1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА № 1**

1. по дисциплине «Формальные грамматики и теории компиляторов»
2. Выполнил
3. студент гр. 4851003/80802 Сошнев М.Д.

<*подпись*>

1. Преподаватель Мясников А.В.

<*подпись*>

1. Санкт-Петербург
2. 2021

Оглавление

[1 Цель работы 3](#__RefHeading___Toc4299_3652333103)

[2 Задача 3](#__RefHeading___Toc4301_3652333103)

[3 Ход работы 3](#__RefHeading___Toc4303_3652333103)

[4 Тестирование 6](#__RefHeading___Toc4305_3652333103)

[5 Вывод 8](#__RefHeading___Toc4307_3652333103)

[6 Приложение 9](#__RefHeading___Toc4309_3652333103)

[Makefile 9](#__RefHeading___Toc4420_3652333103)

[lexic.l 9](#__RefHeading___Toc4424_3652333103)

[grammar.y 11](#__RefHeading___Toc4428_3652333103)

# **Цель работы**

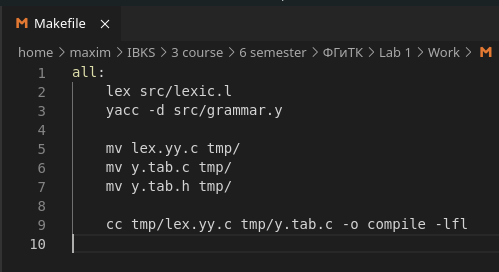
Познакомиться с парсером, лексером выражений.

# Задача

Описать грамматику, которая будет на вход получать файл с расширением \*.z, а на выходе ответ, является ли это корректным исходным файлом без синтаксических ошибок. Если да, то сообщение "ОК", например, иначе номер строки с местом ошибки и сообщение "не ок", например.

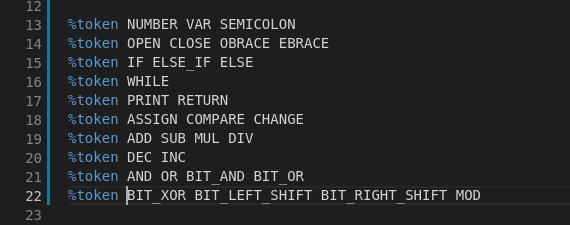
# Ход работы

С помощью утилиты pacman были скачены две программы — lex и bison, которые позволяют задавать лексику и грамматику для нашего языка. Создадим два файла — lexic.l и grammar.y в которых будем их задавать. Далее, с помощью скачанных программ из этих файлов будут сгенерирован код на языке Си, который и будет нашим исходным кодом для компилятора. Собрав его с помощью cc мы получим компилятор на выходе. Соответственно, Makefile для сборки нашего компилятора выглядит следующим образом:

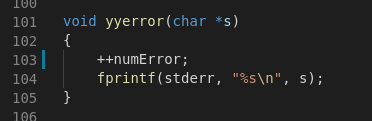
*Рисунок 1 — Makefile для сборки компилятора*

В строчках 2-3 мы генерируем Си-код, далее в строчках 5-7 переносим все исходные тексты будущего компилятора в отдельный каталог и в строчке 9 происходит сборка из исходных текстов.

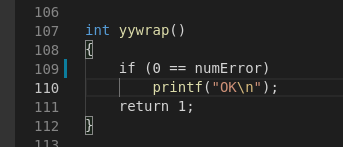
В файле lexic.l происходит разбиение языка на токены, для каждого токена в коде задаётся отдельный макрос и «передаётся» файлу grammar.y в которых задаются всевозможные цепочки из токенов, которые позволены в данном языке.

*Рисунок 2 — токены языка.*

В случае, когда в тексте встретится цепочка, которая не соответствует ни одному из правил, будет вызвана функция yyerror(), которую необходимо прописать:

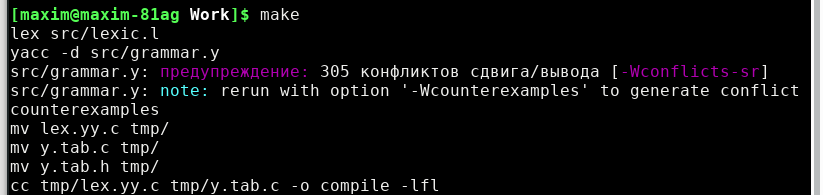
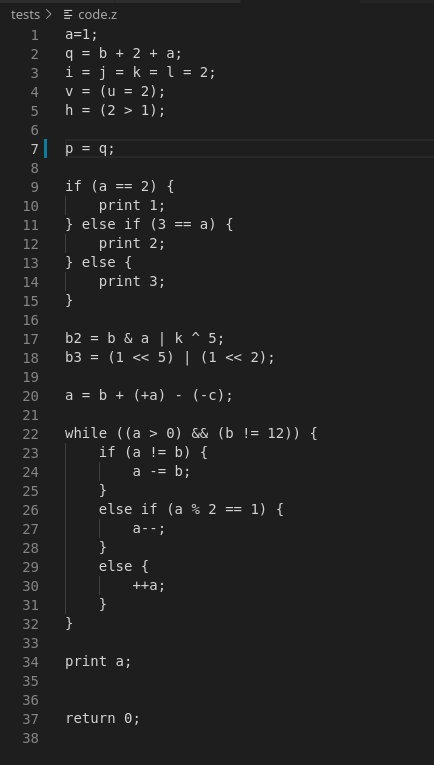
*Рисунок 3 — функция об ошибке*

В конце работы компилятора будет вызвана функция yywrap():

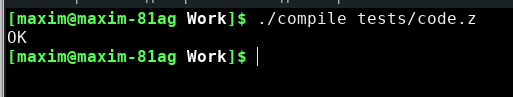
*Рисунок 4 — callback функция конца работы программы*

Входной поток программа определяет с помощью переменной FILE\* yyin, в нашем случае это будет файл, название которого задаётся через аргумент командной строки. Результат будет писаться в стандартный поток вывода — либо сообщение «Ok» - текст соответствует всем правилам языка, либо «Syntax error» - обнаружена синтаксическая ошибка.

# Тестирование

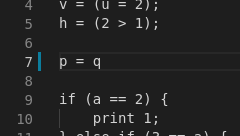
*Р**исунок 5 — успешная сборка компилятор*

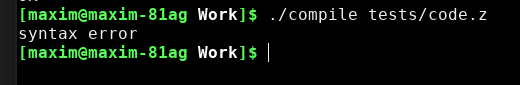
*Рисунок 6 — тестовая программа*

Проверим код на наличие синтаксических ошибок:

*Рисунок 7 — отсутствие синтаксических ошибок*

Теперь добавим синтаксическую ошибку, например удалим один символ ;

*Рисунок 8 — забытый знак ; вконце строки*

*Рисунок 9 — сообщение об ошибке*

# Вывод

Был получен опыт работы с парсером и лексером, был написан простейший компилятор, обнаруживающий синтаксические ошибки в текстах по заданным конструкциям.

# Приложение

## Makefile

all:

lex src/lexic.l

yacc -d src/grammar.y

mv lex.yy.c tmp/

mv y.tab.c tmp/

mv y.tab.h tmp/

cc tmp/lex.yy.c tmp/y.tab.c -o compile -lfl

## lexic.l

%{

#include <stdlib.h>

#include "y.tab.h"

%}

%%

\{ return OBRACE;

\} return EBRACE;

\( return OPEN;

\) return CLOSE;

print return PRINT;

return return RETURN;

if return IF;

else\ if return ELSE\_IF;

else return ELSE;

while return WHILE;

= return ASSIGN;

; return SEMICOLON;

[ \t]+ /\* игнорируем пробелы и знаки табуляции \*/

(\r\n)|\n yylval++;

>|<|>=|<=|==|!= return COMPARE;

(\/=)|(\\*=)|(\+=)|(-=) return CHANGE;

\+\+ return INC;

\-\- return DEC;

\+ return ADD;

\- return SUB;

\\* return MUL;

\/ return DIV;

&& return AND;

\|\| return OR;

& return BIT\_OR;

\| return BIT\_AND;

\^ return BIT\_XOR;

\<\< return BIT\_LEFT\_SHIFT;

>> return BIT\_RIGHT\_SHIFT;

\% return MOD;

[0-9]\* return NUMBER;

[a-zA-Z][a-zA-Z0-9]\* return VAR;

%%

## grammar.y

%{

#include <stdio.h>

void yyerror(char \*s) ;

int yylex(); // ????

extern FILE\* yyin;

int numError = 0;

%}

%start commands

%token NUMBER VAR SEMICOLON

%token OPEN CLOSE OBRACE EBRACE

%token IF ELSE\_IF ELSE

%token WHILE

%token PRINT RETURN

%token ASSIGN COMPARE CHANGE

%token ADD SUB MUL DIV

%token DEC INC

%token AND OR BIT\_AND BIT\_OR

%token BIT\_XOR BIT\_LEFT\_SHIFT BIT\_RIGHT\_SHIFT MOD

%%

commands:

/\* empty \*/ |

commands command;

command:

PRINT expr SEMICOLON /\*{ printf("print\n"); }\*/ |

RETURN expr SEMICOLON /\*{ printf("return"); }\*/ |

assigment SEMICOLON /\*{ printf("assigment\n"); }\*/ |

expr SEMICOLON { /\*printf("expr");\*/ } |

condition |

cycle\_while

;

body:

OBRACE commands EBRACE;

condition:

IF OPEN expr CLOSE

body

else\_case;

else\_case:

/\* empty \*/ |

ELSE\_IF OPEN expr CLOSE

body

else\_case |

ELSE

body;

cycle\_while:

WHILE OPEN expr CLOSE

body;

assigment:

VAR ASSIGN expr { /\*printf("assigment");\*/ };

expr:

OPEN expr CLOSE |

binary\_operation |

unary\_operation |

assigment |

VAR |

NUMBER;

unary\_operation:

INC expr | expr INC |

DEC expr | expr DEC |

ADD expr | SUB expr;

binary\_operation:

comparation |

expr ADD expr |

expr SUB expr |

expr MUL expr |

expr DIV expr |

expr CHANGE expr |

expr AND expr |

expr OR expr |

expr BIT\_AND expr |

expr BIT\_OR expr |

expr BIT\_XOR expr |

expr BIT\_LEFT\_SHIFT expr |

expr BIT\_RIGHT\_SHIFT expr |

expr MOD expr;

comparation:

expr COMPARE expr;

%%

void yyerror(char \*s)

{

++numError;

fprintf(stderr, "%s\n", s);

}

int yywrap()

{

if (0 == numError)

printf("OK\n");

return 1;

}

int main(int argc, void \*argv[])

{

yylval = 0;

if (2 != argc) {

printf("Incorrect arguments\n");

printf("Usage: ./compile [path]\n");

return -1;

}

char\* pathFile = argv[1];

yyin = fopen(pathFile,"r");

if (NULL == yyin) {

printf("No such file: %s\n", pathFile);

return -1;

}

yyparse();

fclose(yyin);

return 0;

}