Минобрнауки России

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Волгоградский государственный технический университет»

Факультет \_электроники и вычислительной техники\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра «Программное обеспечение автоматизированных систем»\_\_\_\_

##### **ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

**к курсовой работе (проекту)**

по дисциплине «Теория формальных языков и методы трансляции»\_\_\_\_\_\_\_\_\_

на тему «Разработка компилятора для языка программирования Golang»\_\_\_\_\_

Студент\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Астрелин Александр Алексеевич\_\_\_\_\_\_\_\_\_\_\_\_\_\_

##### (фамилия, имя, отчество)

Группа\_\_\_ПрИн-467\_\_\_\_\_\_\_\_\_\_\_\_

Студент\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Печенов Иван Павлович\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

##### (фамилия, имя, отчество)

Группа\_\_\_ПрИн-467\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель работы (проекта) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_Сычев О. А.\_\_\_

(подпись и дата подписания) (инициалы и фамилия)

Нормоконтролер \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_ Сычев О. А.\_\_\_\_\_\_

(подпись, дата подписания) (инициалы и фамилия)

Волгоград 2021 г.

Содержание

[1 Синтаксис и семантика 3](#_Toc30341)

[2 Пример программы на языке Golang 4](#_Toc3816)

[3 Карточка курсовой работы 5](#_Toc20636)

[4 Таблица лексем 6](#_Toc15959)

[5 Грамматика языка 9](#_Toc8788)

[Выводы 20](#_Toc23999)

1 Синтаксис и семантика

Golang, или Go - это статически типизированный и компилируемый язык, который поддерживает типобезопасность и многопоточность, а также возможность динамического ввода данных и содержит богатую стандартную библиотеку функций и встроенные типы данных вроде массивов с динамическим размером и ассоциативных массивов.

Golang строготипизированный язык, в нем существуют 3 способа объявить переменную:

1. Указание типа вручную

var a int = 9

1. Объявление без явного указания типа переменной(без ключевого слова var)

a := 9

1. Объявление без явного указания типа переменной

var a = 9

В Golang если объявить переменную но не присвоить ей никакого значение то ей будет присвоено нулевое значение для типа переменной

В Golang разделителем операторов является переход на новую строку или символ “;”

В Golang существует 2 вида import

Единичный импорт

import “fmt”

Множественный импорт

import

( “fmt”

“math”

)

2 Пример программы на языке Golang

Пример кода на языке руби представлен в виде простой программы:

package main

import "fmt"

func main() {

a := [5]int{3, 2, 4, 5, 1}

BubbleSort(a)

fmt.Print(a)

}

func BubbleSort(ar [5]int) {

for i := 0; i < len(ar); i++ {

for j := i; j < len(ar); j++ {

if ar[i] > ar[j] {

swap(ar, i, j)

}

}

}

}

func swap(ar [5]int, i int, j int) {

tmp := ar[i]

ar[i] = ar[j]

ar[j] = tmp

}

3 Карточка курсовой работы

Ниже приведет перечень реализованных функций в рамках данной работы (Таблица 1).

Таблица 1. Реализованные функции

|  |  |  |  |
| --- | --- | --- | --- |
| Возможность | Вес | Особенности в языке | Кто реализует |
| Локальные переменные встроенных типов данных :   * целые числа (один тип), * символы * строки   Выражения с использованием локальных переменных, арифметических операций (4 вида), операций сравнения и присваивания | 4 min |  | Астрелин А.А. |
| Одномерные массивы (или контейнеры),  операция доступа к элементу массива | 3 min |  | Печенов И.П. |
| Числовые (целые числа), символьные и строковые константы (литералы) с поддержкой всех видов констант и служебных последовательностей символов | 2 min |  | Печенов И.П. |
| Управляющие структуры: развилки | 3 min |  | Астрелин А.А. |
| Управляющие структуры: циклы | 3 min |  | Печенов И.П. |
| Функции (процедуры) | 3 min |  | Астрелин А.А. |
| Классы/функции для работы с консолью (ввод/вывод базовых типов данных) | 2 min |  | Печенов И.П. |
| Многомерные массивы | 3 |  | Печенов И.П. |
| Логические операции  (И, ИЛИ, НЕ) | 1 |  | Печенов И.П. |

4 Таблица лексем

|  |  |  |
| --- | --- | --- |
| Название | Семантическое значение | Описание |
| LESS | < | Проверяет, меньше ли значение левого операнда, чем значение правого операнда, если да то значение становится истинным |
| GREATER | > | Проверяет, больше ли значение левого операнда, чем значение правого операнда, если да то значение становится истинным |
| EQUAL | == | Проверяет, равны ли значения двух операндов или нет, если да, то условие становится истинным. |
| NOT\_EQUAL | != | Проверяет, не равны ли значения двух операндов, если да, то условие становится истинным. |
| GREATER\_OR\_EQUAL | >= | Проверяет, больше ли значение левого операнда или равно значению правого операнда, если да, то условие становится истинным |
| LESS\_OR\_EQUAL | <= | Проверяет, меньше ли значение левого операнда или равно значению правого операнда, если да, то условие становится истинным |
| SHORT\_EQUALS | := | Оператор, который дает возможность присваивать левому операнду правый операнд, без точного типа и без ключевого слова var |
| PLUS\_ASSIGN | += | Оператор, который складывает левый и правый операнды и присваивает результат операции в левом операнде |
| MINUS\_ASSIGN | -= | Оператор, который вычитает из левого операнда правый операнд и присваивает результат в левый операнд |
| MULTIPLY\_ASSIGN | \*= | Оператор, который умножает левый операнд на правый операнд и присваивает результат в левый операнд |
| DIVISION\_ASSIGN | /= | Оператор, который делит левый операнд на правый операнд и присваивает результат в левый операнд |
| INC | ++ | Оператор инкремента, увеличивает операнд на 1 |
| DEC | -- | Оператор инкремента, уменьшает операнд на 1 |
| CONST\_KEYWORD | const | Ключевое слово, которое позволяет создавать неизменные значения |
| IF\_KEYWORD | if | Конструктор условных операторов в Golang |
| ELSE\_KEYWORD | else | Добавляет блок с действиями, которые следует выполнять, если условие внутри if является ложным. Должен находится после блока if |
| FOR\_KEYWORD | for | Конструктор циклов в Golang |
| FUNC\_KEYWORD | func | Ключевое слово, является началом для объявления функции |
| RETURN\_KEYWORD | return | Оператор возврата, возвращает все свои операнды из текущей функции в функцию вызывающую |
| VAR\_KEYWORD | var | Ключевое слово, которое является началом объявления переменной |
| BOOL\_KEYWORD | bool | Является одним из главных типов, логический тип |
| INT\_KEYWORD | int | Является одним из главных типов, целочисленный тип |
| STRING\_KEYWORD | string | Является одним из главных типов, строковый тип |
| TRUE\_KEYWORD | true | Булевое значение обозначающее истину |
| FALSE\_KEYWORD | false | Булевое значение обозначающее ложь |
| PACKAGE\_KEYWORD | package | Ключевое слово, обозначающее к какому пакету относится данный файл |
| IMPORT\_KEYWORD | import | Ключевое слово, подключающее к текущему пакету, другие пакеты |
| BREAK\_KEYWORD | break | Ключевое слово, прерывает текущую итерацию цикла и завершает его |
| CONTINUE\_KEYWORD | continue | Ключевое слово, прерывает текущую итерацию цикла и переходит к следующей итерации цикла |

5 Грамматика языка

%{

#include <stdio.h>

#include "create\_tree\_nodes.cpp"

extern struct program\_struct\* root;

extern int yylex(void);

extern int yyparse(void);

int yyerror(char \* s);

%}

%union {

int Int\_val;

char\* Id;

char\* String;

struct node\* node\_value;

struct for\_stmt\_struct\* for\_stmt\_value;

struct stmt\_body\_struct\* body\_value;

struct node\_list\_struct\* node\_list\_value;

struct program\_struct\* program\_value;

struct import\_one\_of\_list\_struct\* import\_one\_of\_list\_value;

struct import\_decl\_list\_struct\* import\_decl\_list\_value;

struct import\_list\_struct\* import\_list\_value;

struct import\_struct\* import\_value;

struct package\_struct\* package\_value;

struct complex\_data\_type\* complex\_data\_value;

struct id\_list\_struct\* id\_list\_value;

struct decl\_all\_struct\* decl\_all\_value;

struct decl\_all\_list\_struct\* decl\_all\_list\_value;

struct decl\_stmt\_struct\* decl\_stmt\_value;

struct stmt\_struct\* stmt\_value;

struct highest\_decl\_struct\* highest\_decl\_value;

struct stmt\_list\_struct\* stmt\_list\_value;

struct if\_stmt\_block\_struct\* if\_stmt\_block\_value;

struct else\_if\_stmt\_list\_struct\* else\_if\_stmt\_list\_value;

struct param\_struct\* param\_value;

struct param\_list\_struct\* param\_list\_value;

struct return\_values\_struct\* return\_value;

struct highest\_decl\_list\_struct\* highest\_decl\_list\_value;

struct array\_element\_struct\* array\_element\_value;

struct array\_element\_list\_struct\* array\_element\_list\_value;

};

%token LESS

%token GREATER

%token EQUAL

%token NOT\_EQUAL

%token GREATER\_OR\_EQUAL

%token LESS\_OR\_EQUAL

%token SHORT\_EQUALS

%token PLUS\_ASSIGN

%token MINUS\_ASSIGN

%token MULTIPLY\_ASSIGN

%token DIVISION\_ASSIGN

%token INC

%token DEC

%token CONST\_KEYWORD

%token IF\_KEYWORD

%token ELSE\_KEYWORD

%token FOR\_KEYWORD

%token FUNC\_KEYWORD

%token RETURN\_KEYWORD

%token VAR\_KEYWORD

%token BOOL\_KEYWORD

%token INT\_KEYWORD

%token BYTE\_KEYWORD

%token STRING\_KEYWORD

%token TRUE\_KEYWORD

%token FALSE\_KEYWORD

%token PACKAGE\_KEYWORD

%token IMPORT\_KEYWORD

%token BREAK\_KEYWORD

%token CONTINUE\_KEYWORD

%token <Id> ID

%token <Int\_val> INT

%token <String> STRING

%type<node\_value> l\_value

%type<node\_list\_value> l\_value\_list\_not\_empty

%type<String> empty\_or\_many\_line\_break

%type<String> one\_similicon\_or\_many\_line\_break

%type<String> many\_line\_break

%type<String> empty\_or\_similicon\_or\_break\_lines

%type<node\_value> expr

%type<stmt\_value> for\_stmt

%type<stmt\_value> body

%type<node\_list\_value> expr\_list

%type<program\_value> program

%type<import\_one\_of\_list\_value> import\_one\_of\_list

%type<import\_decl\_list\_value> import\_decl\_list

%type<import\_list\_value> import\_list

%type<import\_value> import\_v

%type<package\_value> package\_v

%type<complex\_data\_value> complex\_data

%type<node\_list\_value> expr\_list\_not\_empty

%type<decl\_all\_value> const\_v

%type<decl\_all\_list\_value> const\_list\_not\_empty

%type<decl\_all\_list\_value> const\_list

%type<stmt\_value> const\_decl

%type<decl\_all\_value> var\_v

%type<decl\_all\_list\_value> var\_list\_not\_empty

%type<decl\_all\_list\_value> var\_list

%type<stmt\_value> var\_decl

%type<stmt\_value> inc\_dec\_stmt

%type<stmt\_value> assign

%type<stmt\_value> short\_var\_decl

%type<stmt\_value> simple\_stmt\_not\_empty

%type<stmt\_value> return\_stmt

%type<stmt\_value> stmt

%type<stmt\_list\_value> stmt\_list\_not\_empty

%type<stmt\_list\_value> stmt\_list

%type<stmt\_value> for\_stmt\_init\_stmt

%type<stmt\_value> for\_stmt\_post\_stmt

%type<if\_stmt\_block\_value> if\_start\_block

%type<stmt\_value> if\_stmt

%type<if\_stmt\_block\_value> else\_if\_stmt

%type<else\_if\_stmt\_list\_value> else\_if\_stmt\_list

%type<param\_value> param\_decl

%type<param\_list\_value> param\_list

%type<param\_list\_value> params

%type<return\_value> return\_v

%type<highest\_decl\_value> func\_decl

%type<highest\_decl\_value> highest\_decl

%type<highest\_decl\_list\_value> highest\_decl\_list\_not\_empty

%type<highest\_decl\_list\_value> highest\_decl\_list

%type<array\_element\_list\_value> array\_element\_list\_not\_empty

%type<array\_element\_list\_value> array\_element\_list

%type<array\_element\_value> array\_element\_v

%start program

%left LESS GREATER EQUAL NOT\_EQUAL LESS\_OR\_EQUAL GREATER\_OR\_EQUAL

%left '+' '-'

%left '\*' '/'

%right UMINUS

%left '[' ']' '.'

%nonassoc '(' ')'

%%

many\_line\_break: '\n' {$$ = "\n";}

| many\_line\_break '\n' {$$ = $1;}

;

empty\_or\_many\_line\_break: /\* empty \*/ { $$ = 0; }

| many\_line\_break {$$ = $1;}

;

one\_similicon\_or\_many\_line\_break: ';' empty\_or\_many\_line\_break {$$ = ";";}

| many\_line\_break {$$ = $1;}

;

empty\_or\_similicon\_or\_break\_lines: /\* empty \*/ { $$ = 0; }

| one\_similicon\_or\_many\_line\_break {$$ = $1;}

;

program: package\_v one\_similicon\_or\_many\_line\_break highest\_decl\_list { root = create\_program($1, $3); }

| package\_v one\_similicon\_or\_many\_line\_break import\_decl\_list highest\_decl\_list { root = create\_program\_with\_imports($1, $3, $4); }

;

import\_decl\_list: import\_one\_of\_list { $$ = create\_import\_decl\_list($1);}

| import\_decl\_list import\_one\_of\_list { $$ = add\_to\_import\_decl\_list($1, $2);}

;

import\_one\_of\_list: IMPORT\_KEYWORD empty\_or\_many\_line\_break import\_v one\_similicon\_or\_many\_line\_break { $$ = create\_import\_decl($3); }

| IMPORT\_KEYWORD empty\_or\_many\_line\_break '(' empty\_or\_many\_line\_break import\_list empty\_or\_many\_line\_break ')' one\_similicon\_or\_many\_line\_break { $$ = create\_import\_decl\_one\_of\_list\_list($5); }

;

import\_list: import\_v { $$ = create\_import\_list($1); }

| import\_list one\_similicon\_or\_many\_line\_break import\_v { $$ = add\_to\_import\_list($1, $3);}

;

import\_v: '.' empty\_or\_many\_line\_break STRING { $$ = create\_import($3); }

| STRING { $$ = create\_import($1); }

| ID STRING { $$ = create\_import\_with\_alias($1, $2); }

;

package\_v: PACKAGE\_KEYWORD empty\_or\_many\_line\_break ID{ $$ = create\_package\_decl($3); }

;

complex\_data: INT\_KEYWORD { $$ = create\_complex\_data\_type(int\_t); }

| BOOL\_KEYWORD { $$ = create\_complex\_data\_type(bool\_t); }

| BYTE\_KEYWORD { $$ = create\_complex\_data\_type(byte\_t); }

| STRING\_KEYWORD { $$ = create\_complex\_data\_type(string\_t); }

| '[' empty\_or\_many\_line\_break expr ']' complex\_data { $$ = create\_array\_type($3, $5); }

| FUNC\_KEYWORD empty\_or\_many\_line\_break '(' empty\_or\_many\_line\_break param\_list ')' return\_v { $$ = create\_function\_type(create\_prototype(0, $5, $7)); }

| FUNC\_KEYWORD empty\_or\_many\_line\_break '(' empty\_or\_many\_line\_break ')' return\_v { $$ = create\_function\_type(create\_prototype(0, 0, $6)); }

| FUNC\_KEYWORD empty\_or\_many\_line\_break '(' empty\_or\_many\_line\_break param\_list ',' empty\_or\_many\_line\_break ')' return\_v { $$ = create\_function\_type(create\_prototype(0, $5, $9)); }

| FUNC\_KEYWORD empty\_or\_many\_line\_break '(' empty\_or\_many\_line\_break param\_list ')' { $$ = create\_function\_type(create\_prototype(0, $5, 0)); }

| FUNC\_KEYWORD empty\_or\_many\_line\_break '(' empty\_or\_many\_line\_break ')' { $$ = create\_function\_type(create\_prototype(0, 0, 0)); }

| FUNC\_KEYWORD empty\_or\_many\_line\_break '(' empty\_or\_many\_line\_break param\_list ',' empty\_or\_many\_line\_break ')' { $$ = create\_function\_type(create\_prototype(0, $5, 0)); }

;

expr: l\_value { $$ = $1; }

| '[' empty\_or\_many\_line\_break expr ']' complex\_data '{' empty\_or\_many\_line\_break array\_element\_list '}' { $$ = create\_array\_lit($5, $3, $8); }

| INT { $$ = create\_int\_node($1); }

| STRING { $$ = create\_string\_node($1); }

| TRUE\_KEYWORD { $$ = create\_boolean\_node(1); }

| FALSE\_KEYWORD { $$ = create\_boolean\_node(0); }

| '-' empty\_or\_many\_line\_break expr %prec UMINUS { $$ = create\_oper\_node(unary\_minus, $3, 0); }

| expr '-' empty\_or\_many\_line\_break expr { $$ = create\_oper\_node(minus, $1, $4); }

| expr '+' empty\_or\_many\_line\_break expr { $$ = create\_oper\_node(plus, $1, $4); }

| expr '\*' empty\_or\_many\_line\_break expr { $$ = create\_oper\_node(mul, $1, $4); }

| expr '/' empty\_or\_many\_line\_break expr { $$ = create\_oper\_node(divide, $1, $4); }

| expr LESS empty\_or\_many\_line\_break expr { $$ = create\_oper\_node(less, $1, $4); }

| expr GREATER empty\_or\_many\_line\_break expr { $$ = create\_oper\_node(greater, $1, $4); }

| expr GREATER\_OR\_EQUAL empty\_or\_many\_line\_break expr { $$ = create\_oper\_node(greater\_or\_equal, $1, $4); }

| expr LESS\_OR\_EQUAL empty\_or\_many\_line\_break expr { $$ = create\_oper\_node(less\_or\_equal, $1, $4); }

| expr EQUAL empty\_or\_many\_line\_break expr { $$ = create\_oper\_node(equal, $1, $4); }

| expr NOT\_EQUAL empty\_or\_many\_line\_break expr { $$ = create\_oper\_node(not\_equal, $1, $4); }

| ID '(' empty\_or\_many\_line\_break expr\_list ')' { $$ = create\_func\_call($1, $4); }

| expr '.' empty\_or\_many\_line\_break ID '(' empty\_or\_many\_line\_break expr\_list ')' {$$ = create\_method\_use\_in\_package\_node($1, $4, $7); }

;

expr\_list: /\* empty \*/ { $$ = 0; }

| expr\_list\_not\_empty { $$ = $1; }

;

expr\_list\_not\_empty: expr { $$ = create\_node\_list($1); }

| expr\_list\_not\_empty ',' empty\_or\_many\_line\_break expr { $$ = add\_to\_node\_list($1, $4); }

;

const\_v: ID '=' empty\_or\_many\_line\_break expr { $$ = create\_all\_decl(create\_id($1), $4, 0); }

| ID complex\_data '=' empty\_or\_many\_line\_break expr { $$ = create\_all\_decl(create\_id($1), $5, $2); }

;

const\_list\_not\_empty: const\_v { $$ = create\_all\_decl\_list($1); }

| const\_list\_not\_empty one\_similicon\_or\_many\_line\_break const\_v { $$ = add\_to\_all\_decl\_list($1, $3); }

;

const\_list: /\* empty \*/ { $$ = 0; }

| const\_list\_not\_empty one\_similicon\_or\_many\_line\_break { $$ = $1; }

;

const\_decl: CONST\_KEYWORD empty\_or\_many\_line\_break const\_v { $$ = create\_decl\_stmt($3, const\_t); }

| CONST\_KEYWORD empty\_or\_many\_line\_break '(' empty\_or\_many\_line\_break const\_list ')' { $$ = create\_decl\_stmt\_from\_list($5, const\_t); }

;

var\_v: ID complex\_data { $$ = create\_all\_decl(create\_id($1), 0, $2); }

| ID complex\_data '=' empty\_or\_many\_line\_break expr { $$ = create\_all\_decl(create\_id($1), $5, $2); }

| ID '=' empty\_or\_many\_line\_break expr { $$ = create\_all\_decl(create\_id($1), $4, 0); }

;

var\_list\_not\_empty: var\_v { $$ = create\_all\_decl\_list($1); }

| var\_list\_not\_empty one\_similicon\_or\_many\_line\_break var\_v { $$ = add\_to\_all\_decl\_list($1, $3); }

;

var\_list: /\* empty \*/ { $$ = 0; }

| var\_list\_not\_empty empty\_or\_similicon\_or\_break\_lines{ $$ = $1; }

var\_decl: VAR\_KEYWORD empty\_or\_many\_line\_break var\_v one\_similicon\_or\_many\_line\_break{ $$ = create\_decl\_stmt($3, var\_t); }

| VAR\_KEYWORD empty\_or\_many\_line\_break '(' empty\_or\_many\_line\_break var\_list ')' one\_similicon\_or\_many\_line\_break{ $$ = create\_decl\_stmt\_from\_list($5, var\_t); }

;

l\_value: ID { $$ = create\_id\_node($1); }

| '(' empty\_or\_many\_line\_break expr')' { $$ = $3; }

| expr '.' empty\_or\_many\_line\_break ID { $$ = create\_id\_use\_in\_package\_node($1, $4); }

| expr '[' empty\_or\_many\_line\_break expr ']' { $$ = create\_oper\_node(array\_indexing, $1, $4); }

l\_value\_list\_not\_empty: l\_value { $$ = create\_node\_list($1);}

| l\_value\_list\_not\_empty ',' empty\_or\_many\_line\_break l\_value { $$ = add\_to\_node\_list($1, $4); }

;

inc\_dec\_stmt: l\_value INC{ $$ = create\_inc\_dec\_stmt($1, inc\_t); }

| l\_value DEC{ $$ = create\_inc\_dec\_stmt($1, dec\_t); }

;

assign: l\_value\_list\_not\_empty '=' empty\_or\_many\_line\_break expr\_list\_not\_empty { $$ = create\_assignment(simple\_assignment\_t, $1, $4); }

| l\_value\_list\_not\_empty PLUS\_ASSIGN empty\_or\_many\_line\_break expr\_list\_not\_empty { $$ = create\_assignment(plus\_assignment\_t, $1, $4); }

| l\_value\_list\_not\_empty MINUS\_ASSIGN empty\_or\_many\_line\_break expr\_list\_not\_empty { $$ = create\_assignment(minus\_assignment\_t, $1, $4); }

| l\_value\_list\_not\_empty MULTIPLY\_ASSIGN empty\_or\_many\_line\_break expr\_list\_not\_empty { $$ = create\_assignment(mul\_assignment\_t, $1, $4); }

| l\_value\_list\_not\_empty DIVISION\_ASSIGN empty\_or\_many\_line\_break expr\_list\_not\_empty { $$ = create\_assignment(div\_assignment\_t, $1, $4); }

;

short\_var\_decl: ID SHORT\_EQUALS empty\_or\_many\_line\_break expr{ $$ = create\_short\_decl(create\_id($1), $4); }

;

simple\_stmt\_not\_empty: expr { $$ = create\_node\_stmt($1); }

| inc\_dec\_stmt { $$ = $1; }

| assign { $$ = $1; }

| short\_var\_decl { $$ = $1; }

;

return\_stmt: RETURN\_KEYWORD expr\_list { $$ = create\_return\_stmt($2); }

;

stmt: ';' empty\_or\_many\_line\_break {$$ = create\_empty\_stmt();}

|simple\_stmt\_not\_empty one\_similicon\_or\_many\_line\_break { $$ = $1; }

| const\_decl one\_similicon\_or\_many\_line\_break{ $$ = $1; }

| var\_decl{ $$ = $1; }

| return\_stmt one\_similicon\_or\_many\_line\_break{ $$ = $1; }

| if\_stmt one\_similicon\_or\_many\_line\_break{ $$ = $1; }

| for\_stmt one\_similicon\_or\_many\_line\_break{ $$ = $1; }

| body one\_similicon\_or\_many\_line\_break{ $$ = $1; }

| BREAK\_KEYWORD one\_similicon\_or\_many\_line\_break { $$ = create\_break\_stmt(); }

| CONTINUE\_KEYWORD one\_similicon\_or\_many\_line\_break { $$ = create\_continue\_stmt(); }

;

stmt\_list\_not\_empty: stmt { $$ = create\_stmt\_list($1); }

| stmt\_list\_not\_empty stmt { $$ = add\_to\_stmt\_list($1, $2); }

;

stmt\_list: /\* empty \*/ { $$ = 0; }

| stmt\_list\_not\_empty { $$ = $1; }

;

body: '{' empty\_or\_many\_line\_break stmt\_list '}' { $$ = create\_block($3); }

;

for\_stmt\_init\_stmt: ';' { $$ = create\_empty\_stmt(); }

| expr { $$ = create\_node\_stmt($1); }

| inc\_dec\_stmt { $$ = $1; }

| assign { $$ = $1; }

| short\_var\_decl { $$ = $1; }

;

for\_stmt\_post\_stmt: /\* empty \*/ { $$ = create\_empty\_stmt(); }

| expr { $$ = create\_node\_stmt($1); }

| inc\_dec\_stmt { $$ = $1; }

| assign { $$ = $1; }

;

for\_stmt: FOR\_KEYWORD empty\_or\_many\_line\_break body { $$ = create\_empty\_for\_stmt($3); }

| FOR\_KEYWORD empty\_or\_many\_line\_break expr body { $$ = create\_for\_with\_condition($3, $4); }

| FOR\_KEYWORD empty\_or\_many\_line\_break for\_stmt\_init\_stmt one\_similicon\_or\_many\_line\_break expr one\_similicon\_or\_many\_line\_break for\_stmt\_post\_stmt body { $$ = create\_for\_with\_block\_stmt($3, $7, $5, $8); }

;

if\_start\_block: IF\_KEYWORD empty\_or\_many\_line\_break simple\_stmt\_not\_empty one\_similicon\_or\_many\_line\_break expr body { $$ = create\_if\_stmt\_block($3, $5, $6); }

| IF\_KEYWORD empty\_or\_many\_line\_break expr body { $$ = create\_if\_stmt\_block(0, $3, $4); }

;

if\_stmt: if\_start\_block { $$ = create\_if\_stmt($1, 0, 0); }

| if\_start\_block ELSE\_KEYWORD empty\_or\_many\_line\_break body { $$ = create\_if\_stmt($1, 0, $4); }

| if\_start\_block else\_if\_stmt\_list ELSE\_KEYWORD empty\_or\_many\_line\_break body { $$ = create\_if\_stmt($1, $2, $5); }

| if\_start\_block else\_if\_stmt\_list { $$ = create\_if\_stmt($1, $2, 0); }

;

else\_if\_stmt: ELSE\_KEYWORD empty\_or\_many\_line\_break if\_start\_block { $$ = $3; }

;

else\_if\_stmt\_list: else\_if\_stmt { $$ = create\_else\_if\_stmt\_list($1); }

| else\_if\_stmt\_list else\_if\_stmt { $$ = add\_to\_else\_if\_stmt\_list($1, $2); }

;

param\_decl: ID complex\_data { $$ = create\_param($2, create\_id($1)); }

| complex\_data { $$ = create\_param($1, 0); }

;

param\_list: param\_decl { $$ = create\_param\_list($1); }

| param\_list ',' empty\_or\_many\_line\_break param\_decl { $$ = add\_to\_param\_list($1, $4); }

;

params: '(' empty\_or\_many\_line\_break param\_list ')' { $$ = $3; }

| '(' empty\_or\_many\_line\_break ')' { $$ = 0; }

| '(' empty\_or\_many\_line\_break param\_list ',' empty\_or\_many\_line\_break ')' { $$ = $3; }

;

return\_v: params { $$ = create\_return\_with\_values($1); }

| complex\_data { $$ = create\_type\_of\_return($1); }

;

func\_decl: FUNC\_KEYWORD empty\_or\_many\_line\_break ID params return\_v one\_similicon\_or\_many\_line\_break { $$ = create\_func\_decl(create\_prototype($3, $4, $5), 0); }

| FUNC\_KEYWORD empty\_or\_many\_line\_break ID params return\_v body one\_similicon\_or\_many\_line\_break { $$ = create\_func\_decl(create\_prototype($3, $4, $5), $6); }

| FUNC\_KEYWORD empty\_or\_many\_line\_break ID params one\_similicon\_or\_many\_line\_break{ $$ = create\_func\_decl(create\_prototype($3, $4, 0), 0); }

| FUNC\_KEYWORD empty\_or\_many\_line\_break ID params body one\_similicon\_or\_many\_line\_break { $$ = create\_func\_decl(create\_prototype($3, $4, 0), $5); }

;

highest\_decl: const\_decl one\_similicon\_or\_many\_line\_break{ $$ = create\_highest\_declaration($1->decl\_stmt\_field); }

| var\_decl{ $$ = create\_highest\_declaration($1->decl\_stmt\_field); }

| func\_decl { $$ = $1; }

;

highest\_decl\_list\_not\_empty: highest\_decl { $$ = create\_highest\_decl\_list($1); }

| highest\_decl\_list\_not\_empty highest\_decl { $$ = add\_to\_highest\_decl\_list($1, $2); }

;

highest\_decl\_list: /\* empty \*/ { $$ = 0; }

| highest\_decl\_list\_not\_empty { $$ = $1; }

;

array\_element\_list: /\* empty \*/ { $$ = 0; }

| array\_element\_list\_not\_empty { $$ = $1; }

;

array\_element\_list\_not\_empty: array\_element\_v { $$ = create\_array\_element\_list($1); }

| array\_element\_list\_not\_empty ',' empty\_or\_many\_line\_break array\_element\_v { $$ = add\_to\_array\_element\_list($1, $4); }

;

array\_element\_v: expr { $$ = create\_array\_element($1); }

| INT ':' empty\_or\_many\_line\_break expr { $$ = create\_array\_indexed\_element($1, $4); }

;

%%

Выводы

Изучены особенности синтаксиса и семантики статически типизированного языка Golang. Разработаны лексический и синтаксический анализаторы кода.