Министерство образования и науки Российской Федерации

Санкт-Петербургский политехнический университет Петра Великого

—

Институт кибербезопасности и защиты информации

**КУРСОВАЯ РАБОТА**

**«Разработка парсера языка Golang»**

по дисциплине «Формальные грамматики и теория компиляторов»

Выполнили:

студент гр. 4851003/90801 Сороколетов С. А.

подпись   
 студент гр. 4851003/90801 Халилов Р. Н.

подпись

Руководитель

старший преподаватель Семьянов П.В.

подпись

Санкт-Петербург

2022

сОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc515012206)

[Цель работы 3](#_Toc515012207)

[Задачи 4](#_Toc515012208)

[ХОД РАБОТЫ 5](#_Toc515012209)

[Теоретические сведения 5](#_Toc515012210)

[Разработка лексического анализатора 6](#_Toc515012211)

[Разработка синтаксического анализатора 6](#_Toc515012212)

[Возникшие конфликты 10](#_Toc515012213)

[Трудности при разработке 13](#_Toc515012219)

[Примеры работы парсера 15](#_Toc515012220)

[РЕЗУЛЬТАТЫ X](#_Toc515012221)

[ВЫВОД X](#_Toc515012222)

ЦЕЛЬ РАБОТЫ

Получить навыки работы со средствами разработки парсеров языков программирования, а также построения грамматик, позволяющих распознать принадлежность входного кода к определенному языку программирования.

ЗАДАЧИ

Разработать парсер языка Golang используя возможности средств лексического и синтаксического анализа yacc и flex. Составить грамматику, позволяющую определить и разделить синтаксические конструкции языка программирования Go.

# ХОД РАБОТЫ

## Теоретические сведения

Golang - это статически типизированный, скомпилированный язык программирования. Синтаксически он похож на C, но с безопасностью памяти, сборкой мусора, структурной типизацией и параллелизмом в стиле CSP.

Go находится под влиянием C но с акцентом на большую простоту и безопасность. Язык Go имеет следующие характерные особенности синтаксиса:

* Необязательное краткое объявление переменной и инициализация с помощью вывода типа (x := 0 вместо int x = 0; или var x = 0;)
* Встроенные примитивы параллелизма: облегченные процессы (goroutines), каналы и оператор select
* Система интерфейсов вместо виртуального наследования и встраивание типов вместо невиртуального наследования

Пример базовой программы на языке Go:

package main

import “fmt”

func main {

var a = 1

var b int = 2

fmt.Println(a, b)

}

## 

## Разработка лексического анализатора языка

При разработке парсера таких комплексных языков программирования как Go, задачу определения конкретных структур кода берет на себя лексический анализатор.

Так, с помощью возможностей flex и применением методов регулярных выражений, было разработано решение, позволяющее определять и обрабатывать ряд структур языка Go. Например, лексический анализатор может определять ключевые конструкции, обозначающие:

* Типы переменных – int8, int16, uint, float32, complex64 и так далее;
* Операторы – if, else, import, package и так далее;
* Идентификаторы – конструкции вида [A-Za-z\_]+[A-Za-z\_0-9]\*;
* Целые числа - конструкции вида [-]?[0-9]+;
* Дробные числа - конструкции вида [-]?[0-9]+[\.][0-9]+;
* Однострочные (//) и многострочные (/\* … \*/) комментарии.

Помимо перечисленных, разработанный лексический анализатор позволяет обрабатывать еще больший ряд конструкций, что позволяет полностью покрыть и обработать синтаксис языка Go, после чего передать полученную информацию уже синтаксическому анализатору. Полный код разработанного решения представлен в Приложении 1.

## Разработка синтаксического анализатора языка

Разработка грамматики основана на документации к языку, полученный с официального сайта языка. Go устроен таким образом, что каждая строка является, по сути, обособленной командой (в отличие от С++, где команды отделяются друг от друга символом «;»). Однако, Go отличается от, например, языка программирования Python тем, что табуляция в нем не имеет значения и блоки операций отделяются друг от друга парой символов «{…}»

Таким образом, при разработке грамматики ключевой и основной единицей была выбрана именно строка. Такая строка, условно называемая «Утверждением» (Statement, рис. 1), может содержать, например, началом оператора if else, вызовом функции, началом switch – конструкции, началом оператора for и так далее. Описание «Утверждения» в грамматике можно увидеть на Рисунке 1.

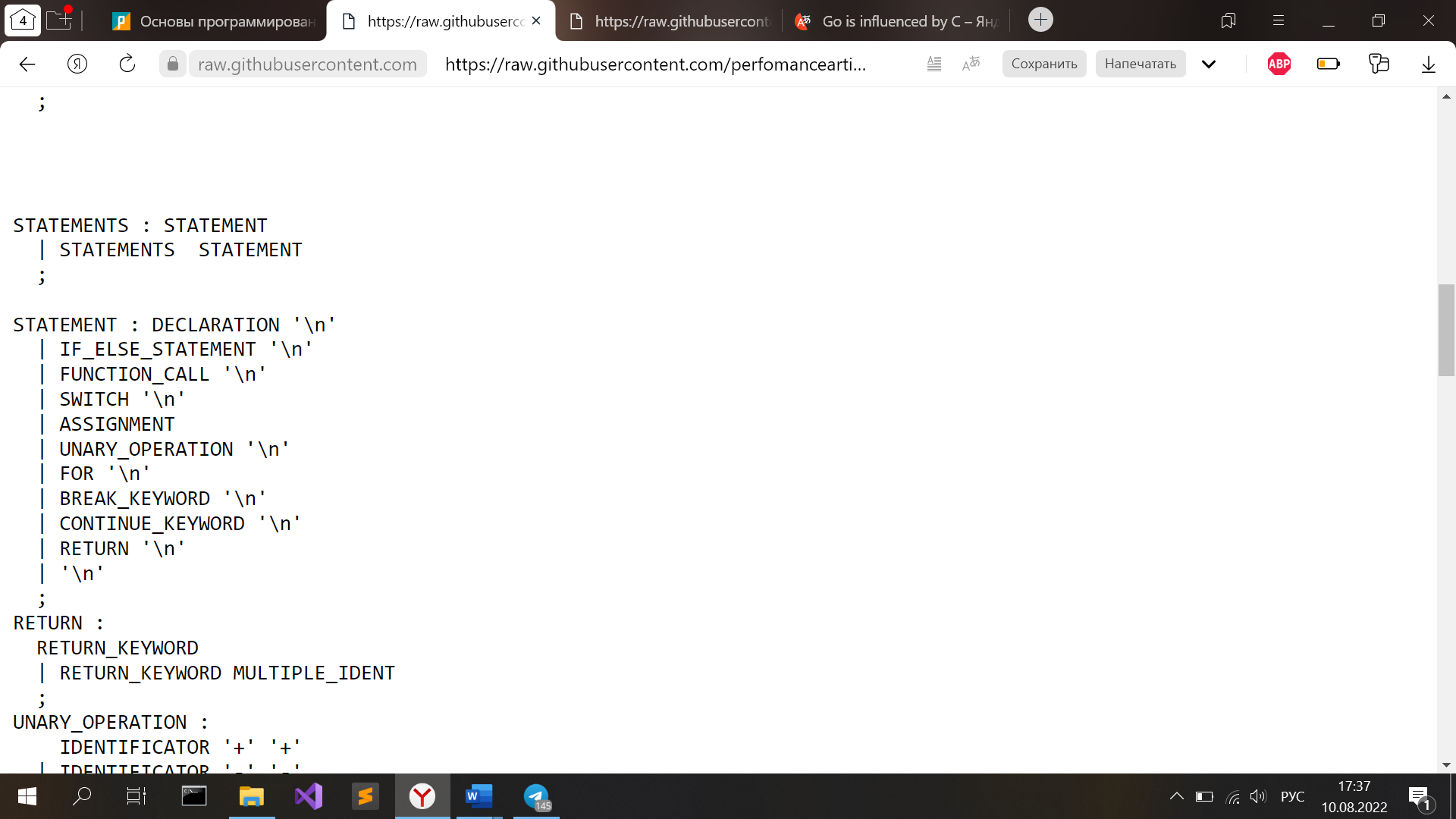


Рисунок 1 – Описание «Утверждения» в грамматике

При дальнейшей разработке грамматики статистического анализатора было установлено, что наиболее комплексной частью грамматики будет, как ни странно, объявление переменных. Это связано с тем, что в языке Go может встречаться несколько различных вариантов объявления переменных (подробнее – в пункте 1.5.2). Разработанное решение в грамматике, обрабатывающее различные возможные варианты объявления переменных представлено на Рисунках 2 и 3. Конкретно на Рисунке 2 показана общая обработка «Определения» (Declaration), через которую может раскрываться описанное выше «Утверждение». В свою очередь на Рисунке 3 представлены «Краткое объявление» (Short\_Defining), «Объявление множественных переменных» (Multiple\_Variable\_Declaration), «Простое объявление переменной» (Variable\_Declaration) и «Объявление переменной с присваиванием ей числового значения» (Variable\_Declaration\_Assignment).

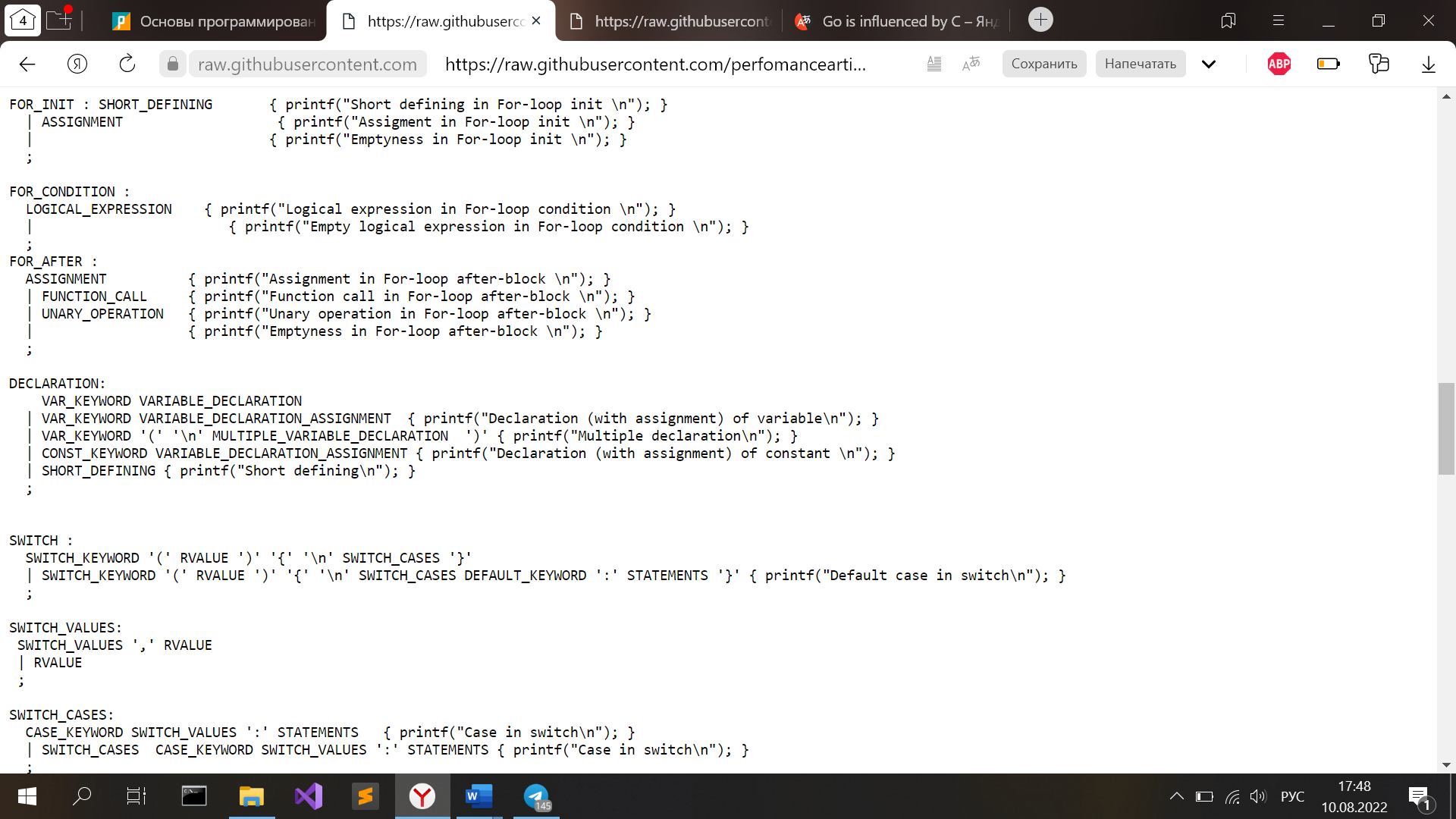


Рисунок 2 – Общее описание объявления переменной

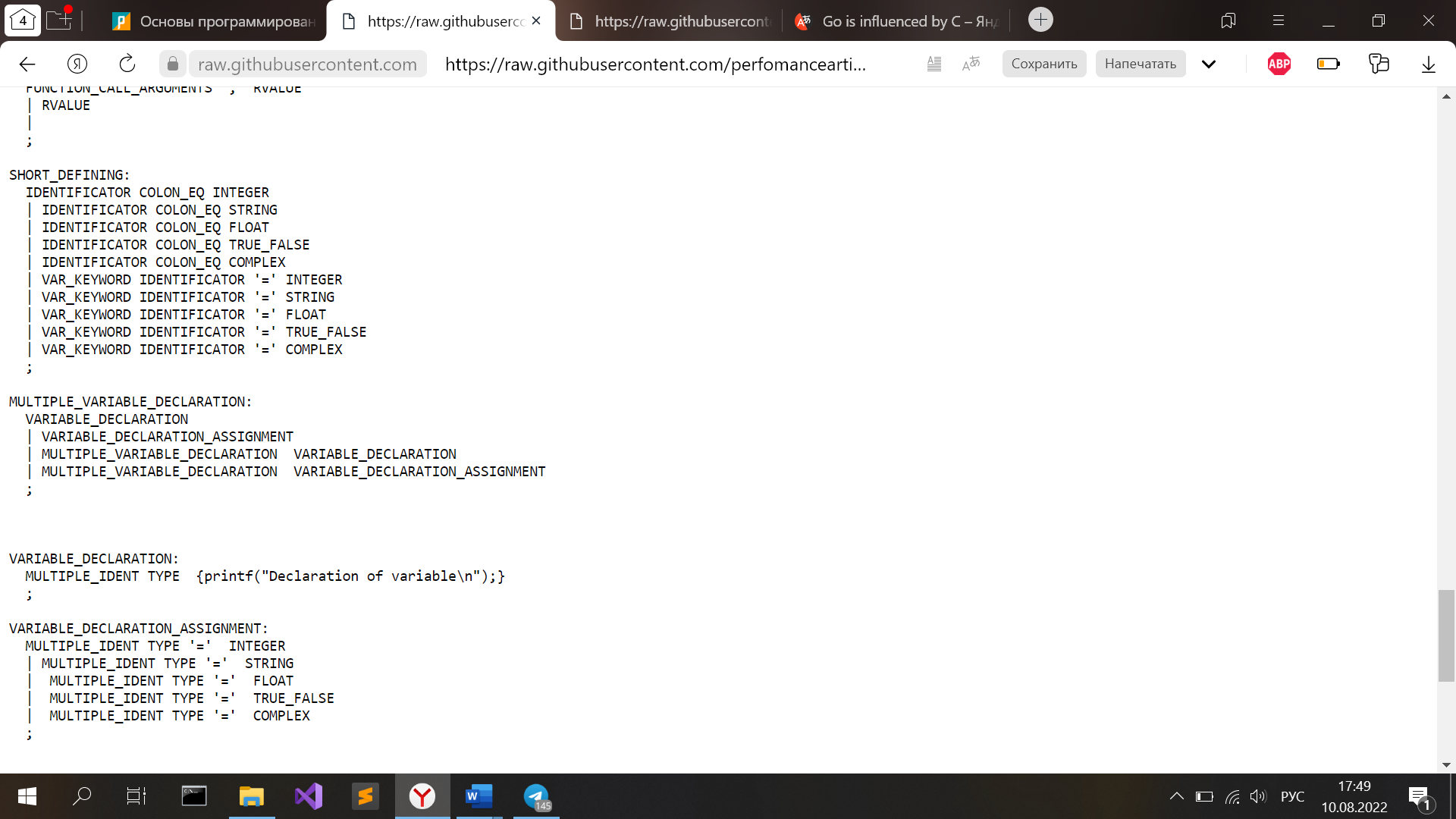


Рисунок 3 – Различные варианты объявления переменной

Помимо вышеупомянутого объявления переменных, разработанная грамматика также обрабатывает большое количество конструкций операций, перечисленных выше в формате «Утверждений». Однако, помимо этого синтаксический анализатор обрабатывает общие и более глобальные конструкции, стоящие отдельно от «Утверждений». Такими конструкциями являются импорты библиотек и объявления используемых пакетов. Обработка подобных конструкций представлена на Рисунке 4.

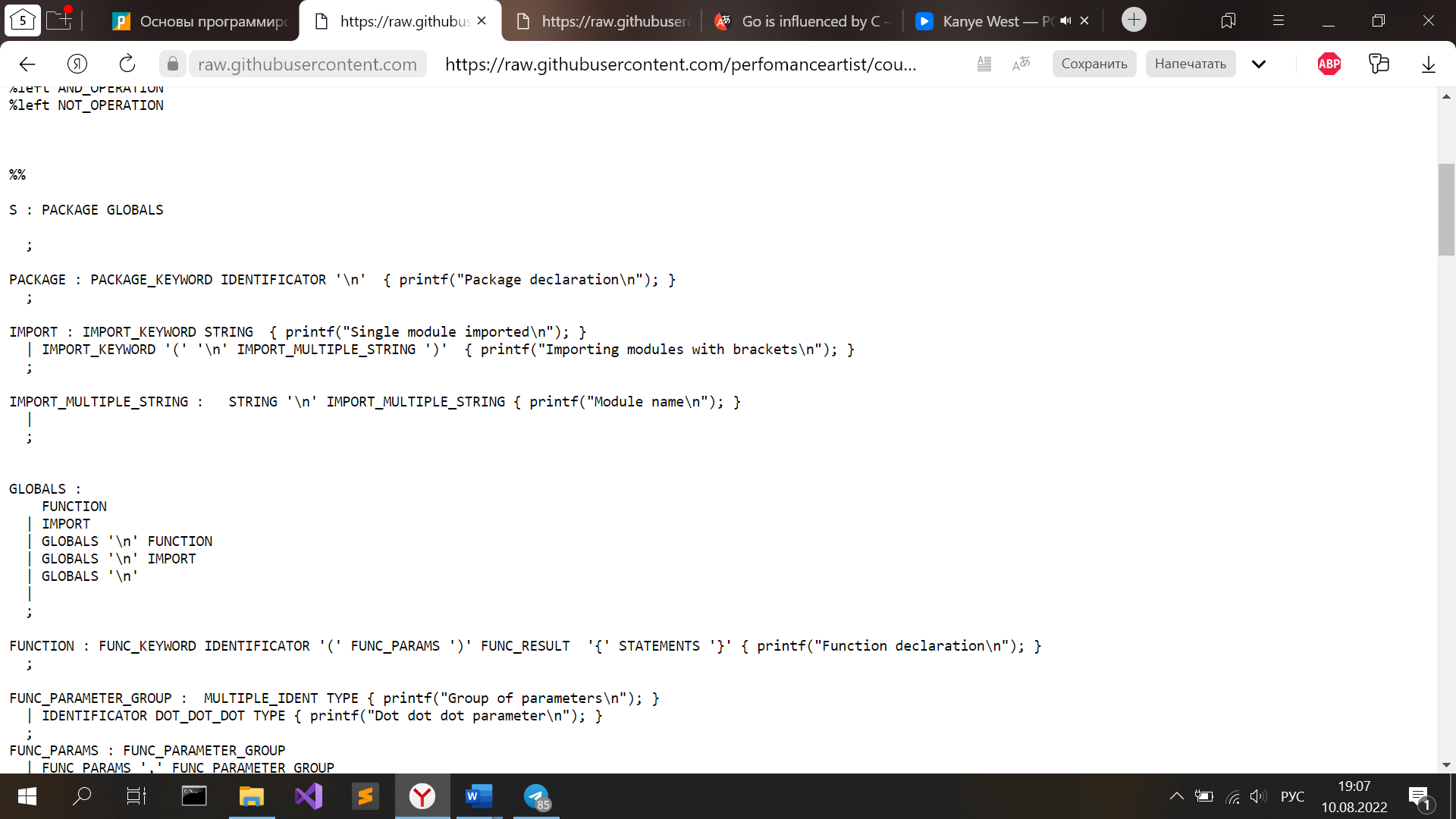


Рисунок 4 – Обработка импортов и объявления пакетов.

Также синтаксический анализатор определяет объявления функций, их вызовы, применения объявленных переменных, логические и арифметические операции и так далее. Разработка грамматики – наиболее трудоемкая и затратная по времени часть данной курсовой работы, основная сложность которой заключается в поддержании полноты и при этом лаконичности грамматики. Подробнее возникшие трудности описаны в пункте 1.5, полный код разработанной грамматики в Приложении 2.

## Конфликты грамматики

### Конфликты при обработке указателей

Во время введения в грамматику понятия указателей и их обработки возникли 3 конфликта вида *сдвиг/свертка*. Определение используемого в конфликтных строках правостороннего значения RVALUE представлено на Рисунке 5, а причиной возникновения конфликтов являются следующие строки грамматики:

POINTER '=' RVALUE { printf("Assignment of pointer by address. \n"); }

и RVALUE '\*' RVALUE

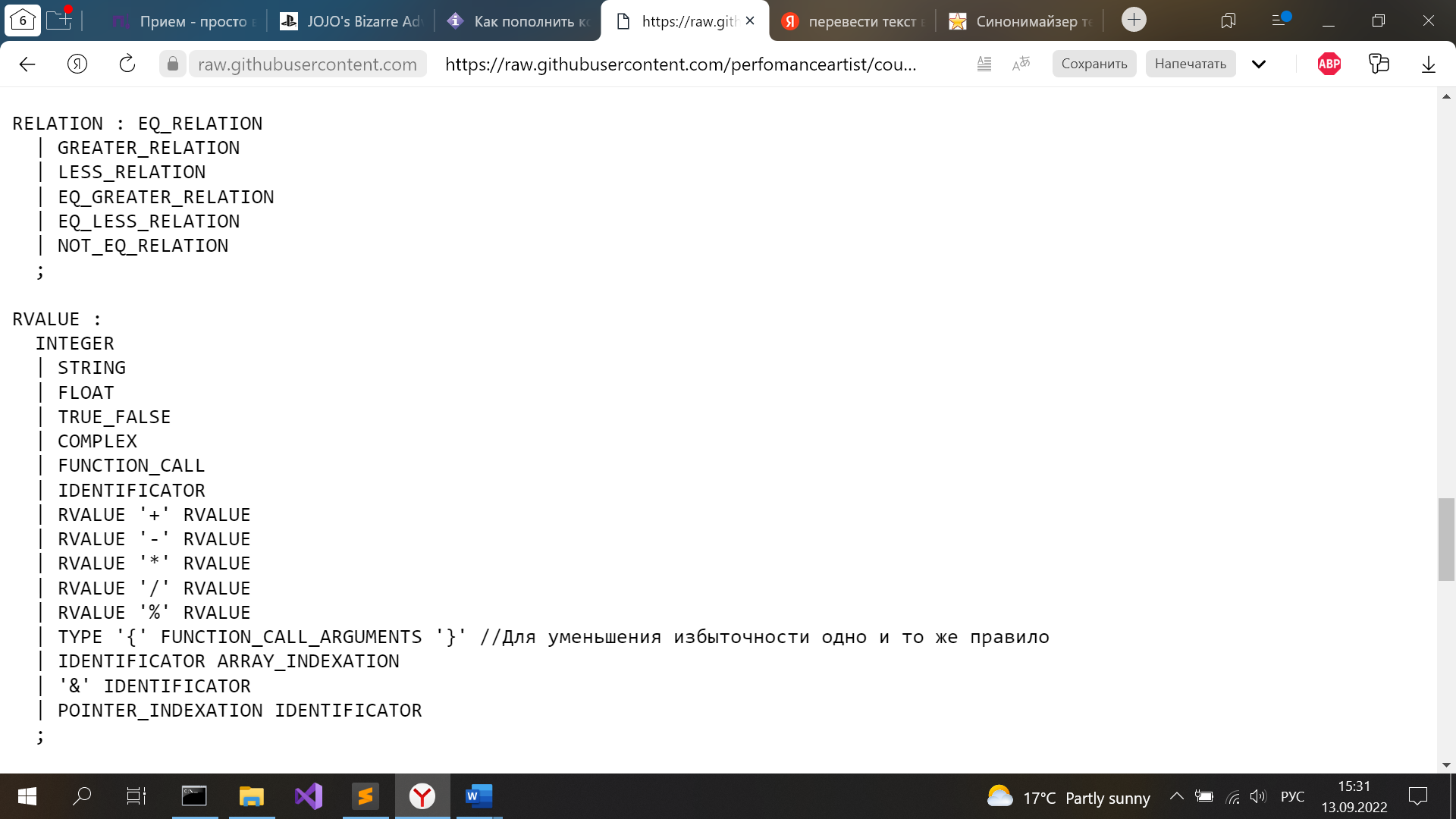


Рисунок 5 – Правостороннее значение

Конфликт возникает так как указатель имеет стандартный вид   
‘\*’ IDENTIFICATOR и при этом может являться правосторонним значением. Однако, правосторонним значением также может являться и конструкция вида RVALUE ‘\*’ RVALUE, где вместо второго RVALUE может стоять указатель, также начинающийся с символа звезды.

Таким образом, анализатор не может определить, какой именно конструкцией является ‘\*’ RVALUE – указателем или частью умножения. При попытках решения данных конфликтов был разработан вариант грамматики, представленный на Рисунке 6 и решающий 2 из 3 конфликтов.

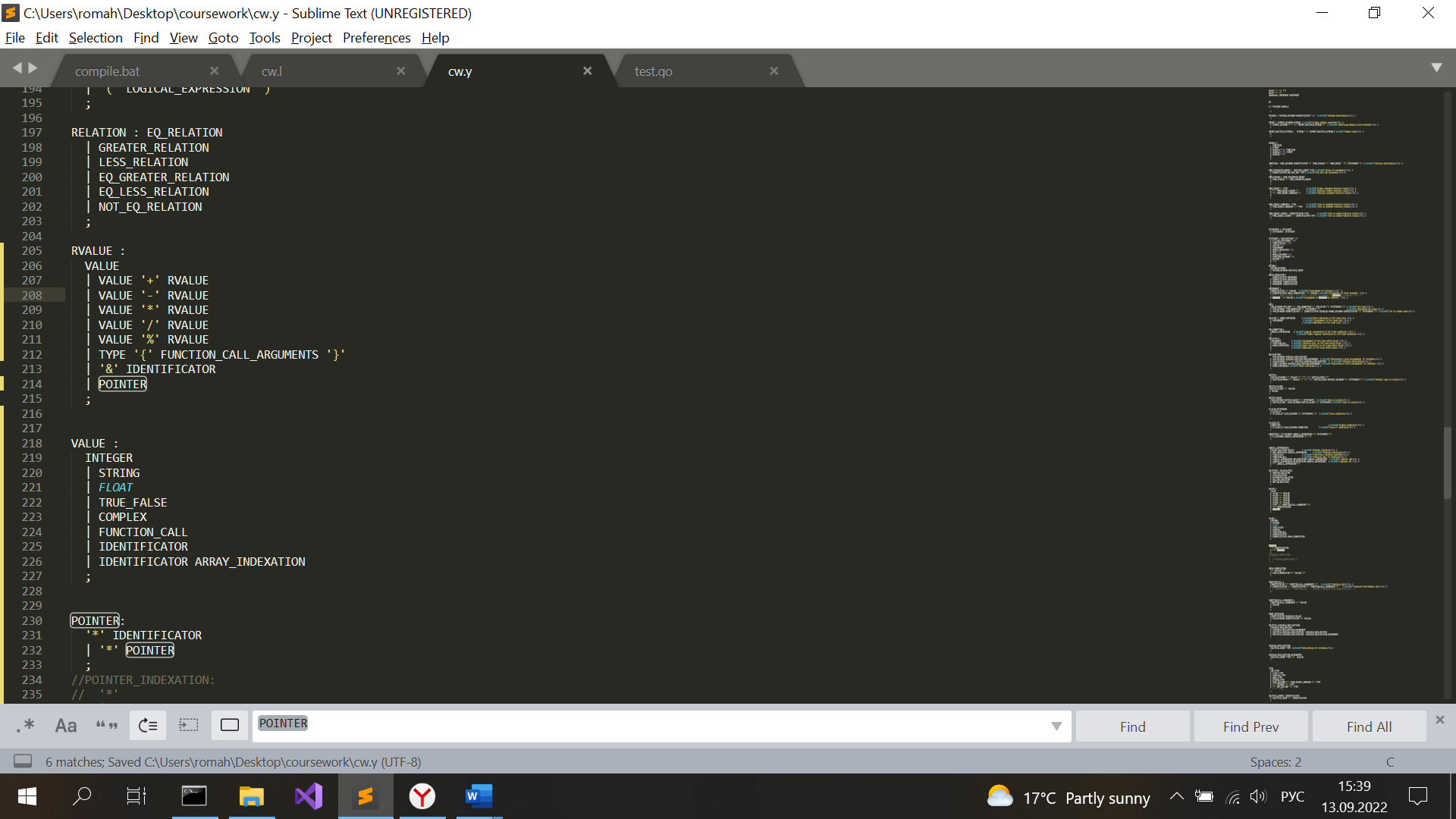


Рисунок 6 – Решение конфликтов

Смысл решения заключается в ограничении левого значения операции, теперь оно не может быть указателем. Однако, такое решение накладывает ограничения на возможности парсера, так как хоть конструкция вида ЧИСЛО \* УКАЗАТЕЛЬ практически не используется в Go, она не противоречит грамматике языка, следовательно было принято решение вернуться к предыдущей версии грамматики.

### Конфликт при обработке массивов и идентификаторов

Обращение к элементу массива в языке Go проводится аналогично многим другим языкам программирования, и структурно выглядит следующим образом:

ИДЕНТИФИКАТОР[ИНДЕКС\_ЭЛЕМЕНТА]

Однако, при обращении, например, к переменной, также используется идентификатор. Описание данного аспекта языка в грамматике (Рисунок 7) очевидно вызывает конфликт вида *сдвиг/свертка*, так как при появлении идентификатора парсеру необходимо решить между сверткой к RVALUE, либо к сдвигу с ожиданием символа [. Данный конфликт не решается для грамматики из-за особенностей устройства большинства языков программирования, однако так как yacc обращается к дальнейшему контексту при принятии решения, он не мешает корректной работе парсера.

### Конфликт при обработке типов и идентификаторов

Язык программирования Go позволяет пользователям создавать, описывать и использовать в коде собственные типы данных. Чтобы учесть данный аспект языка в грамматике изначально было принято решение ввести понятие IDENTIFICATOR в понятие TYPE.

Однако, при дальнейших разработке и тестировании было обнаружено, что данное решение приводит к большому количеству логических проблем и конфликтов (а именно – 5 конфликтам вида *сдвиг/свертка* и 1 конфликту вида *сдвиг/сдвиг*). Например, при описании пользовательских структур возникали проблемы, так как рядом с идентификатором мог стоять тип данных, в который идентификатор мог сворачиваться.

Все данные конфликты были решены отделением понятий IDENTIFICATOR и TYPE, а также дублированием некоторых правил.

## Трудности при разработке

Язык программирования Go имеет комплексный синтаксис, разработанный под влиянием языков Си и Python. Задачей синтаксиса Golang является дать программистам возможности и удобства в написании кода, позволяя иметь вариативность и свободу при написании кода. Для этого в языке Go многие конструкции имеют несколько вариаций составления.

Однако, такое большое количество возможных вариаций стандартных конструкций усложняет задачу однозначного определения этих конструкций, то есть усложняет процесс составления грамматики, распознающей язык. Трудности, возникшие во время курсовой работы, описаны ниже.

### Обработка конструкции If-Else

При составлении грамматики одной из первых серьезных трудностей стала обработка конструкции If-Else. Проблема возникла, так как в языке Go при составлении конструкции не может быть пустых строк. Это можно увидеть на примере ниже.

package main

import "fmt"

func main() {

    var command = "идти на восток"

    if command == "идти на восток" {

         fmt.Println("Вы направляетесь к горе.")

    } else if command == "зайти внутрь" {

         fmt.Println("Вы заходите в пещеру, где будете жить до конца своей жизни.")

    } else {

         fmt.Println("Пока не совсем понятно.")

    }

}

Как видно на примере (строки 10 и 12), при построении конструкции If-Else после первого набора операций (конструкция if {…}), начало следующего набора операций (конструкция else {…}) должно начинаться с той же строки, на которой оканчивается первая конструкция.

Такая особенность отличает данную конструкцию от других, из-за чего возникают трудности. До этого наличие в коде пустых строк парсером просто игнорировалось, однако в данном конкретном моменте наличие пустой строки не допускается грамматикой Golang.

Таким образом, обработка пустых строк не могла быть одинаковой для любого участка кода. Для решения сложившейся проблемы пришлось изменить возвращаемые значения лексического анализатора и их обработку в синтаксическом анализаторе, а именно ввода обработки подобных пустых строк.

### Обработка объявления переменных

Грамматически задать определение конструкции объявления переменных вызвала некоторые сложности из-за большой вариативности записи данной команды в Golang.

Например, в языке программирования Си объявление переменной строится практически всегда либо в формате единичной переменной:

<type> <name> = <value>;

Либо в формате множественных переменных:

<type> <name> = <value>, <name> = <value>;

Или, для еще более простого в обработке примера, в языке Python все переменные объявляются простой конструкцией вида:

<name> = <val>

В отличие от этих простых примеров, язык Go имеет четыре возможных варианта объявления переменных. Эти варианты:

* Простое объявление – var <name>
* Объявление со значением – var <name> := <value>
* Краткое объявление – <name> := <value>
* Два варианта множественного объявления –
  + - var ( <name> := <value> … <name> := <value>)
    - var <name1>, <name2> = <value1>, <value2>

Таким образом, обработка объявления переменных стала более сложной и комплексной. Решить сложившуюся проблему получилось с помощью разбиения Definition на более мелкие варианты раскрытия, что помогло полностью покрыть все варианты данных конструкций.

## Примеры работы парсера

# 2 РЕЗУЛЬТАТЫ

В данной курсовой работе были выполнены следующие задачи:

1. Разработан лексический анализатор языка программирования Golang.
2. Разработан лексический анализатор языка программирования Golang.
3. Разрешено большинство конфликтов типа shift/reduce в разработанной грамматике.
4. С помощью возможностей bison была собрана программа-парсер, распознающая код на языке программирования Golang.

3 ВЫВОД

В ходе выполнения данной курсовой работы были получены практические навык разработки приложения – парсера существующего популярного языка программирования с использованием yacc и flex и написания грамматики для Golang, определяющего корректность написанного и предоставленного на вход программе кода.

Разработка грамматики для определения синтаксиса языка Go проводилась по принципу «от наименьшего наибольшему», то есть основные базовые идентификаторы собираются парсером в более крупные групповые системы, которые уже обрабатывались им. Главной идеей при построении грамматики было разделение элементов на правосторонние и левосторонние значения. Такое решение позволяет логично построить грамматику, построенную на присваивании левым значениям правых и определением их парсером как таковых.

Функционал данного парсера, однако не покрывает возможности обнаружения изменений в сложившейся грамматике Go, возможном использовании фреймворков и так далее. Следовательно, возможности разработанного парсера в дальнейшем может быть расширены с помощью добавления обработки новых элементов в грамматику.

Таким образом, исходя из большого количество возникших трудностей и нерешаемых конфликтов при разработке парсера можно сделать вывод о сложности устройства подобных комплексных языков программирования, а также о сложности разработки и структурирования их грамматических и лексических анализаторов.

Приложение 1 – ЛЕКСИЧЕСКИЙ АНАЛИЗАТОР

%{

#include <stdio.h>

#include <stdlib.h>

#include "cw.tab.h"

extern unsigned line\_number;

extern int yylex();

extern void yyerror(char \* msg);

FILE \*inputStream;

#define yyin inputStream

#define tmp\_scanf(f\_, ...) fscanf(yyin, (f\_), \_\_VA\_ARGS\_\_)

int yywrap()

{

return 1;

}

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

{

inputStream = fopen(argv[1], "r");

if (inputStream == NULL)

{

printf("File wasn't found\n");

return -1;

}

yyparse();

fclose(inputStream);

}

%}

%%

[/][/].\* { ; }

package { return PACKAGE\_KEYWORD; }

import { return IMPORT\_KEYWORD; }

func { return FUNC\_KEYWORD; }

if { return IF\_KEYWORD; }

else { return ELSE\_KEYWORD; }

switch { return SWITCH\_KEYWORD; }

case { return CASE\_KEYWORD; }

default { return DEFAULT\_KEYWORD; }

for { return FOR\_KEYWORD; }

break { return BREAK\_KEYWORD; }

continue { return CONTINUE\_KEYWORD; }

range { return RANGE\_KEYWORD; }

return { return RETURN\_KEYWORD; }

var { return VAR\_KEYWORD; }

int8 { return TYPE\_KEYWORD; }

int16 { return TYPE\_KEYWORD; }

int32 { return TYPE\_KEYWORD; }

int64 { return TYPE\_KEYWORD; }

uint8 { return TYPE\_KEYWORD; }

uint16 { return TYPE\_KEYWORD; }

uint32 { return TYPE\_KEYWORD; }

uint64 { return TYPE\_KEYWORD; }

byte { return TYPE\_KEYWORD; }

rune { return TYPE\_KEYWORD; }

int { return TYPE\_KEYWORD; }

uint { return TYPE\_KEYWORD; }

float32 { return TYPE\_KEYWORD; }

float64 { return TYPE\_KEYWORD; }

complex64 { return TYPE\_KEYWORD; }

complex128 { return TYPE\_KEYWORD; }

bool { return TYPE\_KEYWORD; }

string { return TYPE\_KEYWORD; }

const { return CONST\_KEYWORD;}

[-]?[0-9]+ { return INTEGER; }

[-]?[0-9]+[\.][0-9]+ { return FLOAT; }

true { return TRUE\_FALSE; }

false { return TRUE\_FALSE; }

[-]?[0-9]+[[\+][0-9]+[i]]? { return COMPLEX; }

\"(\\.|[^"\\])\*\" { return STRING; }

:= return COLON\_EQ;

[A-Za-z\_]+[A-Za-z\_0-9]\* { return IDENTIFICATOR; }

"==" { return EQ\_RELATION; }

">" { return GREATER\_RELATION; }

"<" { return LESS\_RELATION; }

">=" { return EQ\_GREATER\_RELATION; }

"<=" { return EQ\_LESS\_RELATION; }

"!=" { return NOT\_EQ\_RELATION; }

"!" { return NOT\_OPERATION; }

"&&" { return AND\_OPERATION; }

"||" { return OR\_OPERATION; }

"..." { return DOT\_DOT\_DOT; }

[-+()\\*=\.,\{\}\:] { return yytext[0]; }

[ \t\f\v] { ; }

"\n" { return yytext[0]; }

"^" { return '^'; }

";" { return ';'; }

%%

Приложение 2 – синтаскический АНАЛИЗАТОР

%{

#include <stdio.h>

#include <stdlib.h>

int yylex();

//extern int yylex();

void yyerror(char \*msg);

%}

%union {

int num;

char letter;

}

%token IDENTIFICATOR COLON\_EQ DOT\_DOT\_DOT

%token STRING INTEGER FLOAT TRUE\_FALSE COMPLEX

%token CONST\_KEYWORD PACKAGE\_KEYWORD IMPORT\_KEYWORD VAR\_KEYWORD TYPE\_KEYWORD FUNC\_KEYWORD RETURN\_KEYWORD

%token IF\_KEYWORD ELSE\_KEYWORD SWITCH\_KEYWORD CASE\_KEYWORD DEFAULT\_KEYWORD

%token EQ\_RELATION GREATER\_RELATION LESS\_RELATION EQ\_GREATER\_RELATION EQ\_LESS\_RELATION NOT\_EQ\_RELATION

%token FOR\_KEYWORD BREAK\_KEYWORD CONTINUE\_KEYWORD RANGE\_KEYWORD

%left OR\_OPERATION

%left AND\_OPERATION

%left NOT\_OPERATION

%%

S : PACKAGE GLOBALS

;

PACKAGE : PACKAGE\_KEYWORD IDENTIFICATOR '\n' { printf("Package declaration\n"); }

;

IMPORT : IMPORT\_KEYWORD STRING { printf("Single module imported\n"); }

| IMPORT\_KEYWORD '(' '\n' IMPORT\_MULTIPLE\_STRING ')' { printf("Importing modules with brackets\n"); }

;

IMPORT\_MULTIPLE\_STRING : STRING '\n' IMPORT\_MULTIPLE\_STRING { printf("Module name\n"); }

|

;

GLOBALS :

FUNCTION

| IMPORT

| GLOBALS '\n' FUNCTION

| GLOBALS '\n' IMPORT

| GLOBALS '\n'

|

;

FUNCTION : FUNC\_KEYWORD IDENTIFICATOR '(' FUNC\_PARAMS ')' FUNC\_RESULT '{' STATEMENTS '}' { printf("Function declaration\n"); }

;

FUNC\_PARAMETER\_GROUP : MULTIPLE\_IDENT TYPE { printf("Group of parameters\n"); }

| IDENTIFICATOR DOT\_DOT\_DOT TYPE { printf("Dot dot dot parameter\n"); }

;

FUNC\_PARAMS : FUNC\_PARAMETER\_GROUP

| FUNC\_PARAMS ',' FUNC\_PARAMETER\_GROUP

|

;

FUNC\_RESULT : TYPE { printf("Single unnamed function result\n"); }

| '(' FUNC\_RESULT\_NAMED ')' { printf("Multiple named function result\n"); }

| '(' FUNC\_RESULT\_UNNAMED ')' { printf("Multiple unnamed function result\n"); }

|

;

FUNC\_RESULT\_UNNAMED : TYPE { printf("Type in unnamed function result\n"); }

| FUNC\_RESULT\_UNNAMED ',' TYPE { printf("Type in unnamed function result\n"); }

;

FUNC\_RESULT\_NAMED : IDENTIFICATOR TYPE { printf("Type in named function result\n"); }

| FUNC\_RESULT\_NAMED ',' IDENTIFICATOR TYPE { printf("Type in named function result\n"); }

;

STATEMENTS : STATEMENT

| STATEMENTS STATEMENT

;

STATEMENT : DECLARATION '\n'

| IF\_ELSE\_STATEMENT '\n'

| FUNCTION\_CALL '\n'

| SWITCH '\n'

| ASSIGNMENT

| UNARY\_OPERATION '\n'

| FOR '\n'

| BREAK\_KEYWORD '\n'

| CONTINUE\_KEYWORD '\n'

| RETURN '\n'

| '\n'

;

RETURN :

RETURN\_KEYWORD

| RETURN\_KEYWORD MULTIPLE\_IDENT

;

UNARY\_OPERATION :

IDENTIFICATOR '+' '+'

| IDENTIFICATOR '-' '-'

| '+' '+' IDENTIFICATOR

| '-' '-' IDENTIFICATOR

;

ASSIGNMENT : IDENTIFICATOR '=' RVALUE

;

FOR :

FOR\_KEYWORD FOR\_INIT ';' FOR\_CONDITION ';' FOR\_AFTER '{' STATEMENTS '}' { printf("For-loop\n"); }

| FOR\_KEYWORD FOR\_CONDITION '{' STATEMENTS '}' { printf("Shortened for-loop\n"); }

| FOR\_KEYWORD IDENTIFICATOR ',' IDENTIFICATOR COLON\_EQ RANGE\_KEYWORD IDENTIFICATOR '{' STATEMENTS '}' { printf("For in range loop\n"); }

;

FOR\_INIT : SHORT\_DEFINING { printf("Short defining in For-loop init \n"); }

| ASSIGNMENT { printf("Assigment in For-loop init \n"); }

| { printf("Emptyness in For-loop init \n"); }

;

FOR\_CONDITION :

LOGICAL\_EXPRESSION { printf("Logical expression in For-loop condition \n"); }

| { printf("Empty logical expression in For-loop condition \n"); }

;

FOR\_AFTER :

ASSIGNMENT { printf("Assignment in For-loop after-block \n"); }

| FUNCTION\_CALL { printf("Function call in For-loop after-block \n"); }

| UNARY\_OPERATION { printf("Unary operation in For-loop after-block \n"); }

| { printf("Emptyness in For-loop after-block \n"); }

;

DECLARATION:

VAR\_KEYWORD VARIABLE\_DECLARATION

| VAR\_KEYWORD VARIABLE\_DECLARATION\_ASSIGNMENT { printf("Declaration (with assignment) of variable\n"); }

| VAR\_KEYWORD '(' '\n' MULTIPLE\_VARIABLE\_DECLARATION ')' { printf("Multiple declaration\n"); }

| CONST\_KEYWORD VARIABLE\_DECLARATION\_ASSIGNMENT { printf("Declaration (with assignment) of constant \n"); }

| SHORT\_DEFINING { printf("Short defining\n"); }

;

SWITCH :

SWITCH\_KEYWORD '(' RVALUE ')' '{' '\n' SWITCH\_CASES '}'

| SWITCH\_KEYWORD '(' RVALUE ')' '{' '\n' SWITCH\_CASES DEFAULT\_KEYWORD ':' STATEMENTS '}' { printf("Default case in switch\n"); }

;

SWITCH\_VALUES:

SWITCH\_VALUES ',' RVALUE

| RVALUE

;

SWITCH\_CASES:

CASE\_KEYWORD SWITCH\_VALUES ':' STATEMENTS { printf("Case in switch\n"); }

| SWITCH\_CASES CASE\_KEYWORD SWITCH\_VALUES ':' STATEMENTS { printf("Case in switch\n"); }

;

IF\_ELSE\_STATEMENT:

IF\_ELSE\_IF

| IF\_ELSE\_IF ELSE\_KEYWORD '{' STATEMENTS '}' { printf("Else condition\n"); }

;

IF\_ELSE\_IF:

CONDITION { printf("Simple condition\n"); }

| IF\_ELSE\_IF ELSE\_KEYWORD CONDITION { printf("Else-if condition\n"); }

;

CONDITION : IF\_KEYWORD LOGICAL\_EXPRESSION '{' STATEMENTS '}'

| IF\_KEYWORD LOGICAL\_EXPRESSION '{' '}'

;

LOGICAL\_EXPRESSION :

RVALUE RELATION RVALUE { printf("Rvalues relation\n"); }

| NOT\_OPERATION LOGICAL\_EXPRESSION { printf("Denying expression\n"); }

| TRUE\_FALSE { printf("True/False relation operand\n"); }

| FUNCTION\_CALL { printf("Function call in relation\n"); }

| LOGICAL\_EXPRESSION AND\_OPERATION LOGICAL\_EXPRESSION { printf("Logical AND\n"); }

| LOGICAL\_EXPRESSION OR\_OPERATION LOGICAL\_EXPRESSION { printf("Logincal OR \n"); }

| '(' LOGICAL\_EXPRESSION ')'

;

RELATION : EQ\_RELATION

| GREATER\_RELATION

| LESS\_RELATION

| EQ\_GREATER\_RELATION

| EQ\_LESS\_RELATION

| NOT\_EQ\_RELATION

;

RVALUE :

INTEGER

| STRING

| FLOAT

| TRUE\_FALSE

| COMPLEX

| FUNCTION\_CALL

| IDENTIFICATOR

;

FUNCTION\_CALL : IDENTIFICATOR '(' FUNCTION\_CALL\_ARGUMENTS ')' { printf("Function call\n"); } ;

FUNCTION\_CALL\_ARGUMENTS :

FUNCTION\_CALL\_ARGUMENTS ',' RVALUE

| RVALUE

|

;

SHORT\_DEFINING:

IDENTIFICATOR COLON\_EQ INTEGER

| IDENTIFICATOR COLON\_EQ STRING

| IDENTIFICATOR COLON\_EQ FLOAT

| IDENTIFICATOR COLON\_EQ TRUE\_FALSE

| IDENTIFICATOR COLON\_EQ COMPLEX

| VAR\_KEYWORD IDENTIFICATOR '=' INTEGER

| VAR\_KEYWORD IDENTIFICATOR '=' STRING

| VAR\_KEYWORD IDENTIFICATOR '=' FLOAT

| VAR\_KEYWORD IDENTIFICATOR '=' TRUE\_FALSE

| VAR\_KEYWORD IDENTIFICATOR '=' COMPLEX

;

MULTIPLE\_VARIABLE\_DECLARATION:

VARIABLE\_DECLARATION

| VARIABLE\_DECLARATION\_ASSIGNMENT

| MULTIPLE\_VARIABLE\_DECLARATION VARIABLE\_DECLARATION

| MULTIPLE\_VARIABLE\_DECLARATION VARIABLE\_DECLARATION\_ASSIGNMENT

;

VARIABLE\_DECLARATION:

MULTIPLE\_IDENT TYPE {printf("Declaration of variable\n");}

;

VARIABLE\_DECLARATION\_ASSIGNMENT:

MULTIPLE\_IDENT TYPE '=' INTEGER

| MULTIPLE\_IDENT TYPE '=' STRING

| MULTIPLE\_IDENT TYPE '=' FLOAT

| MULTIPLE\_IDENT TYPE '=' TRUE\_FALSE

| MULTIPLE\_IDENT TYPE '=' COMPLEX

;

TYPE: TYPE\_KEYWORD

| FUNC\_KEYWORD '(' FUNC\_RESULT\_UNNAMED ')' TYPE

;

MULTIPLE\_IDENT: IDENTIFICATOR

| MULTIPLE\_IDENT ',' IDENTIFICATOR

;

%%

void yyerror(char \* msg) {

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

exit(1);

}