**Российская федерация  
Ханты-Мансийский автономный округ – Югра  
Департамент образования и науки  
Сургутский государственный университет ХМАО**

Политехнический институт  
Кафедра Автоматики и компьютерных систем

**Пояснительная записка**  
к курсовому проекту   
по дисциплине “Объектно-ориентированное программирование”

Выполнил: студент группы 609-31

Гаврилов Е.Е.

Проверил: старший преподаватель

кафедры автоматики и компьютерных систем

Карамышев Э.Р.

Сургут

2025 г.

**Задание**

Реализовать лексер, парсер и драйвер для интерпретатора C-подобного языка программирования.

**Аннотация**

В данной пояснительной записке описаны этапы написания курсового проекта по дисциплине «объектно-ориентированное программирование» в процессе которого будет реализованы модули для лексического и синтаксического анализа.  
Записка содержит в себе этапы анализа изучаемой темы, проектирования системы классов и алгоритмов и описание процесса разработки конечного программного продукта.

Оглавление

[1. Введение 6](#_Toc188213690)

[2. Анализ 7](#_Toc188213691)

[2.1 Анализ предметной области данной задачи 7](#_Toc188213692)

[2.2 Обзор существующих решений 8](#_Toc188213693)

[2.2.2 GCC 9](#_Toc188213694)

[2.2.3 Clang 9](#_Toc188213695)

[2.3. Требования к разработанной программе 9](#_Toc188213696)

[3.Проектирование 10](#_Toc188213697)

[3.1 Формальное описание задачи 10](#_Toc188213698)

[3.1.1 Описание языка 10](#_Toc188213699)

[3.1.2 Интерпретатор 11](#_Toc188213700)

[3.2 Поведенческая модель программного продукта 12](#_Toc188213701)

[3.3 Структура ПО 12](#_Toc188213702)

[3.4 Файловая структура проекта 13](#_Toc188213703)

[4.Кодирование 15](#_Toc188213704)

[4.1 Определение языка и вспомогательных средств 15](#_Toc188213705)

[4.2 Генерация класса-парсера (прил. 2) 15](#_Toc188213706)

[4.3 Генерация класса-лексера 17](#_Toc188213707)

[4.4 Класс-драйвер 19](#_Toc188213708)

[5. Тестирование 23](#_Toc188213709)

[5.1 Планирование тестирования 23](#_Toc188213710)

[5.2 Разработка тестов 23](#_Toc188213711)

[5.3 Тестирование ПО 23](#_Toc188213712)

[5.4 Итоги тестирования 27](#_Toc188213713)

[6. Сопровождение 28](#_Toc188213714)

[6.1 Требования к программно-аппаратно платформе 28](#_Toc188213715)

[6.2 Инструкция по установке и использованию 28](#_Toc188213717)

[7. Заключение 29](#_Toc188213721)

[Список литературы 30](#_Toc188213722)

[**Приложения** 31](#_Toc188213723)

[Приложение 1. Формальное описание грамматики 31](#_Toc188213724)

[Приложение 2. Листинг grammar.y 32](#_Toc188213725)

[Приложение 3. Листинг grammar.l 37](#_Toc188213726)

[Приложение 4. Листинг driver.hh 39](#_Toc188213727)

[Приложение 5. Листинг driver.cc 39](#_Toc188213728)

[Приложение 6. Листинг main.cpp 40](#_Toc188213729)

1. Введение

Интерпретатор (от лат. interpretator — толкователь) — программа (разновидность транслятора), выполняющая интерпретацию.

Интерпретация — построчный анализ, обработка и выполнение исходного кода программы или запроса, в отличие от компиляции, где весь текст программы перед запуском анализируется и транслируется в машинный или байт-код без её выполнения [1].

Первым интерпретируемым языком программирования высокого уровня был Lisp. Его интерпретатор был создан в 1958 году Стивом Расселом на компьютере IBM 704. Рассел вдохновился работой Джона Маккарти и выяснил, что функция eval в Lisp может быть встроена в машинный код [2].

2. Анализ

2.1 Анализ предметной области данной задачи

Для выполнения кода на каком-либо языке программирования, необходимо сначала определить правила этого языка.

Формальная грамматика (далее – просто грамматика) – способ описания формального языка (в том числе и языков программирования), то есть выделения некоторого подмножества из множества всех слов некоторого конечного алфавита. Различают порождающие и распознающие (или аналитические) грамматики — первые задают правила, с помощью которых можно построить любое слово языка, а вторые позволяют по данному слову определить, входит ли оно в язык или нет.

Терминал (терминальный символ) — объект, непосредственно присутствующий в словах языка, соответствующего грамматике, и имеющий конкретное, неизменяемое значение (обобщение понятия «буквы»). В формальных языках, используемых на компьютере, в качестве терминалов обычно берут все или часть стандартных символов ASCII — латинские буквы, цифры и спецсимволы.

Нетерминал (нетерминальный символ) — объект, обозначающий какую-либо сущность языка (например, формула, арифметическое выражение, команда) и не имеющий конкретного символьного значения.

По иерархии Хомского, грамматики делятся на 4 типа, каждый последующий является более ограниченным подмножеством предыдущего (но и легче поддающимся анализу):

тип 0. неограниченные грамматики — возможны любые правила

тип 1. контекстно-зависимые грамматики — левая часть может содержать один нетерминал, окруженный «контекстом» (последовательности символов, в том же виде присутствующие в правой части); сам нетерминал заменяется непустой последовательностью символов в правой части.

тип 2. контекстно-свободные грамматики — левая часть состоит из одного нетерминала.

тип 3. регулярные грамматики — более простые, эквивалентны конечным автоматам [3].  
Интерпретацию можно разбить на следующие этапы:

1. Лексический анализ.
2. Синтаксический анализ.
3. Выполнение.

Кроме этого в некоторых интерпретаторах можно выделить этапы препроцессинга и оптимизации, однако в данной работе они не будут рассмотрены.

Лексический анализ — процесс разбора входной последовательности символов на распознанные группы — лексемы — с целью получения на выходе идентифицированных последовательностей, называемых «токенами» (подобно группировке букв в словах).

Синтаксический анализ — процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево). Обычно применяется совместно с лексическим анализом.

Синтаксический анализатор — это программа или часть программы, выполняющая синтаксический анализ.

В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки.

Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное либо в виде дерева зависимостей, либо в виде дерева составляющих, либо в виде некоторого сочетания первого и второго способов представления.

2.2 Обзор существующих решений

За все время было создано огромное количество языков программирования, как компилируемых, так и интерпретируемых и для каждого языка существует как минимум один компилятор или интерпретатор. Далее рассмотрены некоторые из них:

2.2.1 CPython

CPython – наиболее распространенный интерпретатор языка Python. байт-кода, написан на C. Разработка ведётся группой разработчиков под руководством создателя Python Гвидо ван Россума. CPython является программным обеспечением с открытым исходным кодом.

2.2.2 GCC

GCC (GNU Compiler Collection): Один из самых популярных компиляторов для языков, подобных C. GCC поддерживает множество языков, включая C, C++, Fortran и другие, и является мощным инструментом для создания и оптимизации программ.

2.2.3 Clang

Clang: Современный компилятор с открытым исходным кодом, использующий инфраструктуру LLVM. Он активно используется для разработки языков, близких к C, и обладает высокой производительностью и поддержкой отладки.

2.3. Требования к разработанной программе

Пользовательский инетрфейс  
Реализовать консольный интерфейс, который позволит пользователю запустить выполнение кода программы из файла.

Программа должна принимать текстовый файл и корректно выполнять записанный в нем код. Программа должна проверять корректность исходного кода, в случаи ошибки указывать строку и символ ошибки.

Лексический анализ  
Программа, должна принимать на вход текст и превращать его в последовательность токенов.

Синтаксический анализ.

Программа должна строить синтаксическое дерево согласно полученному потоку токенов.

3.Проектирование

3.1 Формальное описание задачи

3.1.1 Описание языка

Разработанный язык (далее – просто язык) по синтаксису схож с языком Си. Он поддерживает:

Целочисленный тип данных int;

Объявление переменных:  
[имя переменной] = [значение];

Арифметические операторы:  
сложения (+),  
вычитания и унарного минуса(-),  
умножения (\*),  
деления (/),  
получения остатка от деления (%)

Логические операторы:  
или (&&),  
и (||),  
не (!)

Операторы сравнения:  
меньше (<),  
больше (>),  
меньше или равно (<=),  
больше или равно (>=),  
равно (==),  
не равно (!=)

Условный оператор if-else, его полную форму (if ( условие ) { действия } else {действия}) и неполную (if (условие) {действия} форму);

Цикл while: while (условие) { действия }

Оператор вывода значения в консоль print:  
print [значение];

Оператор ввода значения переменной из консоли input:  
input [имя переменной];

Комментарии:  
Однострочные вида // [текст комментария]   
Многострочные вида /\* [текст комментария] \*/

Все символы-разделители (пробелы, символы табуляции, перевода строки, возврата каретки и пр.) игнорируются.

Формальная грамматика языка описана в приложении 1.

3.1.2 Интерпретатор

Программа получает на вход файл с кодом, написанном на выше обозначенном языке. Полученный текст разбивается на список токенов, таких как “IF”, “Открывающая скобка”, “Целое число” и т.д. В случае обнаружения символа, не входящего в алфавит языка, программа прекращает работу и сообщает об ошибке. После программа проводит синтаксический анализ последовательности полученных ранее токенов. Согласно грамматике, составляется абстрактное синтаксическое дерево[5], узлами которого являются операторы, циклы, условия и прочие конструкции языка.  
На рисунке 1 пример синтаксического дерева, построенного для программы  
a = 0;  
if (a == 0) { print 1; }  
else { print 0; }

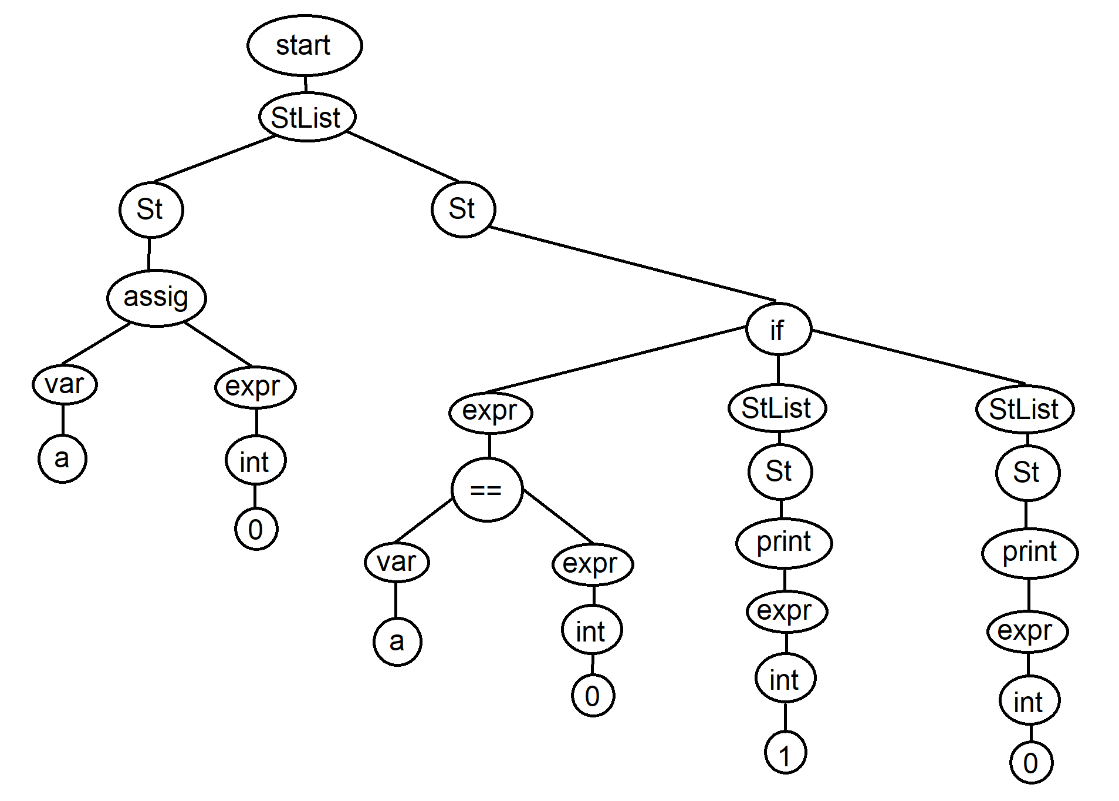


Рис 3.1. Пример построенного абстрактного синтаксического дерева.

По завершению построения происходит свертка дерева и выполнение указанных действий.  
В случае синтаксической ошибки (нарушение ожидаемой формы или структуры языка) программа останавливает анализ и сообщает об ошибке.

3.2 Поведенческая модель программного продукта

Единственный способ взаимодействия с программой – вызов ее через консоль с одним параметром – именем файла с исходным кодом.  
После выполнения, программа построчно выводит значения, после команды print, или сообщение об ошибке и ее тип.

3.3 Структура ПО

Программа делится на следующие модули: лексер, парсер, абстрактное синтаксическое дерево и драйвер.

Модуль лексера состоит из класса FlexLexer, который генерирует flexer из исходного файла с грамматикой (с расширением .l).

Модуль парсера состоит из класса parser, который генерирует bison из исходного файла с грамматикой (с расширением .y). Класс содержит метод parser.parse(), непосредственно запускающий разбор.

Модуль абстрактного синтаксического дерева состоит из класса-интерфейса INode и его дочерних классов, представляющих собой узлы дерева – элементы языка, например Print\_Node, If\_Node, Var\_Node и пр. Другим модулям программы должен быть доступен только класс-интерфейс, реализация классов-наследников должна быть скрыта.

Модуль драйвера состоит из класса driver, содержащий указатель на лексер и метод для запуска нисходящего разбора.

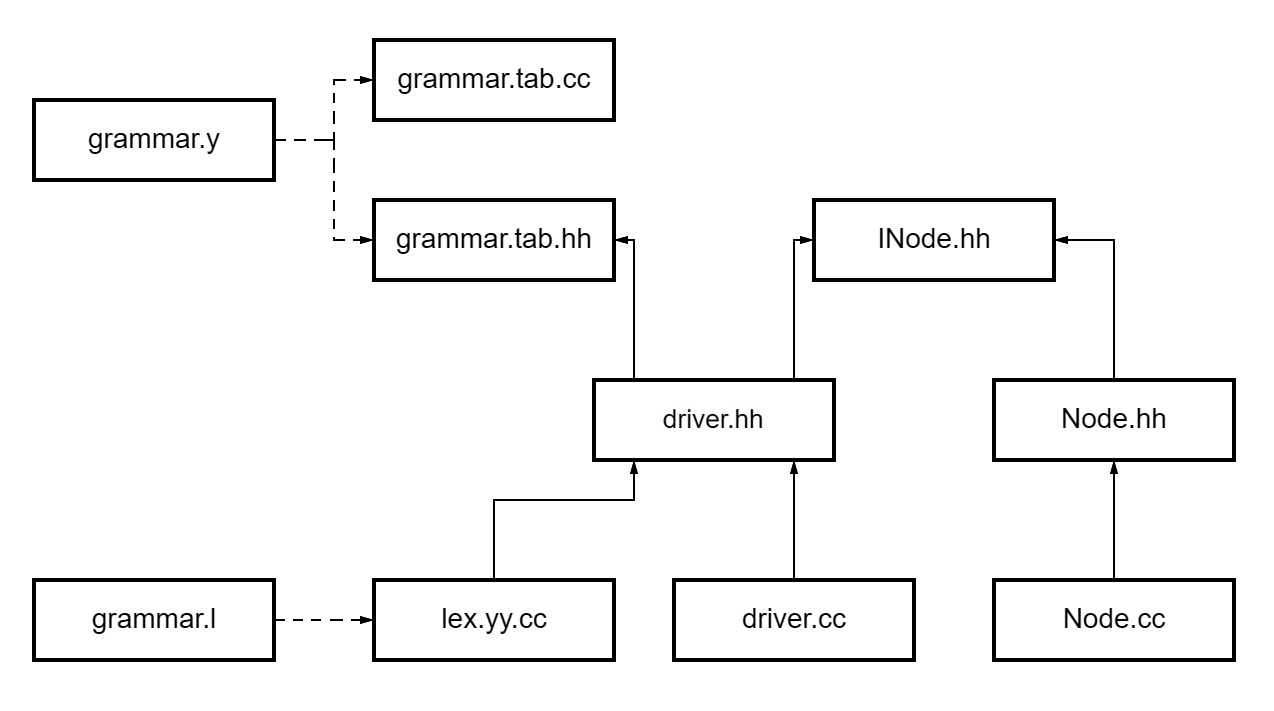


Рис 3.2. Структура ПО

3.4 Файловая структура проекта

Файловая структура проекта выглядит следующим образом:

│ main.cpp

│ Makefile

│

├───AST

│ INode.hh

│ Node.cc

│ Node.hh

│

├───driver

│ driver.cc

│ driver.hh

│

└───grammar

grammar.l

grammar.y

Где в директории AST хранятся файлы с определением и реализацией классов для непосредственной работы с синтаксическим деревом; в директории driver – определение и реализация классов драйвера; в директории grammar – исходные файлы грамматик для flex и bison.

4.Кодирование

4.1 Определение языка и вспомогательных средств

Программа реализована на языке C++ стандарта C++17, с использованием стандартных библиотек языка, а так же инструментов Fast Lexical Analyzer[4] и GNU Bison [6]

4.2 Генерация класса-парсера (прил. 2)

GNU Bison – генератор синтаксических анализаторов. Обычно используется в комплексе с лексическим анализатором flex.  
Flex используется для описания базовых токенов и создания программы (кода на C или C++), перерабатывающей поток символов в поток токенов. 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.

Для настройки Bison использованы следующие параметры:

%language "c++"

%require "3.0"

%skeleton "lalr1.cc"

%param { yy::NumDriver \* driver }

%define parse.lac full

%define api.value.type variant

%define parse.error verbose

language "c++" – параметр, обеспечивающий то, что сгенерированный парсер будет совместим с языком C++ (по умолчанию совместим только с C).

require "3.0" – обеспечивает использование достаточно новой версии Bison.

skeleton "lalr1.cc" – выбор алгоритма синтаксического разбора для парсера. В данном случае выбран lalr1 разбор – восходящий алгоритм синтаксического разбора. [7][8]

param { yy::NumDriver \* driver } – определение параметров для конструктора класса - парсера.

define parse.lac full – включение предупреждающего исправления – для улучшения обработки синтаксических ошибок.

define api.value.type variant – директива, для настройки хранения типов семантических значений, которая позволит хранить любые типы объектов C++.

define parse.error verbose – директива управления генерации сообщений об ошибках.

Далее описан код, предшествующий правилам грамматики:

%code requires{

#include "../AST/INode.hh"

namespace yy { class NumDriver; }

}

%code{

#include "../driver/driver.hh"

namespace yy {

void yyerror(const char \*);

parser::token\_type yylex(parser::semantic\_type\* yylval, NumDriver\* driver);

}

}

Здесь описаны директивы include и декларации функций, которые будут использовать далее.

В поле %token описаны названия токенов, которые будут в последствии получены от лексера. Токены являются целыми числами.

В полях %nterm записаны ожидаемые нетерминальные выражения и их тип. Полей несколько, так как у нетерминальных выражений разные приоритет, тот терминал, который записан ниже считается более приоритетным.

Поля %right %left описывают ассоциативность и так же как и %nterm, приоритет. Так, например, умножение приоритетнее сложения, и оба оператора левоассоциативны, поэтому

%left ADD SUB;

%left MUL DIV MOD;

Токены типов данных записаны отдельно, так как кроме номера, хранят значения:

%token <int> INT

%token <std::string> NAME

Далее описаны правила грамматики: слева пишется правило грамматики, справа код на C++, который будет выполняться во время свертки правил. $$ используется для хранения результата правила. Каждое такое правило будет вызывать функцию из AST, возвращающую указатель на объект-узел дерева INode.

После описания правил грамматики записана реализация ранее объявленных функций:

namespace yy {

parser::token\_type yylex(parser::semantic\_type\* yylval,

NumDriver\* driver)

{

return driver->yylex(yylval);

}

void parser::error(const std::string& mess){

std::cout<<mess<<" in line "<<driver->plex\_->lineno()<<std::endl;

}

}

Метод yylex возвращает тип обрабатываемого токена.

Метод error будет вызван в случае ошибки синтаксического анализа, выведет сообщение об ошибке, а также строку на которой обнаружена ошибка. Класс parser определен в пространстве имен yy.

4.3 Генерация класса-лексера

Flex (Fast Lexical Analyzer) — генератор лексических анализаторов. Заменяет Lex в системах на базе пакетов GNU и имеет аналогичную функциональность. [4]. На входе программа получает текст в свободном формате и правила выделения лексем, а на выходе даёт код анализатора, в виде функции на языке С или класса на языке C++;

Правила задаются в виде регулярных выражений слева и, обычно, кода на языке C справа. Они содержат три секции, отделяющиеся строкой «%%». Определения содержат стартовые значения и определения, правила, непосредственно сами выражения и соответствующие им действия; пользовательский код включается в вывод flex. Функция класса анализатора получает текст на входе и выполняет заданный код для каждой найденной лексемы.   
Для настройки FlexLexer используются следующие параметры: [4]

%option c++ - обеспечение того, что сгенерированный код будет совместим с C++=.

%option noyywrap – метод wrap() используется Flex для поддержки языков с препроцессором. В разрабатываемом языке нет препроцессора, опция отключена.

%option yylineno – позволяет сохранять номер текущей строки внутри поля класса. Будет использовано для отслеживания строки при обнаружении синтаксической ошибки в методе void parser::error.

Далее с помощью регулярных выражений определяются правила для Flex: пустое пространство, целое число, комментарий, имя.

WS [ \n\t\v\r]+

NUM10 [1-9][0-9]\*|0

NUM {NUM10}

COMMENT \/\\*(.|\n)\*\\*\/|\/\/.\*\n

NAME [A-Za-z\_][A-Za-z0-9\_]\*

Для каждого найденного правила или символа указан код, который нужно выполнить при обнаружении этого правила:

%%

{WS}|{COMMENT} /\*do nothing\*/

"+" {return yy::parser::token\_kind\_type::ADD;}

"-" {return yy::parser::token\_kind\_type::SUB;}

"\*" {return yy::parser::token\_kind\_type::MUL;}

"/" {return yy::parser::token\_kind\_type::DIV;}

"%" {return yy::parser::token\_kind\_type::MOD;}

"<" {return yy::parser::token\_kind\_type::LESS;}

">" {return yy::parser::token\_kind\_type::GREATER;}

"<=" {return yy::parser::token\_kind\_type::LS\_EQ;}

">=" {return yy::parser::token\_kind\_type::GR\_EQ;}

"==" {return yy::parser::token\_kind\_type::EQ;}

"!=" {return yy::parser::token\_kind\_type::NEQ;}

"||" {return yy::parser::token\_kind\_type::OR;}

"&&" {return yy::parser::token\_kind\_type::AND;}

"!" {return yy::parser::token\_kind\_type::NOT;}

"=" {return yy::parser::token\_kind\_type::ASSIGN;}

";" {return yy::parser::token\_kind\_type::SCL;}

"(" {return yy::parser::token\_kind\_type::LB;}

")" {return yy::parser::token\_kind\_type::RB;}

"{" {return yy::parser::token\_kind\_type::LCB;}

"}" {return yy::parser::token\_kind\_type::RCB;}

"if" {return yy::parser::token\_kind\_type::IF;}

"else" {return yy::parser::token\_kind\_type::ELSE;}

"while" {return yy::parser::token\_kind\_type::WHILE;}

"print" {return yy::parser::token\_kind\_type::PRINT;}

"input" {return yy::parser::token\_kind\_type::INPUT;}

{NUM} {return yy::parser::token\_kind\_type::INT; }

{NAME} {return yy::parser::token\_kind\_type::NAME; }

. {return yy::parser::token\_kind\_type::ERR;}

%%

При обнаружении комментария или пустого пространства, ничего не происходит, при обнаружении любого допустимого символа, последовательности символов или правила, метод yylex() класса FlexLexer вернет соответствующий токен. Токены были определены в грамматике для Bison.

4.4 Класс-драйвер

Класс драйвера определен в пространстве имен yy. Единственным его полем является указатель на класс-лексер. Единственный конструктор класса – с параметром- указателем на объект класса лексер. Класс-парсер является дружественным классом, так как для отслеживания ошибок парсеру потребуется лексер, указатель но который является полем драйвера.

namespace yy

{

class Driver

{

friend class parser;

FlexLexer \*plex\_;

public:

Driver(FlexLexer \*plex) : plex\_(plex) {};

parser::token\_type yylex(parser::semantic\_type \*yylval);

bool parse();

void switch\_streams(std::istream &new\_in, std::ostream &new\_out);

};

}

Метод void Driver::switch\_streams(std::istream &new\_in, std::ostream &new\_out) позволяет изменить потоки ввода – вывода. Он вызывает метод изменения потоков у лексера по указателю.

void Driver::switch\_streams(std::istream &new\_in, std::ostream &new\_out)

{

plex\_->switch\_streams(new\_in, new\_out);

}

Метод parser::token\_type Driver::yylex(parser::semantic\_type \*yylval) вызывает метод yylex() лексера, который возвращает тип токена. Если тип является именем переменной или целым числом, то глобальной переменной yylval, создаваемой flex-ом, присваивается это значение.

parser::token\_type Driver::yylex(parser::semantic\_type \*yylval)

{

parser::token\_type tt = static\_cast<parser::token\_type>(plex\_->yylex());

if (tt == parser::token\_type::INT)

yylval->emplace<int>(std::stoi(plex\_->YYText()));

if (tt == parser::token\_type::NAME)

yylval->emplace<std::string>(std::string{plex\_->YYText()});

return tt;

}

Метод bool Driver::parce() запускает механизм парсинга и возвращает логическое значение: истину если разбор произошел без ошибок и ложь в случае ошибки.

bool Driver::parse()

{

parser parser(this);

bool res = parser.parse();

return !res;

}

4.5 Основная функция

Основная функция ожидает, что при запуске команды из консоли последует ровно 1 аргумент – имя файла с исходным кодом.

В основной функции создан корень дерева root из модуля синтаксического дерева, объект-лексер lexer и объект-драйвер driver. В случае некорректного количества аргументов командной строки программам выведет соответствующее сообщение и завершит работу.

В ином случае, будет открыт поток указанного файла. Если файл не существует или не может быть открыт по любой другой причине, программа выведет соответствующее сообщение и завершит работу.

В случае успешного открытия, драйвер меняет потоки на поток файла и стандартный поток вывода. В try-блоке вызывается метод парсинга и сворачивание дерева. В случае ошибки она будет выведена в консоль.

int main(int argc, char \*\*argv)

{

auto root = AST::make\_scope();

CURRENT\_SCOPE = root.get();

std::ifstream in;

FlexLexer \*lexer = new yyFlexLexer;

yy::Driver driver(lexer);

if (argc == 1)

std::cout << "File name is required"<< std::endl<<"zmm [file name]" << std::endl;

else if (argc == 2)

{

in.open(argv[1]);

if (in.is\_open())

{

driver.switch\_streams(in, std::cout);

try

{

if (!driver.parse())

return 1;

root->calculate();

}

catch (std::runtime\_error &err)

{

std::cerr << err.what() << std::endl;

}

}

else

std::cout << argv[1] << ": file does not exist" << std::endl;

}

else

std::cout << "To many arguments"<< std::endl<<"zmm [file name]" << std::endl;

delete lexer;

return 0;

}

5. Тестирование

5.1 Планирование тестирования

Тестирование должно включать в себя:

Проверки работоспособности программы в целом;   
Проверки корректной работы программы для всех конструкций языка: создания переменных, вычисления математических выражений ( в том числе корректность приоритета операций), операторов ввода и вывода, циклов и ветвлений, и комментариев;   
Проверки обработки ошибок;

5.2 Разработка тестов

Для тестов будут подготовлены несколько входных файлов с разным содержимым для проверки корректной работы программы.

5.3 Тестирование ПО

Содержимое тестового файла, ожидаемы и реальны вывод представлены в таблице 5.3

|  |  |  |
| --- | --- | --- |
| **Содержимое тестового файла** | **Ожидаемый вывод** | **Реальный вывод** |
| print 1234; | 1234 | 1234 |
| print 1234 | syntax error, unexpected end of file in line 1 | syntax error, unexpected end of file in line 1 |
| variable = 321;  print variable; | 321 | 321 |
| variable = 0;  variable = 1 + 1;  print variable; | 2 | 2 |
| variable = 0;  variable = 1 + 2 \* 3;  print variable; | 7 | 7 |
| variable = 0;  variable = -3;  print variable; | -3 | -3 |
| variable = 0;  variable = --3;  print variable; | 3 | 3 |
| variable = 0;  variable = 3 --- 3;  print variable; | 0 | 0 |
| variable = 0;  variable = 1/0  print variable; | Divide by zero! | Divide by zero! |
| variable = 0;  variable = 123/20;  print variable; | 6 | 6 |
| variable = 0;  variable = 1%0;  print variable; | Divide by zero! | Divide by zero! |
| variable = 0;  variable = 123%20;  print variable; | 3 | 3 |
| print 1 < 0;  print 1 < 1;  print 1 < 2;  print 1 > 0;  print 1 > 1;  print 1 > 2; | 0  0  1  1  0  0 | 0  0  1  1  0  0 |
| print 1 <= 0;  print 1 <= 1;  print 1 <= 2;  print 1 >= 0;  print 1 >= 1;  print 1 >= 2; | 1  1  0  1  1  0 | 1  1  0  1  1  0 |
| print 1 == 0;  print 1 == 1;  print 12345 == 12345;  print 1 != 0;  print 1 != 1;  print 12345 != 12345; | 0  1  1  1  0  0 | 0  1  1  1  0  0 |
| print 1 && 0;  print 1 || 0;  print !1;  print !11111;  print !0; | 0  1  0  0  1 | 0  1  0  0  1 |
| var = 0;  print var;  input var;  print var;  ввод: 4 | 0  4 | 0  4 |
| var = 0;  print var;  input var;  print var;  ввод: 11111111111111111 | 0  Invalid value int stdin! | 0  Invalid value int stdin! |
| var = 1111111111111111111111111;  print var ; | Int out of range in line: 1 | Int out of range in line: 1 |
| i = 0;  while (i<10){  print i;  i = i+1;  } | 0  1  2  3  4  5  6  7  8  9 | 0  1  2  3  4  5  6  7  8  9 |
| while ( ) { } | syntax error, unexpected RB in line 1 | syntax error, unexpected RB in line 1 |
| if ( 1 + 2 > 0) print (111);  else print (0); | 111 | 111 |
| if ( 1 + 2 > 0) { print (111); }  else print (0); | 111 | 111 |
| if ( 1 + 2 > 0) print (111);  else { print (0); } | 111 | 111 |
| if ( 1 + 2 > 0) { print (111); }  else { print (0); } | 111 | 111 |
| if ( 1 + 2 > 0) print (111); | 111 | 111 |
| if ( 1 + 2 < 0) print (111); |  |  |
| a = 0;  //comment  print a + 1; | 1 | 1 |
| //comment  a = 0;  //comment  //comment  //comment  print a + 1;  //comment | 1 | 1 |
| a = 0;  // comment // comment //comment  print a + 1; | 1 | 1 |
| a = 0;  /\* multi-line comment  multi-line comment  multi-line comment  \*/  print a + 1; | 1 | 1 |
| var1 = 0;  var1 = var2;  print var1; | Unknown variable 'var2' | Unknown variable 'var2' |

Таблица 5.1. Результаты тестирования.

5.4 Итоги тестирования

При прохождении тестов, ошибок в функциях или алгоритмах не обнаружены. Поведение программы соответствует ожиданием.

6. Сопровождение

6.1 Требования к программно-аппаратно платформе

Из-за использования инструментов FLex и GNU Bison, а также нестандартной библиотеки FlexLexer.h программа может быть собрана только в операционных системах с ядром Linux. Требуемое место на диске 1.26Мб. Размер исполняемого файла может меняться в зависимости от системы.

Рекомендуемая система для использования: Ubuntu.

6.2 Инструкция по установке и использованию

Для сборки в проекте используется makefile.

Для генерации лексера и парсера потребуются инструменты Flex и Bison.

Для установки Flex и Bison в makefile-е есть цель install.

Для сборки проекта используется цель all в makefile.

Программа собирается в исполняемый файл zmm в главной директории проекта.

Для запуска программы необходимо вызвать ее из консоли с 1 аргументом – путем до файла с исходным кодом, который необходимо исполнить

7. Заключение

В ходе выполнения курсового проекта был создан продукт, соответствующий заданию.

Был реализован интерпретатор си-подобного языка.

Достоинства:

-Позволяет выполнять код на самодельном си-подобном языке.  
-Позволяет вводить, хранить, обрабатывать и выводить целочисленные переменные.

Недостатки:

-Не является полноценным транслятором.  
-Программа вынуждена обрабатывать файл целиком, а не построчно, и при этом не создает исполняемых файлов как компилятор.  
-Язык не позволяет написать «Hello, world!»

Перспективы дальнейшего развития:

-Оптимизация алгоритмов.  
-Добавление новых конструкция языка, таких, как, например, функции.  
-Добавление новых типов данных: дробных чисел, символов, строк и тд.  
-Изменение программы таким образом, чтобы программа транслировала код на другой язык, например Ассемблер.  
-Изменение программы таким образом, чтобы исходный код обрабатывался и выполнялся построчно.

Список литературы

1. Вычислительная техника. Терминология: Справочное пособие. Выпуск 1 / Рецензент канд. техн. наук Ю. П. Селиванов. — М.: Издательство стандартов, 1989. — 168 с. [Электронный курс] // [fileskachat.com](https://fileskachat.com) [сайт] URL: <https://fileskachat.com/view/111336_687c865431c28a591982fab8891b3af4.html>
2. Компиляция и интерпретация кода. [Электронный курс] \\ ru.hexlet.io [сайт] URL: <https://ru.hexlet.io/blog/posts/kompilyatsiya-i-interpretatsiya-koda-chto-eto-takoe-i-v-chem-raznitsa>
3. Формальная грамматика. [Электронный курс] // Википедия [сайт] URL: [https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F\_%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0#cite\_ref-\_6c47eefe4a382056\_1-0](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0%23cite_ref-_6c47eefe4a382056_1-0)
4. Lexical Analysis With Flex, for Flex 2.6.2. [Электронный курс] // westes.github.io [сайт] URL: <https://westes.github.io/flex/manual/>
5. Абстрактное синтаксическое дерево. [Электронный курс] // Википедия [сайт] URL: <https://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D0%BE%D0%B5_%D1%81%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE>
6. Bison 3.8.1. [Электронный курс] // gnu.org [сайт] URL: <https://www.gnu.org/software/bison/manual/bison.html>
7. LR(1)-разбор. [Электронный курс] // neerc.ifmo.ru [сайт] URL: <https://neerc.ifmo.ru/wiki/index.php?title=LR(1)-%D1%80%D0%B0%D0%B7%D0%B1%D0%BE%D1%80>
8. LALR(1). [Электронный курс] // Википедия [сайт] URL: <https://ru.wikipedia.org/wiki/LALR(1)>

# **Приложения**

# Приложение 1. Формальное описание грамматики

Programm: StatementList

StatementList: Statement

| StatementList Statement

Statement: IfStatement

| WhileStatement

| Assigment

| Print

| Input

IfStatement: IF LB Expression RB

LCB StatementList RCB

| IF LB Expression RB

LCB StatementList RCB

ELSE LCB StatementList RCB

WhileStatement: WHILE LB Expression RB

LCB StatementList RCB

Assigment: Identifier AssignmentOperator Expression SCL

Print: PRINT Expression SCL

Input: INPUT Variable SCL

Expression: UnaryOpertor Expression

| Expression BinaryOperator Expression

| LB Expression RB

| Identifier

| IntegerLiteral

IntegerLiteral: [1-9][0-9]\*

BinaryOperator: '+' | '-' | '\*' | '/' | '%' | '<' | '>' | '<=' | '>=' | '==' | '!=' | '&' | '&&'

UnaryOpertor: '!' | '-'

AssignmentOperator: '='

Identifier: [a-zA-Z\_][a-zA-Z0-9\_]\*

IF: 'if'

ELSE: 'else'

WHILE: 'while'

PRINT: 'print'

INPUT: 'input'

LB: '('

RB: ')'

LCB: '{'

RCB: '}'

SCL: ';'

Comment: ('//' [.]\* '/n') | ('/\*' [.]\* '\*/')

Приложение 2. Листинг grammar.y

%language "c++"

%require "3.0"

%skeleton "lalr1.cc"

%param { yy::NumDriver \* driver }

%define parse.lac full

%define api.value.type variant

%define parse.error verbose

%code requires{

#include "../AST/INode.hh"

namespace yy { class NumDriver; }

}

%code{

#include "../driver/driver.hh"

namespace yy {

void yyerror(const char \*);

parser::token\_type yylex(parser::semantic\_type\* yylval, NumDriver\* driver);

}

}

%token

END

IF

ELSE

THEN

WHILE

PRINT

INPUT

LB

RB

LCB

RCB

CL

SCL

ADD

SUB

MUL

DIV

MOD

LESS

GREATER

LS\_EQ

GR\_EQ

EQ

NEQ

OR

AND

NOT

NEG

ERR

ASSIGN

;

%nterm <AST::IScope\_ptr>

Scope

OpenBracketScope

BracketScope

BracketStatement

;

%nterm <AST::INode\_ptr>

Statement

StatementList

;

%nterm <AST::INode\_ptr>

IfStatement

WhileStatement

Print

Assigment

Input

;

%nterm <AST::INode\_ptr>

Expression

UnaryExpression

TerminalExpression

;

%left EQ NEQ OR;

%left AND NOT;

%left LESS GREATER LS\_EQ GR\_EQ;

%left ADD SUB;

%left MUL DIV MOD;

%right NEG;

%right ELSE THEN;

%token <int> INT

%token <std::string> NAME

%start Program;

%%

Program: StatementList {}

;

Scope: OpenScope StatementList CloseScope {}

;

BracketScope: OpenBracketScope StatementList CloseScope {$$ = $1;}

;

OpenScope: LCB {

auto parent = CURRENT\_SCOPE;

auto sc = AST::make\_scope(parent);

CURRENT\_SCOPE = sc.get();

if(parent) parent->push(sc);

}

;

OpenBracketScope: LCB {

$$ = AST::make\_scope(CURRENT\_SCOPE);

CURRENT\_SCOPE = $$.get();

}

;

CloseScope: RCB {CURRENT\_SCOPE = CURRENT\_SCOPE->reset\_scope();}

;

StatementList: Statement {CURRENT\_SCOPE->push($1);} |

StatementList Statement {CURRENT\_SCOPE->push($2);}|

Scope {}|

StatementList Scope {}

;

Statement: IfStatement {$$ = $1;}|

WhileStatement {$$ = $1;}|

Assigment {$$ = $1;}|

Print {$$ = $1;}|

Input {$$ = $1;}|

Expression SCL {$$ = $1;}

;

BracketStatement: Statement {$$ = AST::make\_scope(CURRENT\_SCOPE);

$$->push($1);}|

BracketScope {$$ = $1;}

;

IfStatement: IF LB Expression RB BracketStatement ELSE BracketStatement {$$ = AST::make\_if\_else($3, $5, $7);}|

IF LB Expression RB BracketStatement %prec THEN {$$ = AST::make\_if($3, $5);}

;

WhileStatement: WHILE LB Expression RB BracketStatement {$$ = AST::make\_while($3,$5);}

;

Assigment: NAME ASSIGN Expression SCL {$$ = AST::make\_assign($1, $3);}

;

Print: PRINT Expression SCL {$$ = AST::make\_print($2); }

;

Input: INPUT NAME SCL {$$ = AST::make\_assign($2, AST::make\_input());}

;

Expression: Expression ADD Expression {$$ = AST::make\_operator($1,AST::Operator\_t::ADD, $3);}|

Expression SUB Expression {$$ = AST::make\_operator($1,AST::Operator\_t::SUB, $3);}|

Expression MUL Expression {$$ = AST::make\_operator($1,AST::Operator\_t::MUL, $3);}|

Expression DIV Expression {$$ = AST::make\_operator($1,AST::Operator\_t::DIV, $3);}|

Expression MOD Expression {$$ = AST::make\_operator($1,AST::Operator\_t::MOD, $3);}|

Expression LESS Expression {$$ = AST::make\_operator($1,AST::Operator\_t::LESS, $3);}|

Expression GREATER Expression {$$ = AST::make\_operator($1,AST::Operator\_t::GREATER, $3);}|

Expression LS\_EQ Expression {$$ = AST::make\_operator($1,AST::Operator\_t::LS\_EQ, $3);}|

Expression GR\_EQ Expression {$$ = AST::make\_operator($1,AST::Operator\_t::GR\_EQ, $3);}|

Expression EQ Expression {$$ = AST::make\_operator($1,AST::Operator\_t::EQ, $3);}|

Expression NEQ Expression {$$ = AST::make\_operator($1,AST::Operator\_t::NEQ, $3);}|

Expression OR Expression {$$ = AST::make\_operator($1,AST::Operator\_t::OR, $3);}|

Expression AND Expression {$$ = AST::make\_operator($1,AST::Operator\_t::AND, $3);}|

UnaryExpression {$$ = $1;}

;

UnaryExpression: SUB UnaryExpression %prec NEG {$$ = AST::make\_unary\_operator(AST::Operator\_t::NEG,$2);}|

NOT UnaryExpression %prec NOT {$$ = AST::make\_unary\_operator(AST::Operator\_t::NOT,$2);}|

TerminalExpression {$$ = $1;}

;

TerminalExpression: LB Expression RB {$$ = $2;}|

INT {$$ = AST::make\_value($1);}|

NAME {$$ = AST::make\_reference($1);}

;

%%

namespace yy {

parser::token\_type yylex(parser::semantic\_type\* yylval,

NumDriver\* driver)

{

return driver->yylex(yylval);

}

void parser::error(const std::string& mess){

std::cout<<mess<<" in line "<<driver->plex\_->lineno()<<std::endl;

}

}

Приложение 3. Листинг grammar.l

%option c++

%option noyywrap

%option yylineno

%{

#include "grammar.tab.hh"

%}

WS [ \t\v\r]+

NUM10 [1-9][0-9]\*|0

NUM {NUM10}

COMMENT \/\\*(.|\n)\*\\*\/|\/\/.\*\n

NAME [A-Za-z\_][A-Za-z0-9\_]\*

%%

{WS}|{COMMENT} /\*do nothing\*/

"+" {return yy::parser::token\_kind\_type::ADD;}

"-" {return yy::parser::token\_kind\_type::SUB;}

"\*" {return yy::parser::token\_kind\_type::MUL;}

"/" {return yy::parser::token\_kind\_type::DIV;}

"%" {return yy::parser::token\_kind\_type::MOD;}

"<" {return yy::parser::token\_kind\_type::LESS;}

">" {return yy::parser::token\_kind\_type::GREATER;}

"<=" {return yy::parser::token\_kind\_type::LS\_EQ;}

">=" {return yy::parser::token\_kind\_type::GR\_EQ;}

"==" {return yy::parser::token\_kind\_type::EQ;}

"!=" {return yy::parser::token\_kind\_type::NEQ;}

"||" {return yy::parser::token\_kind\_type::OR;}

"&&" {return yy::parser::token\_kind\_type::AND;}

"!" {return yy::parser::token\_kind\_type::NOT;}

"=" {return yy::parser::token\_kind\_type::ASSIGN;}

";" {return yy::parser::token\_kind\_type::SCL;}

"(" {return yy::parser::token\_kind\_type::LB;}

")" {return yy::parser::token\_kind\_type::RB;}

"{" {return yy::parser::token\_kind\_type::LCB;}

"}" {return yy::parser::token\_kind\_type::RCB;}

"if" {return yy::parser::token\_kind\_type::IF;}

"else" {return yy::parser::token\_kind\_type::ELSE;}

"while" {return yy::parser::token\_kind\_type::WHILE;}

"print" {return yy::parser::token\_kind\_type::PRINT;}

"input" {return yy::parser::token\_kind\_type::INPUT;}

{NUM} {return yy::parser::token\_kind\_type::INT; }

{NAME} {return yy::parser::token\_kind\_type::NAME; }

. {return yy::parser::token\_kind\_type::ERR;}

%

Приложение 4. Листинг driver.hh

#pragma once

#include <iostream>

#include "../grammar/grammar.tab.hh"

#if !defined(yyFlexLexerOnce)

#include <FlexLexer.h>

#endif

namespace yy

{

class Driver

{

friend class parser;

FlexLexer \*plex\_;

public:

Driver(FlexLexer \*plex) : plex\_(plex) {};

parser::token\_type yylex(parser::semantic\_type \*yylval);

bool parse();

void switch\_streams(std::istream &new\_in, std::ostream &new\_out);

};

}

Приложение 5. Листинг driver.cc

#include "driver.hh"

AST::IScope \*CURRENT\_SCOPE = nullptr;

namespace yy

{

bool Driver::parse()

{

parser parser(this);

bool res = parser.parse();

return !res;

}

parser::token\_type Driver::yylex(parser::semantic\_type \*yylval)

{

parser::token\_type tt = static\_cast<parser::token\_type>(plex\_->yylex());

if (tt == parser::token\_type::INT)

try

{

yylval->emplace<int>(std::stoi(plex\_->YYText()));

}

catch (std::out\_of\_range &err)

{

throw std::runtime\_error("Int out of range in line: " + std::to\_string(plex\_->lineno()));

}

if (tt == parser::token\_type::NAME)

yylval->emplace<std::string>(std::string{plex\_->YYText()});

return tt;

}

void Driver::switch\_streams(std::istream &new\_in, std::ostream &new\_out)

{

plex\_->switch\_streams(new\_in, new\_out);

}

}

Приложение 6. Листинг main.cpp

#include <iostream>

#include "driver/driver.hh"

#include <fstream>

int main(int argc, char \*\*argv)

{

auto root = AST::make\_scope();

CURRENT\_SCOPE = root.get();

std::ifstream in;

FlexLexer \*lexer = new yyFlexLexer;

yy::Driver driver(lexer);

if (argc == 1)

std::cout << "File name is required"<< std::endl<<"zmm [file name]" << std::endl;

else if (argc == 2)

{

in.open(argv[1]);

if (in.is\_open())

{

driver.switch\_streams(in, std::cout);

try

{

if (!driver.parse())

return 1;

root->calculate();

}

catch (std::runtime\_error &err)

{

std::cerr << err.what() << std::endl;

}

}

else

std::cout << argv[1] << ": file does not exist" << std::endl;

}

else

std::cout << "To many arguments"<< std::endl<<"zmm [file name]" << std::endl;

delete lexer;

return 0;

}