

گزارش کار پروژه کامپایلر

یاسمن گودرزی - ۹۹۳۱۱۰۰

در ابتدا گرامر داده شده در دستور کار را به فرمت **antlr** در میاوریم و شرط های که در دستور کار بیان شده را در گرامر رعایت میکنیم


```
grammar Grammer;

// Lexer rules
STRINGLIST : '"' (ESC | ~["\\r\n])* '"' ;
ESC : '\\' . ;
INTEGER : '0' | [1-9] [0-9]*;
DOUBLE : INTEGER '.' [0-9]+ ([eE] [+|-]? [0-9]+)?;
ID : [a-zA-Z_] [a-zA-Z_0-9]*;

prog : func_list ;

func_list : func_def func_list
          | func_def
          ;

func_def : data_type ID '(' param_list? ')' code_block
          ;

param_list : param ',' param_list
           | param
           | /* empty */
           
           ;

param : data_type ID ;

data_type : 'int' | 'double' | 'boolean' | 'void' ;

code_block : '{' stmt_list '}' ;

stmt_list : stmt stmt_list
          | /* empty */
          ;
```

```
stmt : ';'
      | code_block
      | data_type var_list ';'
      | ID '=' expr ';'
      | ID '++' ';'
      | ID '--' ';'
      | 'return' ';'
      | 'return' expr ';'
      | loop_stmt
      | decide '(' expr ')' stmt ('else' stmt)?
      | expr ';'

      ;

decide : 'if' ;

loop_stmt : 'while' '(' expr ')' stmt
           ;

init_stmt : data_type ID '=' expr
           | ID '=' expr
           | /* empty */
           ;

post_stmt : ID '=' expr
           | ID '++'
           | ID '--'
           | /* empty */
           ;

var_list : var (',' var)*
         ;
```

```

var : ID
    | ID '=' expr
    ;
expr : number
    | ID
    | 'true'
    | 'false'
    | STRINGLIST
    | ID '(' args ')'
    | '(' expr ')'
    | unop expr
    | expr ( '*' | '/' | '%' ) expr
    | expr ( '+' | '-' ) expr
    | expr ( '<' | '>' | '<=' | '>=' ) expr
    | expr ( '==' | '!=' ) expr
    | expr and expr
    | expr or expr
    ;
and : '&&' ;
or : '||' ;

args : expr ( ',' expr )*
    | /* empty */
    ;
unop : '-' | '!' ;
number : INTEGER | DOUBLE;

Whitespace: [ \t\r\n]+ -> skip ;
Newline: '\r'? '\n' -> skip ;
COMMENT : '//' ~[\r\n]* -> skip;
MULTILINE_COMMENT : '/*' .*? '*/' -> skip;

```

گرامر رو با استفاده از antlr ران میکنیم
رعایت اولویت بندی :

```

expr : number
      | ID
      | 'true'
      | 'false'
      | STRINGLIST
      | ID '(' args ')'
      | '(' expr ')'
      | unop expr
      | expr ( '*' | '/' ) expr
      | expr ( '+' | '-' ) expr
      | expr ( '<' | '>' | '<=' | '>=' ) expr
      | expr ( '==' | '!=' ) expr
      | expr 'and' expr
      | expr 'or' expr

```

برای رعایت کردن اولویت ها گرامر را به صورت بالا اصلاح کردیم. در گرامر قوانین به ترتیب اجرا و چک میشوند یعنی ابتدا قانونی که شامل \times و \div بررسی میشود و سپس دستور مربوط به $+$ و $-$ بررسی میشود و وقتی با قانونی مچ بشود جایگزین میشود. با توجه به این نکته قانون ها را به ترتیب اولویتی که عملگر هاشون دارن اصلاح میکنیم. یعنی ابتدا قانون ضرب و تقسیم را مینویسم و سپس قانون مربوط به جمع و تفریق را می نویسیم.

هندل کردن ارور `dangling else` :

هنگامی که ما `if` های تودرتو داشته باشیم این ارور و مشکل ایجاد میشود. در حالت `nested if` وقتی چندتا `if` پشت سرهم تعریف شود `else` را به ابهام برمیخورد که این `else` مخربوط به کدام `if` و شرط است. برای هندل کردن این ارور روش های گوناگونی وجود دارد که یکی از این روش ها `nearest if` نام دارد. در این متد میایم `else` را به نزدیک ترین و اولین `if` قبل از این `else` نسبت میدهیم. در واقع هر `else` به نزدیکترین `if` ربط دارد.

```

| decide (< expr >) < stmt >
| decide (< expr >) < stmt > else < stmt >

```

در گرامر داده شده به علت وجود این دو خط ابهام رخ میداد. این دو خط را به صورت زیر اصلاح می کنیم.

```

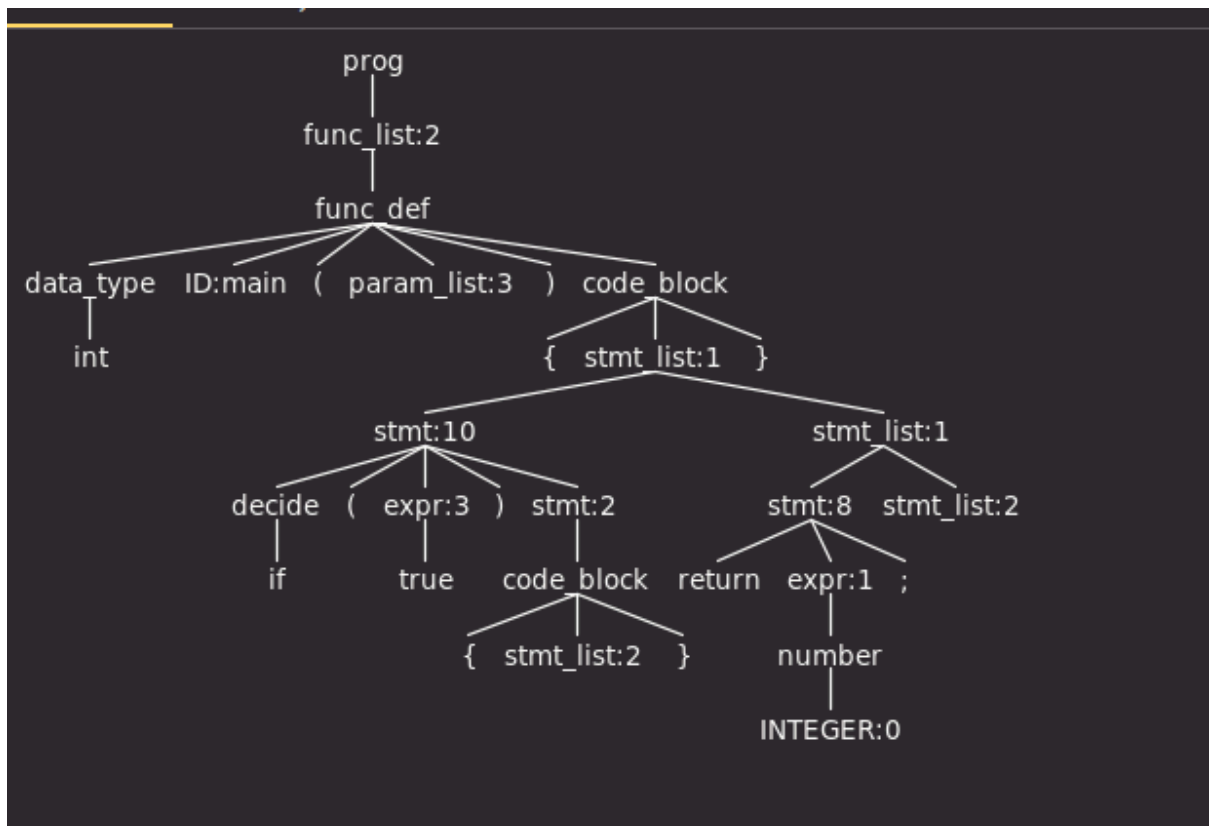
| 'if' '(' expr ')' stmt ('else' stmt)?
| expr ';'
| 'if' '(' expr ')' stmt

```

در این گرامر ما اولویت رو به دستوراتی دادیم که شامل `ELSE` باشند و در این حالت اگر در یک عبارت `if` وجود داشته باشد همیشه اولویت با `else` است که در ادامه میاید. به عبارت دیگر یک `if` همیشه می تواند یک `else` داشته

باشد و این `else` به نزدیک‌ترین `if` تعلق دارد. این به این دلیل است که `(else' stmt)?` در انتهای قاعده‌ی `if` آمده است و به وضوح بیان می‌کند که `else` به همان `if` مرتبط است.

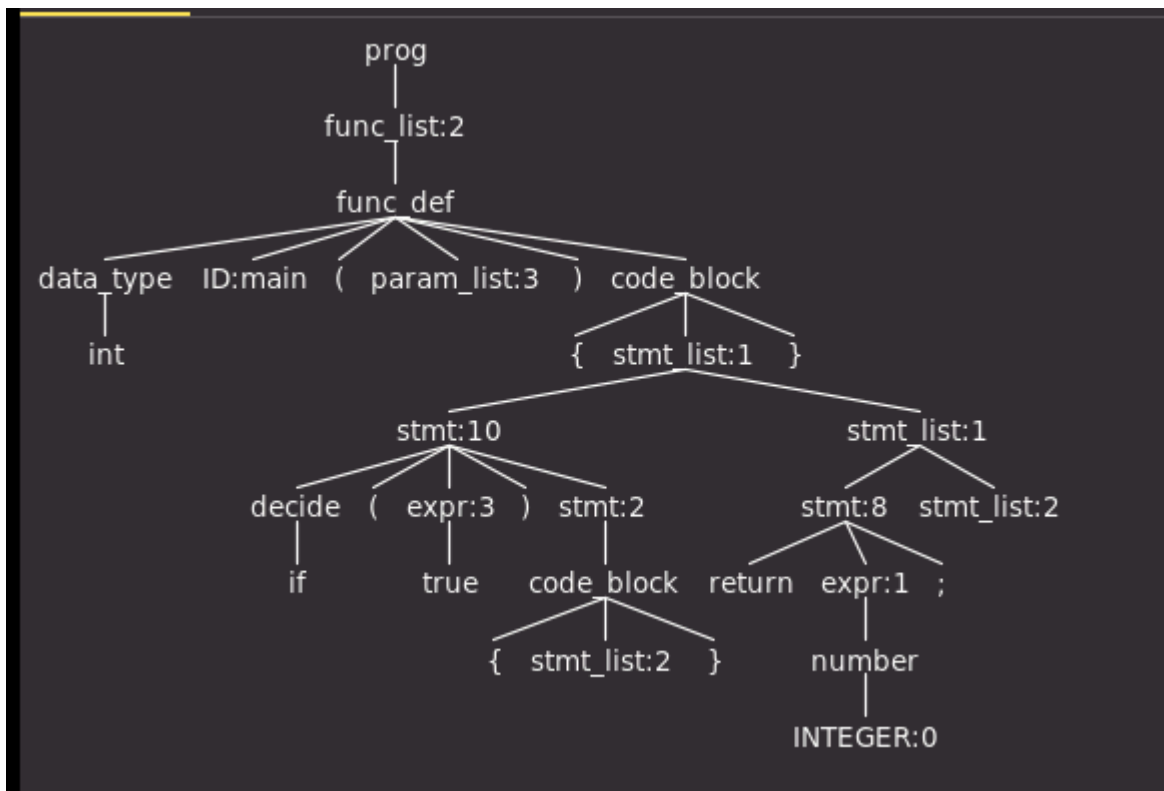
توضیح دوتا تست کیس



```
int main() {
    if (true){
        return 0;
    }
}
```

تست کیس ۲۳:

در این تست کیس کد ما شامل یک `if` همیشه درست هست. پس درخت آن ابتدا به صورت یک `int` و ادامه تابع و `code block` استفاده شده نیز به صورت درخت `if` شکسته شده است. در هنگام استفاده از این تست کیس ما در گرامر کلمه `decide` رو داشتیم اما احتیاج به یک `if` داشتیم پس گرامر را تغییر دادیم و این کلمه رو اضافه کردیم. خروجی کد ۰ است.



```

1 // Calling functions which take zero parameters
2
3 int main() {
4   int x = foo();
5   printInt(x);
6   return 0 ;
7 }
8
9
10 int foo() {
11   return 10;
12 }
13

```

در این تست کیس گرامر از ریشه شروع به ساختن قوانین میکند. از بین قواعد مربوط به `func_list` قاعده `func_def` استفاده شده است. قواعد به ترتیب ورودی های داده شده استفاده شده اند. در اینجا قواعد مربوط به `func_def` به ترتیب اجرا میشوند. همچنین این مورد نوع پیاده سازی با اولویت ها رو هم هندل میکند. در نهایت نیز 10 پرینت میشود.

با اجرای کامند فایل های مناسب برای اجرای گرامر ساخته میشود. اما در اینجا به توضیح سه فایل ساخته شده میپردازیم

lexer

این فایل برای lexical analyzer تحلیل درست میشود. در واقع رشته ورودی را میگیرد و توکن های را می سازد این توکن ها کوچک اجزای زبان تجزیه هستند. این توکن ها به عنوان ورودی های پارسر به پارسر فرستاده میشوند.

parser

این فایل ورودی های توکن را میگیرد و تجزیه بر روی آنها را شروع میکند. در نهایت با تجزیه بر روی توکن ها درخت تجزیه آن را می سازد که نشان دهنده ساختار نحوی ورودی است. در نهایت این فایل قواعد مربوط به دستور زبان در فایل antlr را اجرا میکند

listener


این فایل برای پیمایش در درخت و انجام وظایفی حین این پیمایش طراحی شده است. در واقع وظیفه این فایل و توابع درونش این است که حین پیمایش روی درخت parser عملیات های مورد نیاز برای هر گره را در حین ورود و خروج به آن گره پیاده سازی کنند. پیاده سازی عملیات های درون این فایل بر عهده خودمان است و از آن برای گرامر های پیچیده استفاده می کنیم.

برای ارتباط این سه بخش نیز از کد زیر استفاده خواهیم کرد

```
import sys
from antlr4 import *

from Compiler.gen.GrammerLexer import GrammerLexer
from Compiler.gen.GrammerListener import GrammerListener
from Compiler.gen.GrammerParser import GrammerParser

def main(argv):
    input_stream = InputStream("""int main () {
    int b=2;
    }""")
    lexer = GrammerLexer(input_stream)
    tokens_stream = CommonTokenStream(lexer)
    parser = GrammerParser(tokens_stream)
    tree = parser.prog()
    printer = GrammerListener()
    walker = ParseTreeWalker()
    walker.walk(printer, tree)
    # print(tree.code)

> if __name__ == '__main__':
    main(sys.argv)
```