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 парсера учебного языка программирования: включаемые модули»

Выполнил студент
Феоктистова Анастасия Сергеевна
Группа П1-17
_____(Подпись)
_____ (Дата сдачи работы)

Проверил преподаватель
Гусятинер Леонид Борисович
_____(Подпись)
_____(Оценка)

Королёв 2020 г.

Оглавление

Введение 4
1. Теоретическая часть 5
1.1 Описание предметной области 5
1.1.1 Лексер 5
1.1.2 Программы для лексического анализа 6
1.1.3 Парсер 9
1.1.4 Объединяющий модуль 10
2. Проектная часть 11
2.1 Диаграмма прецедентов 11
2.2 Сценарий работы программы 13
2.3 Выбор инструментов 14
2.4 Диаграмма классов 16
2.5 Описание модулей 17
2.6 Описание спецификаций к модулям 20
2.7 Описание модуля Token и его функций 21
3. Эксплуатационная часть 24
АННОТАЦИЯ 24
3.1 Руководство пользователя 25
3.1.1 Назначение программы 25
3.1.2 Установка программы 25
3.1.2 Выполнение программы 25
Заключение 27
Список литературы и интернет-источников 28
Приложение 1. Код главного модуля main.cpp 29
Приложение 2. Код модуля Lexer lexer.hpp 30
Приложение 3. Код модуля Lexer lexer.cpp 31
Приложение 4. Код модуля Token tokens.hpp 39
Приложение 5. Код модуля Token tokens.cpp 41

Введение

Данный курсовой проект посвящён созданию парсера учебного языка программирования как универсального инструмента в IT-сфере. Парсер – это компонент будущего интерпретатора, который служит для преобразования исходного текста в поток токенов, позволяющий структурно представить правила создаваемого языка. Целью данного проекта является развитие в сфере информационных технологий, а также для получения опыта в написании будущего языка программирования.

В процессе разработки использовался GCC компилятор, что позволяет запускать программу на любой системе, где есть терминал и установлен компилятор языка C++ стандарта 17.

В первой части курсового проекта будет проанализирована предметная область, а также программы по данной теме.

Во второй части будут рассмотрены инструменты, которые использовали при создании проекта, модули, структура программной части и листинги модулей.

В третьей части будет рассмотрено взаимодействие пользователя с программой.

В заключительной части будет сделаны выводы о проекте и полученных при работе с ним знаниях.

1. Теоретическая часть

1.1 Описание предметной области

Язык программирования — это так называемый «формальный» язык, который служит для записи компьютерных программ. Язык программирования определяет набор лексических, синтаксических и сематических правил, определяющих, в первую очередь, внешний вид программы и действия, которые выполнит ЭВМ под её управлением. Полноценный язык программирования реализуется посредством написания компилятора или интерпретатора. При этом каждый компилятор или интерпретатор состоит из следующих модулей:

- Лексер (лексический анализатор)
- Парсер (синтаксический анализатор)
- Генератор или стековая машина
- Объединяющий модуль

В следующих пунктах рассмотрим каждый из них по отдельности.

1.1.1 Лексер

Лексический анализатор – это первый компонент компилятора. Роль этого компонента заключается в том, чтобы разделять текст программы на токены.

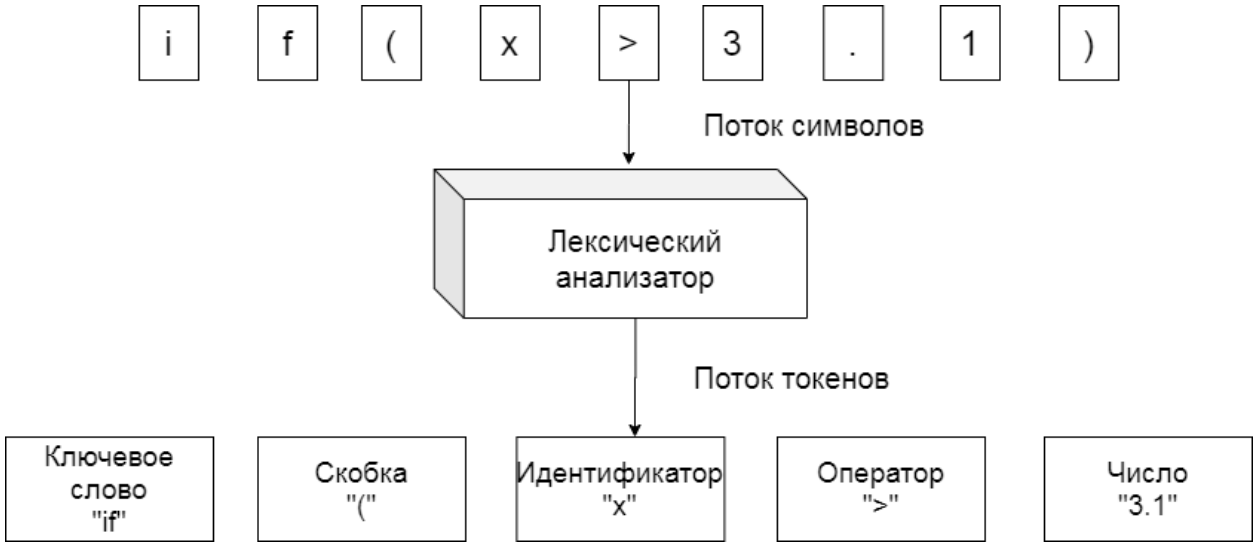

Рисунок 1. Пример работы лексера

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.

1.1.2 Программы для лексического анализа

Для упрощения работы лексера и уменьшения количества возникающих багов используют лексические генераторы, один из таких – это Flex. Принцип действия таков: он принимает на вход файл с описанием грамматики языка (Листинг 1), а потом создаёт программу на C, которая в свою очередь анализирует строки и выдаёт нужный результат.

Листинг 1. Файл с грамматикой

```
%option noyywrap

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

// Flex использует макрос YY_DECL как основу объявления функции, сканирующей 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; }
```

1.1.3 Парсер

Синтаксический анализатор преобразует исходный текст, то есть список токенов, с учетом скобок и порядка операций, в абстрактное синтаксическое дерево, которое позволяет структурно представить правила создаваемого языка.

Рисунок 2. Список токенов проходит через парсер и превращается в АСД

Также для оптимизации синтаксических анализаторов используется сторонняя библиотека – Bison, которая во многом схожа по принципу работы с Flex. А принцип её действия таков: пользовательский файл с синтаксическими правилами структурируется с помощью программы на языке C.

Bison используется для описания грамматики, построенной на базе алфавита токенов, и используется для генерации программы (кода на языке C, C++ или Java), которая получает на вход поток токенов и находит в этом потоке структурные элементы (нетерминальные токены) согласно заданной грамматике.

1.1.4 Объединяющий модуль

Одна из основных задач определить – компилируемый или интерпретируемый язык программирования. Во многом это зависит от выбранного языка-посредника, потери производительности и времени на выполнение программы.

Если язык программирования интерпретируемый, то программа выполняется построчно в режиме реального времени, для этого нужно запустить главный модуль, который в свою очередь запустит все вышеперечисленные модули и выполнит работу. Если язык компилируемый, то программа анализируется целиком и без дополнительных средств здесь не обойтись.

Одним из таких средств является LLVM IR, который написан на языке C++ и обеспечивает оптимизацию на этапе компиляции, компоновки и исполнения.

2. Проектная часть

2.1 Диаграмма прецедентов

Одной из важных частей создания курсового проекта является проектирование, а самым популярным инструментов проектирования у разработчиков – UML. Он считается языком графического описания моделирования в области разработки программного обеспечения и бизнес процессов. UML был создан для определения, визуализации, проектирования и документирования, в основном, программных систем.

Для данного курсового проекта были построены следующие диаграммы прецедентов: «Студент», «Преподаватель», «Программист». На первой диаграмме показана схема взаимодействия студента непосредственно с программой. Студент получает вывод программы в консоль и файл для дальнейшей работы с данными.

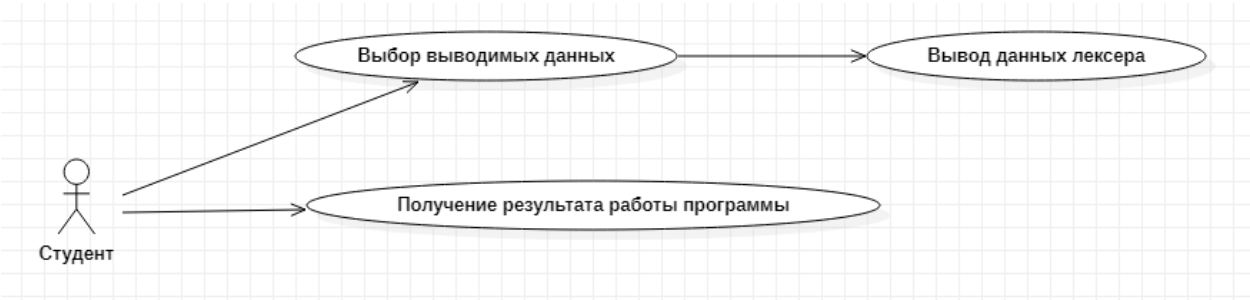

Рисунок 3. Диаграмма прецедентов для "Студент"

На второй диаграмме изображено, как происходит взаимодействие со стороны преподавателя. Одна отличительная особенность от предыдущей диаграммы – это возможность у преподавателя просмотреть исходный код интерпретатора.

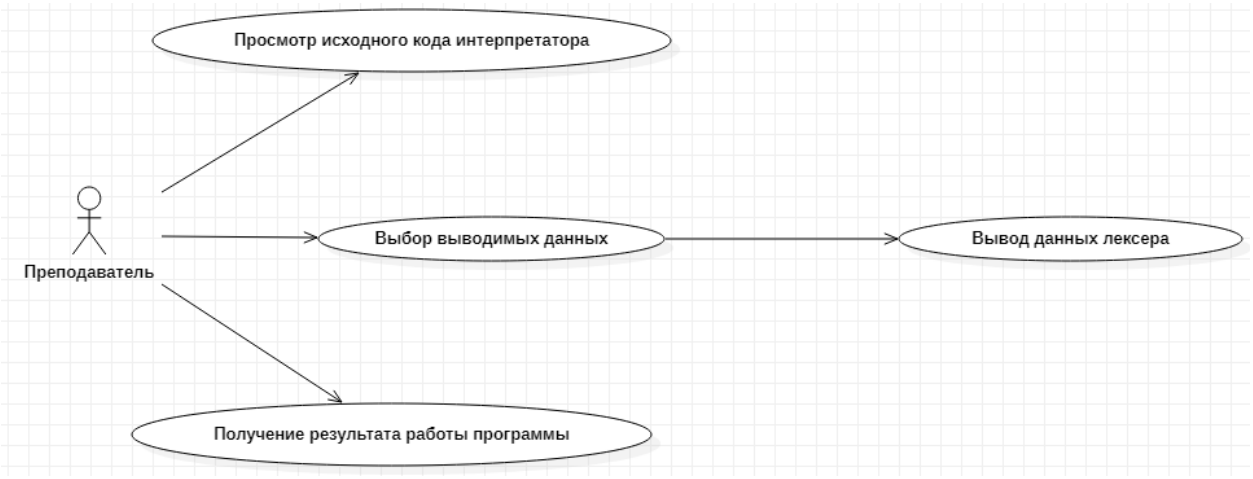

Рисунок 4. Диаграмма прецедентов для "Преподаватель"

На третьей диаграмме представлены видимые области программы для разработчика. Пожалуй, самая показательная диаграмма из всех. На ней мы можем наблюдать, что для программиста открыты все области данного проекта.

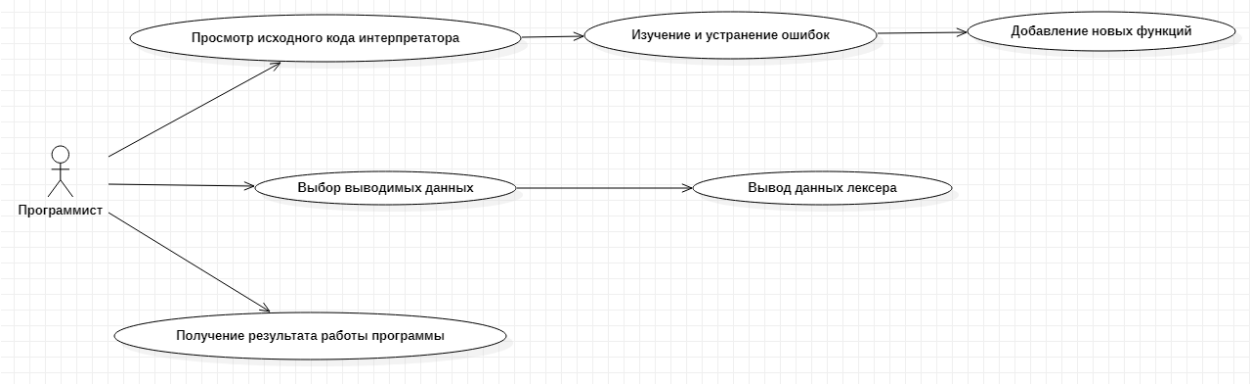

Рисунок 5. Диаграмма прецедентов для "Программист"

2.2 Сценарий работы программы

В данном разделе приведен сценарий использования программы парсер (Рисунок 6).

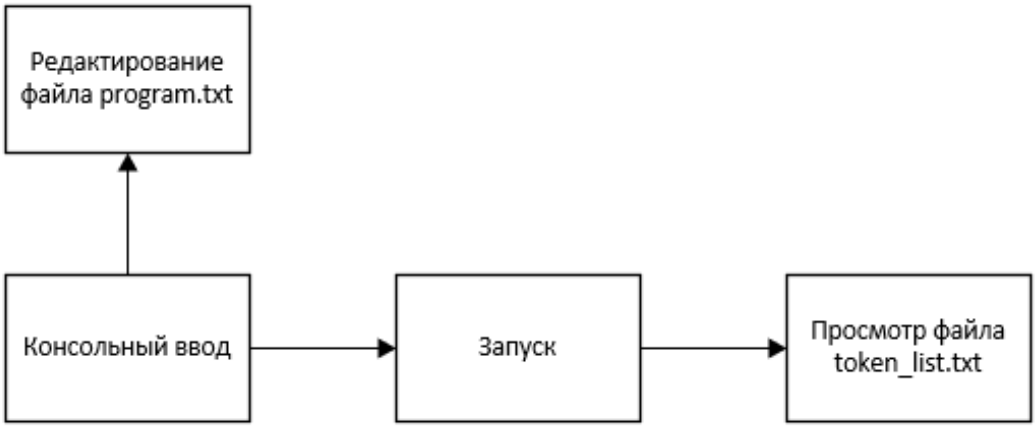

Рисунок 6. Сценарий работы программы

2.3 Выбор инструментов

При выборе инструментов обращалось внимание на следующие критерии:

- Уровень производительности
- Набор функциональных возможностей
- Время выполнения программы
- Наличие необходимых навыков
- Наличие документации на русском языке
- Актуальность
- Доступность

Исходя из этих критериев, был выбран язык программирования C++ из-за повышенной вычислительной производительности, большого набора функциональных возможностей, таких как: наличие богатой стандартной библиотеки и шаблонов, и доступности. Для C++ создано огромное количество учебной литературы, переведённой на всевозможные языки.

Данный язык программирования имеет высокий порог вхождения, обладая наиболее широкими возможностями среди всех языков такого рода. В процессе разработки также использовался GCC компилятор.

Также для работы с курсовым проектом были выбраны следующие средства:

Visual Studio Code - редактор исходного кода, разработанный Microsoft для Windows, Linux и macOS. Считается «легким» редактором кода для кроссплатформенной разработки. Имеет широкие возможности для кастомизации: пользовательские темы, сочетания клавиш и файлы конфигурации.

Cygwin - UNIX-подобная среда и интерфейс командной строки для Microsoft Windows. Обеспечивает тесную работу приложений и данных Windows с UNIX-подобной средой, позволяя запускать программы и использовать средства из одной среды в другую. Cygwin состоит из двух частей: динамически подключаемой библиотеки cygwin1.dll, которая

обеспечивает совместимость взаимодействия одной программы в другой и реализует значительную часть стандарта интерфейса между операционной системой и прикладной программой, и огромной коллекции приложений, которые обеспечивают привычную среду UNIX, включая Unix shell.

GNU C++ (g++) – бесплатный компилятор C++ от проекта GNU. Очень простой и интуитивно понятный компилятор. Установлен в Cygwin.

2.4 Диаграмма классов

В данном пункте указаны зависимости двух классов друг от друга.

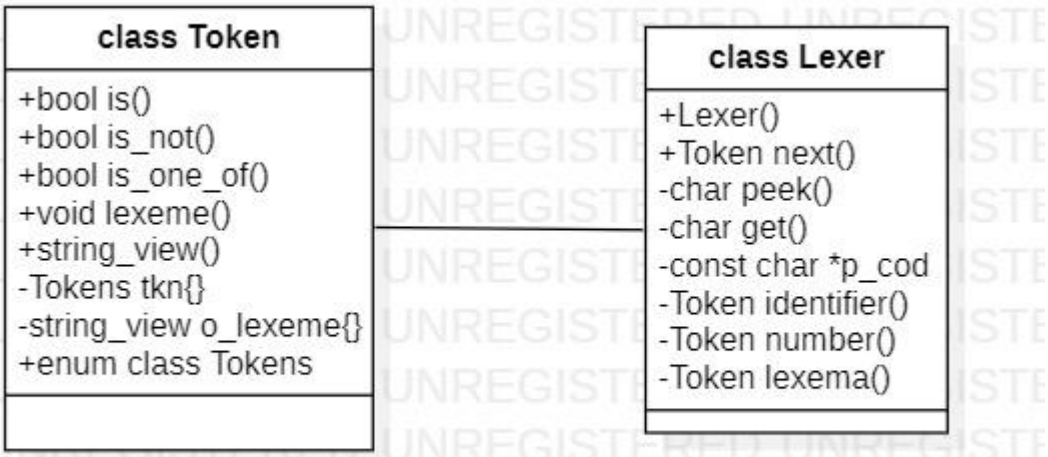

Рисунок 7. Диаграмма классов

На диаграмме показана вид связи классов – ассоциация, поскольку класс Lexer использует некоторые функции из класса Token.

Класс «Token» возвращает значение токенов, а также их проверяет, получает значение имени токена и выводит имя токена.

Класс «Lexer» сопоставляет символ с таблицей и возвращает имя токена, проверяет на идентификатор, на число, и получает на вход результат из tokens().

2.5 Описание модулей

Данное приложение содержит в себе следующие модули:

1. Модуль Token и его функции.

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 Lexer и его функции.

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.

В таблице ниже приведены все файлы и их описание:

Таблица описания 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

Таблица свойств классов содержит описание классов в файлах. В таблице свойств функций описываются все функции программы, к каким классам они принадлежат.

Таблица свойств 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 свойств функций:

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
Lexer	bool is_space(char c)	Проверка на символ
Lexer	bool is_digit(char c)	Проверка на число
Lexer	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

2.6 Описание спецификаций к модулям

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

Из предыдущих стандартов для оптимизации программа использует функции auto и noexcept. 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.

2.7 Описание модуля Token и его функций

Одним из важных модулей программы является модуль Token, реализующей проверку на правильность вводимых лексем.

В этом модуле содержится два файла tokens.cpp и tokens.hpp, один из которых реализует функции и конструкторы, а второй является заголовочным файлом, который содержит в себе класс имен токенов, прототипы функций и конструкторов.

Также в модуль Token входит набор функций для проверки на недействительность лексем.

Листинг 2. Класс Токенов.

```
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 Амперсант
        Unexpected,  //токен Неопределенный
    };
```

Здесь наглядно показано создание вложенного класса для перечисления лексем.

Листинг 3. Конструкторы класса Token.

```
//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)) {}
```

Здесь описаны конструкторы класса Token, которые принимают на вход массив символов и передают на обработку методам класса Lexer.

Листинг 4. Прототипы функций.

```
    //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;
```

В этом месте описаны прототипы функций и шаблон для функции is_one_of.

Листинг 5. Проверка токенов.

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

Первая функция возвращает True, если значение o_tokens равно значению функции tokens, т.к. функция булева и возвращает два значения True или False, функция может не обрабатывать исключения.

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

Вторая функция возвращает True, если o_tokens не равно значению функции tokens, также не обрабатывает исключения.

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

Третья функция при выводе вызывает функцию is token и передаёт туда значения токенов, возвращает True, если хотя бы один из них True. Также не обрабатывает исключения.

3. Эксплуатационная часть

АННОТАЦИЯ

В этой части приведено руководство оператора по применению и эксплуатации парсера учебного языка программирования.

В данном документе, в разделе «Назначение программы», указана информация, достаточная для понимания программы и ее эксплуатации.

В разделе «Установка программы» приведены способы, необходимые для работы программы.

В данном документе, в разделе «Выполнение программы» указана последовательность действий пользователя, обеспечивающих запуск, выполнение и завершение программы.

3.1 Руководство пользователя

3.1.1 Назначение программы

Программа представляет из себя готовый модуль лексического анализатора языка программирования. Данный модуль достаточно быстрый и не требует больших усилий в использовании. Можно использовать отдельно или в качество модуля в программах распознавания текста.

3.1.2 Установка программы

Для запуска программы не требуется установка. На компьютере должен находиться только компилятор языка C++17 или терминал Linux.

3.1.2 Выполнение программы

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. 9). 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 8. Команда ls

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

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

Рисунок 9. Редактирование файлы progam.txt

3. Сохраните и выйдите из редактора.

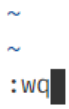

Рисунок 10. Сохранение и выход из редактора

4. Запустите Makefile:

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

Рисунок 11. 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 12. Запуск программы

После запуска мы видим выведенный список токенов и символов им соответствующих, который был параллельно записан в файл token_list.txt. Весь исходный код расположен в приложениях.

Заключение

В результате выполнения курсового проекта был разработан парсер учебного языка программирования для дальнейшего написания интерпретатора и с целью получения новых знаний в данной сфере.

В ходе работы была проанализирована предметная область и получены практические навыки по созданию лексического и синтаксического анализаторов.

Области применения:

- Модуль компиляторов и интерпретаторов
- Модуль в программах распознавания текста

Также планируется продолжить работу над данным проектом с целью расширения возможностей программы для пользователей.

Список литературы и интернет-источников

1. Написание языка программирования: теория
https://habr.com/ru/post/316460/
2. Как реализовать язык программирования: парсер
https://habr.com/ru/post/443630/
3. Как создать свой язык программирования: теория, инструменты и советы от практика
https://tproger.ru/translations/how-to-create-programming-language/
4. Создаем свой язык: пособие с примерами
https://proglib.io/p/your-own-programming-language
5. Парсинг: Что? Зачем? Как?
http://parser.valemak.com/info-what-why-how
6. Parser. Язык веб-программирования от "Студии Артемия Лебедева"
https://www.igromania.ru/article/3365/Parser_Yazyk_veb-programmirovaniya_ot_Studii_Artemiya_Lebedeva.html

Приложение 1. Код главного модуля 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
}
```

Приложение 2. Код модуля Lexer 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. Код модуля Lexer 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 таюы 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. Код модуля Token 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 Амперсант
        Unexpected,  //токен Неопределенный
    };
```

```
    /*
    Вместо блока try catch для обработки исключений говорим компилятору,
    что функция и конструкторы исключений не обрабатывают при прмощи
    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
```

Приложение 5. Код модуля Token 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;
}
```