view - позволяет выводить строки определенные в другом участке кода, чтобы не забивать память и не создавать лишних копий строки string.

auto – определяет тип функции по возвращаемому значению. Используется в программе для определения типов Token::Tokens::значение, где значение является нумерованным списком имен токенов.

noexcept – команда компилятору не обрабатывать исключения для функции. Используется для тех функций, которые точно не выдают исключений (например, булевы). Ускоряет компиляцию кода.

move – функция меняющая местами значения переменных вместо копирования значения и присваивания, экономит память и ускоряет работу кода.

2.3 Подробный разбор некоторых функций

2.3.1 Функция main()

Модуль, подключающий все заголовочные файлы и запускающий работу программы. Листинг кода приведён ниже. После подключения файла читаем, инициализируем массив символов code, и начинаем читать в него файл. Инициализируем функцию lex класса Lexer и передаем ему значение code. Далее запускается цикл обработки символов пока не встретится символ конца или неизвестный символ. В этом же цикле происходит вывод значений токенов.

Листинг 1. Главный модуль:

```
#include <iostream>
#include <string>
#include <locale>
#include <iomanip>
#include <fstream>
#include "tokens.hpp"
#include "lexer.hpp"
#define size 1000

using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
    ifstream in;
```

```

ofstream out;
FILE *f = fopen("program.txt", "r");//чтение файла program.txt
char code[size];
fgets(code, size, f);

out.open("token_list.txt");//создание файла token_list.txt
Lexer lex(code);//запуск функции lex класса Lexer
for (auto token = lex.next();
    not token.is_one_of(Token::Tokens::End, Token::Tokens::Unexpected);
    token = lex.next())
/*
    Определяем символ при помощи next()
    При помощи is_one_of() проверяем новый токен на символ конца
    файла и неизвестный символ
    И если таких токенов нет проверяем символы
    дальше
*/
{
    cout << setw(12) << token.tokens() << " [" << token.lexeme()
    << "]\n";
    out << setw(12) << token.tokens() << " [" << token.lexeme()
    << "]\n";
}
cout << "Записано в файл token_list.txt";
//Выводим полученный результат в файл и окно консоли
out.close();
}

```

2.3.2 Функция next()

Является главной функцией программы т.к. она передает значения токенов полученным символам. Листинг функции next() приведен ниже. Данная функция при помощи операторов case проверяет и символ и возвращает нужное имя токена.

Листинг 2. Функция next():

```

//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);
}
}

```

2.3.3 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 3. Функция is_space():

```

//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;
    }
}

```

2.3.4 Функция `is_identifier_char(char c)`

Данная функция вызывается функцией `next()` если попадаете символ из английского алфавита (вне зависимости от регистра). Возвращает `true` если символ из английского алфавита.

Листинг 4. Функция `is_identifier_char()`:

```

//Проверка на идентификатор
bool is_identifier_char(char c) noexcept
{
    switch (c)
    {
        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':
case '0':
case '1':
case '2':
case '3':
case '4':
case '5':
case '6':
case '7':
case '8':
case '9':
case '_':
    return true;
default:
    return false;
}
}
```

2.3.5 Функция is_digit()

Эта функция также вызывается функцией next() если попадаете число.
Возвращает true если символ – число.

Листинг 5. Функция is_digit():

```
//Проверка на число
bool is_digit(char c) noexcept
{
    switch (c)
    {
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            return true;
        default:
            return false;
    }
}
```

2.4 Диаграммы прецедентов и классов программы

Неотъемлемой частью создания программы является проектирование, самым популярным инструментом проектирования у разработчиков является UML. Он позволяет визуализировать элементы программы, создавать диаграммы отношений между классами и диаграммы использования программы.

2.4.1 Диаграммы прецедентов

Диаграммы прецедентов или диаграммы использования позволяют наглядно представить варианты работы с программой. Существует несколько способов использования данной программы, данные способы приведены на диаграммах прецедентов.

Диаграмма 1 (рис.5) иллюстрирует использование программы студентом:

В результате работы программы студент может получить вывод как в консоль, так и в файл.

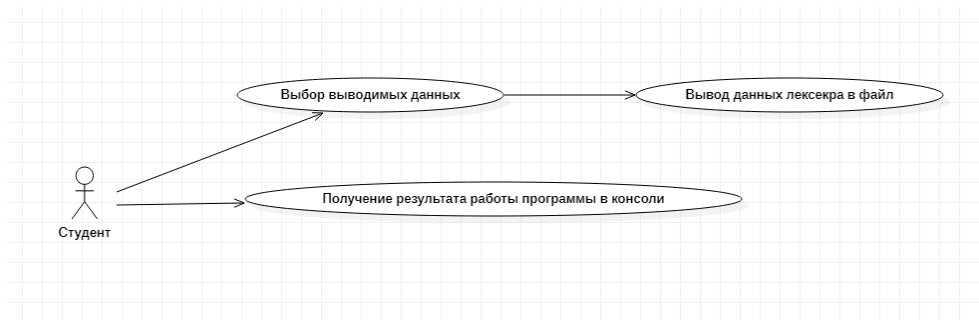


Рисунок 5 – Диаграмма 1

Диаграмма 2 (рис.6) иллюстрирует использование программы программистом:

Помимо вывода в файл программист может работать с исходным кодом программы, добавлять новые определяемые символы и работать над обработкой ошибок.

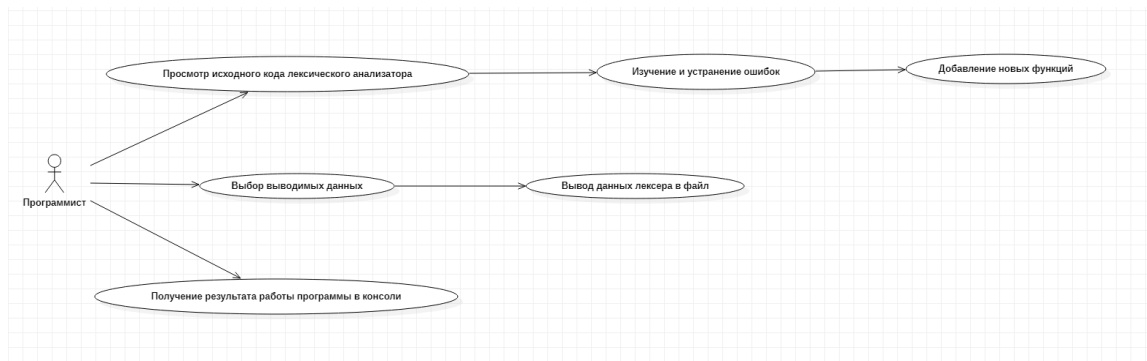


Рисунок 6 – Диаграмма 2

Диаграмма 3 (рис.7) иллюстрирует использование программы преподавателем:

Как и все он может получать вывод как в консоли, так и в файле, но при этом может просматривать исходные коды программы.

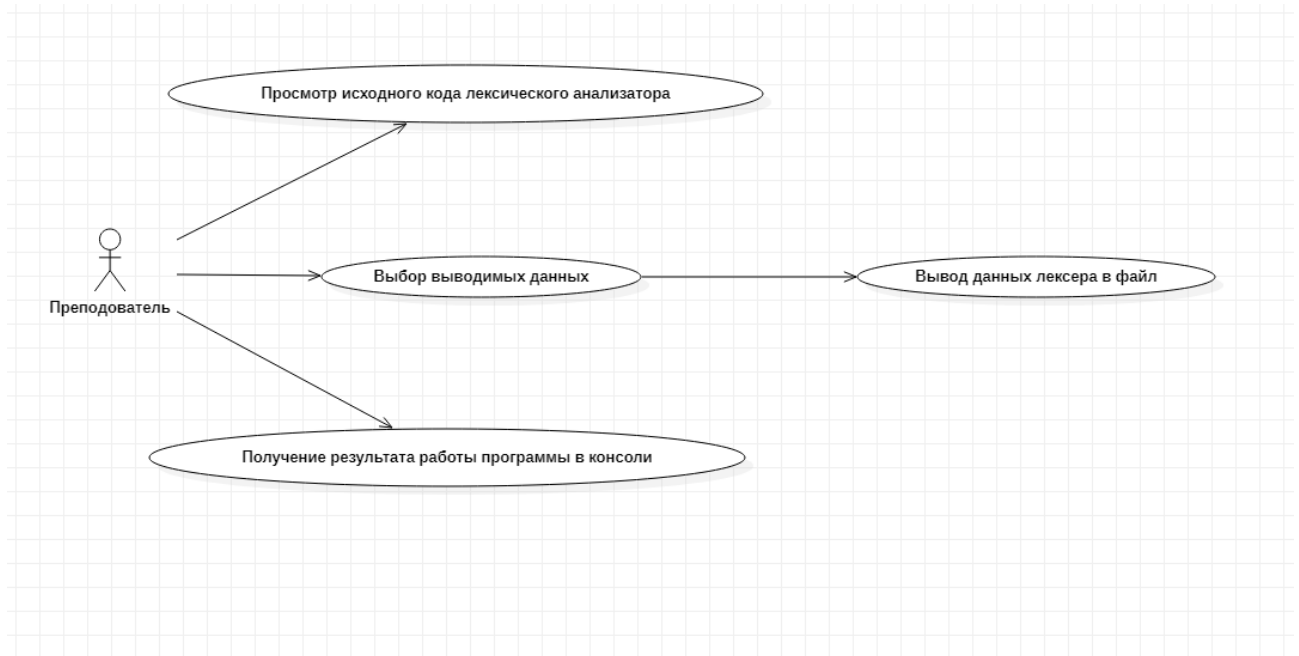


Рисунок 7 – Диаграмма 3

2.4.2 Диаграмма классов

Диаграмма классов (рис.8) наглядно показывает в каких зависимостях друг от друга находятся классы программы.

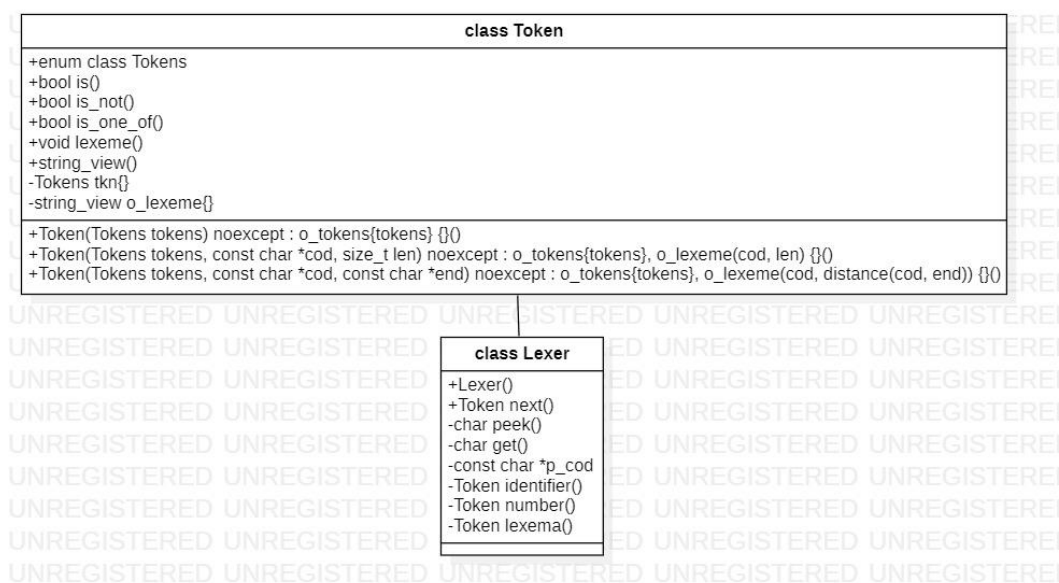


Рисунок 8 – Диаграмма классов

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

2.4.3 Сценарий работы программы

На рисунке 9 представлен сценарий работы программы:

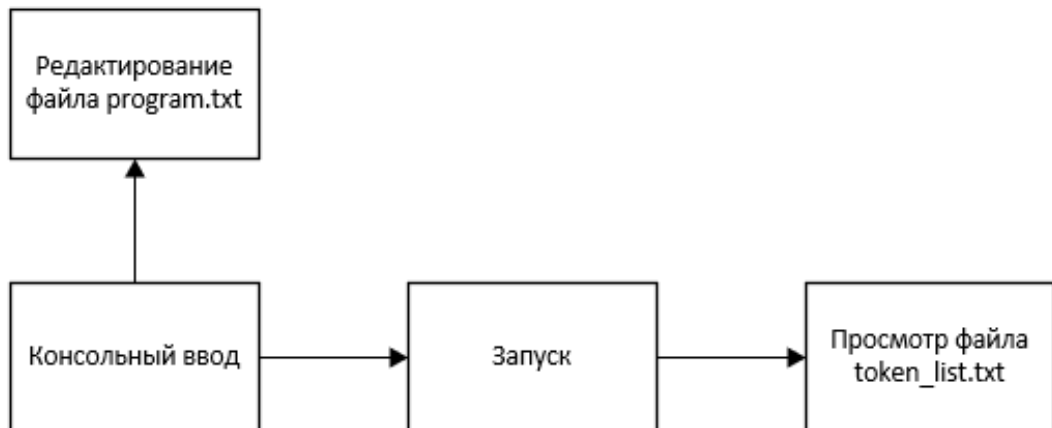


Рисунок 9 – Сценарий работы программы

3.Эксплуатационная 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 C++ или терминал Linux.

3.3 Руководство оператора

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. Проверьте находятся ли файлы в директории при помощи команды ls (рис.10). Файлы:

- lexer.cpp
- lexer.hpp
- token.cpp
- token.hpp
- main.cpp
- program.txt

```
$ ls  
lexer.cpp lexer.hpp main.cpp main.exe Makefile program.txt token_list.txt tokens.cpp tokens.hpp
```

Рисунок 10 – проверка файлов в директории

2. При помощи текстового редактора (в данном случае vim), отредактируйте файл program.txt записав программу в одну строку (рис.11).

```
chislo = 5;vtoroe_chislo = 1;while n > 0 do p = p * n;n = n - 1;end;
~
~
```

Рисунок 11 – редактирование файла

3.Сохраните и выйдите из редактора (рис.12).

```
~
~
:wq
```

Рисунок 12 – сохранение и выход из редактора

4.Запустите Makefile (рис.13):

```
$ make -f Makefile
g++ main.cpp tokens.cpp lexer.cpp tokens.hpp lexer.hpp -o main -Wall -Wextra -std=c++17
```

Рисунок 13 – запуск Makefile

5.Запустите программу:

```
$ ./main
ID [chislo]
Equal [=]
Number [5]
Semicolon [;]
ID [vtoroe_chislo]
Equal [=]
Number [1]
Semicolon [;]
ID [while]
ID [n]
GreaterThan [>]
Number [0]
ID [do]
ID [p]
Equal [=]
ID [p]
Asterisk [*]
ID [n]
Semicolon [;]
ID [n]
Equal [=]
ID [n]
Minus [-]
Number [1]
Semicolon [;]
ID [end]
Semicolon [;]
Записано в файл token_list.txt
```

Рисунок 14 – вывод программы

После запуска мы видим выведенный список токенов и символов им соответствующих (рис.14), который был параллельно записан в файл `token_list.txt`. Весь исходный код расположен в приложении.

Заключение

В ходе курсового проекта был разработан лексический анализатор текста на ПК. Была изучена работа схожих программ, а также возможности стандарта C++ 17 (оптимизация работы с памятью и ускорение работы 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.

Список литературы и интернет источников

1. Статья про создание языков программирования:
<https://tproger.ru/translations/how-to-create-programming-language/>
2. Статья про способы написания синтаксических анализаторов:
<https://habr.com/ru/post/266589/>
3. Статья про C++ 17 стандарта:
<https://ru.wikipedia.org/wiki/C%2B%2B17>
4. Статья про использование string_view:
https://ravesli.com/vvedenie-v-klass-std-string_view-v-s/

Приложение 1.Файл main.cpp

```

#include <iostream>
#include <string>
#include <locale>
#include <iomanip>
#include <fstream>
#include "tokens.hpp"
#include "lexer.hpp"
#define size 1000

using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
    ifstream in;
    ofstream out;
    FILE *f = fopen("program.txt", "r");//чтение файла program.txt
    char code[size];
    fgets(code, size, f);

    out.open("token_list.txt");//создание файла token_list.txt
    Lexer lex(code);//запуск функции lex класса Lexer
    for (auto token = lex.next();
        not token.is_one_of(Token::Tokens::End, Token::Tokens::Unexpected)
        ;
        token = lex.next())
    /*
    Определяем символ при помощи next()
    При помощи is_one_of() проверяем новый токен на символ конца файла
    и неизвестный символ
    И если таких токенов нет проверяем символы дальше
    */
    {
        cout << setw(12) << token.tokens() << " [" << token.lexeme() << "]"
        \n";
        out << setw(12) << token.tokens() << " [" << token.lexeme() << "]" \
        n";
    }
    cout << "Записано в файл token_list.txt";
    //Выводим полученный результат в файл и окно консоли
    out.close();
}

```

Приложение 2.Файл lexer.hpp

```

#ifndef LEXER_HPP
#define LEXER_HPP

#include "tokens.hpp"

class Lexer // Создаем класс лексер
{
public:
    Lexer(const char *cod) noexcept : p_cod{cod} {} //Функция Лексера
    принимающая массив символов и обрабатывающая его
    Token next() noexcept; //Прототип функции next()

private:
    //Прототипы функций
    Token identifier() noexcept;
    Token number() noexcept;
    Token lexema(Token::Tokens) noexcept;

    //Приватные конструкторы
    char peek() const noexcept { return *p_cod; }
    char get() noexcept { return *p_cod++; }

    const char *p_cod = nullptr; // Нулевой указатель
};

//Прототип перегруженного оператора
ostream &operator<<(ostream &os, const Token::Tokens &tokens);

//Прототипы функций
bool is_space(char c) noexcept;
bool is_digit(char c) noexcept;
bool is_identifier_char(char c) noexcept;

#endif

```

Приложение 3.Файл lexer.cpp

```

#include <iostream>
#include <string>
#include <locale>
#include <iomanip>
#include "tokens.hpp"
#include "lexer.hpp"

//Функция lexema() возвращающая полученный символ
Token Lexer::lexema(Token::Tokens tokens) noexcept
{
    return Token(tokens, p_cod++, 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);

```

```

    }
}

//Проверка на идентификатор и вызов функции is_identifier_char()
Token Lexer::identifier() noexcept
{
    const char *start = p_cod;
    get();
    while (is_identifier_char(peek()))
        get();
    return Token(Token::Tokens::ID, start, p_cod);
}

//Проверка на число и вызов функции is_digit()
Token Lexer::number() noexcept
{
    const char *start = p_cod;
    get();
    while (is_digit(peek()))
        get();
    return Token(Token::Tokens::Number, start, p_cod);
}

//Проверка на пробелы табулы и переходы строки
bool is_space(char c) noexcept
{
    switch (c)
    {
        case ' ':
        case '\t':
        case '\r':
        case '\n':
            return true;
        default:
            return false;
    }
}

//Проверка на число
bool is_digit(char c) noexcept
{
    switch (c)
    {
        case '0':
        case '1':
        case '2':

```

```

        case '3':
        case '4':
        case '5':
        case '6':
        case '7':
        case '8':
        case '9':
            return true;
        default:
            return false;
    }
}

//Проверка на идентификатор
bool is_identifier_char(char c) noexcept
{
    switch (c)
    {
        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':
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9':
    case '_':
        return true;
    default:
        return false;
}
}

//Перегруженный оператор вывода для работы с именами токенов
ostream &operator<<(ostream &os, const Token::Tokens &tokens)
{
    static const char *const names[]={
        "Number",

```

```

    "ID",
    "LeftParen",
    "RightParen",
    "LeftSquare",
    "RightSquare",
    "LeftCurly",
    "RightCurly",
    "LessThan",
    "GreaterThan",
    "Equal",
    "Plus",
    "Minus",
    "Asterisk",
    "Slash",
    "Dot",
    "Comma",
    "Colon",
    "Semicolon",
    "SingleQuote",
    "DoubleQuote",
    "Pipe",
    "End",
    "Sharp",
    "Dog",
    "Ampersand",
};
return os << names[static_cast<int>(tokens)];
}

```

Приложение 4.Файл tokens.cpp

```

#include <iostream>
#include <string>
#include <locale>
#include <iomanip>
#include "tokens.hpp"

using namespace std;

//Реализация функций из файла tokens.hpp

void Token::lexeme(string_view lexeme) noexcept
{
    o_lexeme = move(lexeme); //move перемещает значение переменной
    вместо копирования
}

bool Token::is(Tokens tokens) const noexcept
{
    return o_tokens == tokens;
}

bool Token::is_not(Tokens tokens) const noexcept
{
    return o_tokens != tokens;
}

bool Token::is_one_of(Tokens t1, Tokens t2) const noexcept
{
    return is(t1) || is(t2);
}

template <typename... Ts> //шаблон для функции is_one_of
bool Token::is_one_of(Tokens t1, Tokens t2, Ts... ts) const noexcept
{
    return is(t1) || is_one_of(t2, ts...);
}

string_view Token::lexeme() const noexcept
{
    return o_lexeme;
}

Token::Tokens Token::tokens() const noexcept
{

```

```
    return o_tokens;  
}
```


Приложение 5.Файл tokens.hpp

```

#ifndef TOKENS_HPP
#define TOKENS_HPP

#include <iostream>
using namespace std;

class Token //класс Токенов
{
public:
    enum class Tokens //создание вложенного класса с перечислением
        СИМВОЛЬНЫХ КОНСТАНТ
    {
        Number,          //токен Число
        ID,               //токен Идентификатор
        LeftParen,        //токен Левая скобка
        RightParen,       //токен Правая скобка
        LeftSquare,       //токен Левая квадратная скобка
        RightSquare,      //токен Правая квадратная скобка
        LeftCurly,       //токен Левая фигурная скобка
        RightCurly,      //токен Правая фигурная скобка
        LessThan,         //токен Меньше
        GreaterThan,      //токен Больше
        Equal,            //токен Равно
        Plus,             //токен Плюс
        Minus,            //токен Минус
        Asterisk,         //токен Звездочка
        Slash,            //токен Слеш
        Dot,              //токен Точка
        Comma,            //токен Запятая
        Colon,            //токен Двоеточие
        Semicolon,        //токен Точка с запятой
        SingleQuote,      //токен Одинарная кавычка
        DoubleQuote,      //токен Двойная кавычка
        Comment,          //токен Комментарий
        Pipe,             //токен Вертикальная черта
        End,              //токен Конец
        Sharp,            //токен Шарп
        Dog,              //токен Собака
        Ampersand,        //токен Амперсant
        Unexpected,       //токен Неопределенный
    };
    /*

```

Вместо блока try catch для обработки исключений говорим компилятору

что функция и конструкторы исключений не обрабатывают при промощи

```
noexcept, что ускоряет компиляцию
*/
```

//Конструкторы класса Token принимают на вход массив символов и передают

```
//на обработку методам класса Lexer
```

```
Token(Tokens tokens) noexcept : o_tokens{tokens} {}
```

```
Token(Tokens tokens, const char *cod, size_t len) noexcept : o_tokens{tokens}, o_lexeme(cod, len) {}
```

```
Token(Tokens tokens, const char *cod, const char *end) noexcept : o_tokens{tokens}, o_lexeme(cod, distance(cod, end)) {}
```

```
//Прототипы функций
```

```
Tokens tokens() const noexcept;
```

```
bool is(Tokens tokens) const noexcept;
```

```
bool is_not(Tokens tokens) const noexcept;
```

```
bool is_one_of(Tokens t1, Tokens t2) const noexcept;
```

```
template <typename... Ts> //шаблон для функции is_one_of
```

```
bool is_one_of(Tokens t1, Tokens t2, Ts... ts) const noexcept;
```

```
string_view lexeme() const noexcept;
```

```
void lexeme(string_view lexeme) noexcept;
```

```
private:
```

```
Tokens o_tokens{};
```

```
string_view o_lexeme{};
```

```
};
```

```
#endif
```