

2021-1 Compiler class 02 ( Prof. Hyo su Kim )

## Project #2 Syntax Analyzer

2021.06.03



### Student (Team 16)

Software department

Team members : 배인경(20190323) 좌민주(20190912)

# Content

## 1. non-ambiguous CFG G

- 1) Discard an ambiguity in the CFG
- 2) CFG G

## 2. SLR TABLE

## 3. IMPLEMENTATION

- 1) Lexical Analyzer (DFA.py, lexical\_analyzer.py )
- 2) Syntax Analyzer (rules.py, SLR.py, syntax\_analyzer.py )

## 4. ERROR REPORT

## 5. TEST INPUT & OUTPUT

## 6. GITHUB & Google Docs & KakaoTalk & Zoom

## 7. LINUX

## #1 non-ambiguous CFG G

### Discard an ambiguity in the CFG ①

#### **ambiguous**

05:EXPR → EXPR addsub EXPR | EXPR multdiv EXPR

06:EXPR → lparen EXPR rparen | id | num

#### **non-ambiguous**

EXPR → TERM addsub EXPR | TERM

TERM → FACTOR multdiv TERM | FACTOR

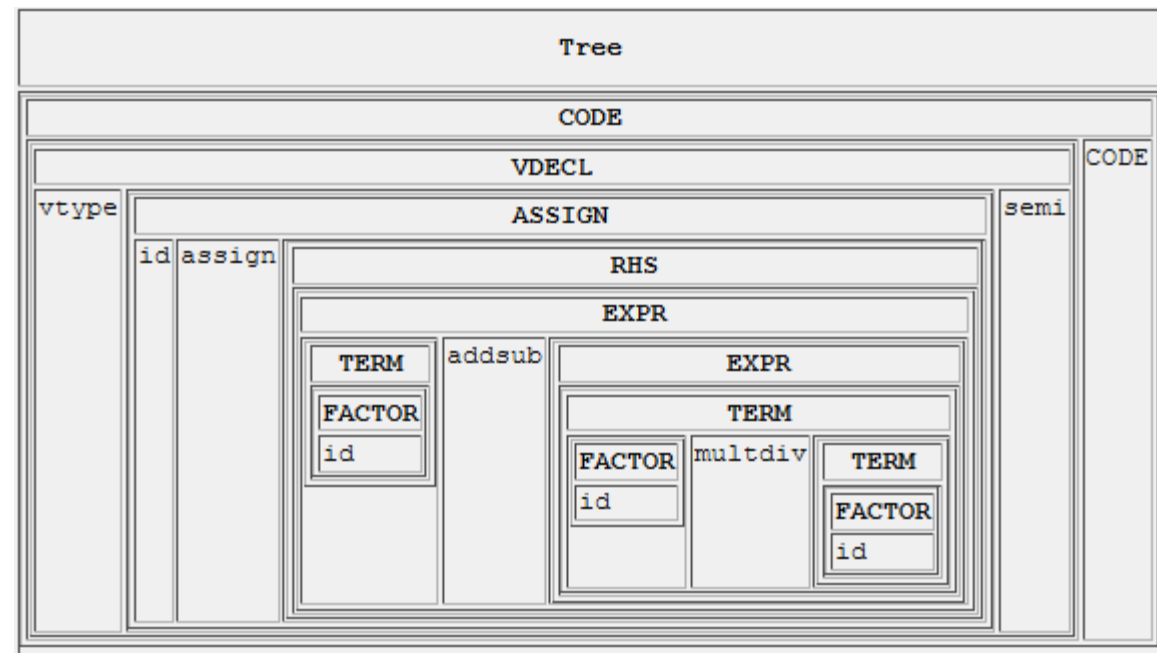
FACTOR → lparen EXPR rparen | id | num

#### **Disambiguating rule**

**“ multdiv 가 addsub 보다 높은 우선순위를 갖는다”**

연산자 우선순위를 두어 모호성을 없앴다. 이를 위해 TERM과 FACTOR라는 non terminal을 추가하였다. TERM은 multdiv와 FACTOR로 production된다. FACTOR는 괄호가 있는 EXPR와 terminal들로 production 되어서 만약에 addsub연산자를 먼저 계산해야할 경우 괄호에 싸여 높은 우선순위를 갖게 된다. 다음과 같이 parse tree가 하나만 생성된다. multdiv가 먼저 계산되고, 그 결과가 addsub의 피연산자가 되는 것을 볼 수 있다.

Input (tokens): vtype id assign id addsub id multdiv id semi



### Discard an ambiguity in the CFG ②

#### **ambiguous**

14:COND → COND comp COND | boolstr

### **non-ambiguous**

$COND \rightarrow T \text{ comp } T \mid T$

$T \rightarrow \text{lparen } COND \text{ rparen} \mid \text{boolstr}$

### **Disambiguating rule**

예를 들어 JAVA 문법에서 **boolean result = ((1<2) == true);** 를 실행하면 괄호 안에 식인 1<2 연산을 먼저 수행 해 true로 판단하고 좌항의 결과값이 true로 대입되어, true == true 가 결론적으로 계산 되어 result 변수값은 true가 된다. 그러므로 우리도 'boolstr comp boolstr' 에서 끝나지 않고 연산자가 한 번에 여러 개 사용되는 경우를 고려해줘야 할 것이라는 결론을 내렸다.

#### **“괄호 안에 있는 comp 가 우선순위를 갖는다”**

JAVA 문법에서 true > true > true를 어떻게 받아들일지 가정하였다. 순서대로 계산하도록 CFG를 만들었을 때는, 위와 같은 형태의 식은 JAVA 언어에 없는 문법이었다. 따라서 괄호로 우선순위를 만들었다. ( true > true ) > true 든 true > ( true > true ) 든 괄호 안의 comp를 계산하고, 그 결과는 괄호 바깥의 comp의 피연산자가 된다. addsub 와 multdiv의 모호성을 없애기 위해 넣은 FACTOR처럼 T를 넣어 괄호로 우선순위를 주었다.

$T \rightarrow \text{lparen } COND \text{ rparen} \mid \text{boolstr}$

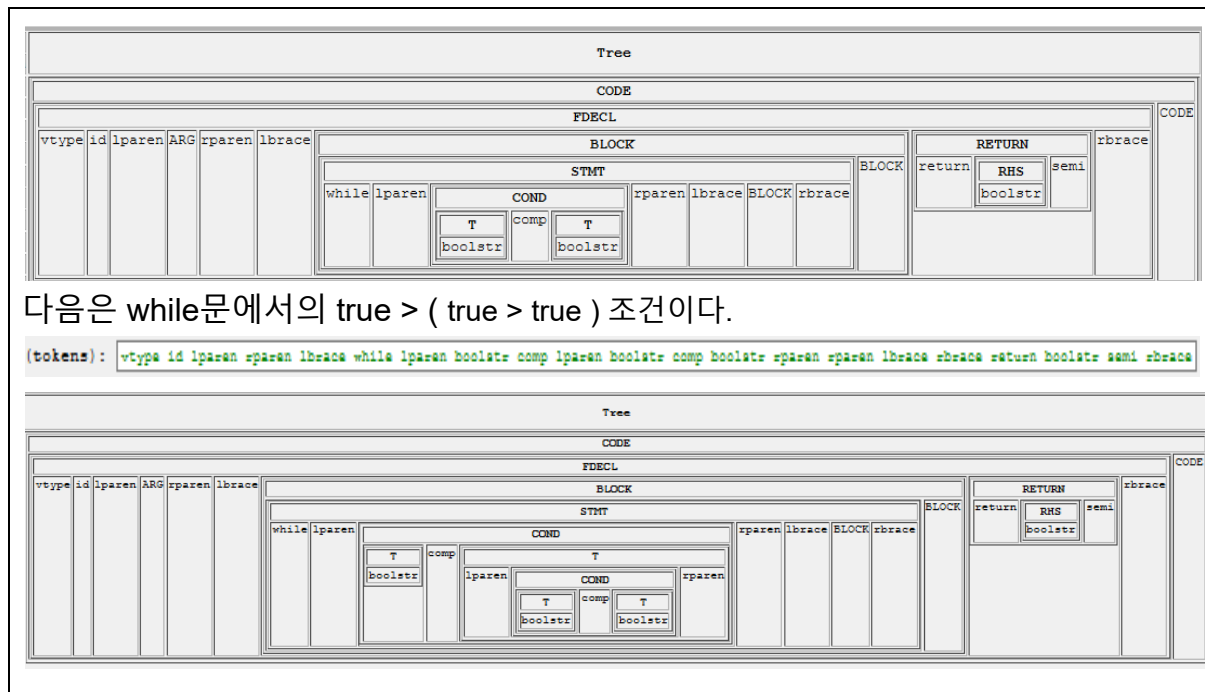
또한 COND는  $T \text{ comp } T \mid T$ 로 production된다.  $T \text{ comp } T$ 인 이유는 comp 연산자는 2개의 피연산자가 boolstr 인 연산자이기 때문이다. 반드시 T는 boolstr로 변환된 후 comp 연산을 하게 된다. 이 규칙을 적용한다면 true > true > ( true > true )라는 조건식은 가운데 '>' 를 기준으로 보면 '비교문( true > true ) > boolstr' 이 되기 때문에 문법적으로 틀린 문장이 된다. 어떤 언어에서도 한 번에 세 변수를 비교할 수는 없다. 따라서 이와 같은 입력의 모호성도 없어진다. 따라서 ( true > true ) > ( true > true ) 와 같은 입력이 올바르다.

$COND \rightarrow T \text{ comp } T \mid T$

( COND → COND comp T | T 인가 COND → T comp T | T 인가에 대한 고민을 하다가 COND → T comp T | T로 확정을 지은 이유는 COND → COND comp T | T로 SLR TABLE을 구성할 시 true > true > (true < false) 처럼 한 번에 세 변수를 비교하는 경우도 맞다고 처리한다. ( 실제 JAVA에서 == 은 여러 boolean 타입 변수를 한번에 비교하는 것이 가능하지만, 그 외 연산자(>, <, <=, >=)는 논리 연산자를 사용하지 않으면 오직 두 개의 피연산자를 비교하는 것만 가능하므로 > 에 맞추었다.) 따라서 괄호 없이 세변수의 comp 연산은 옳지 않으므로 COND → T comp T | T 가 non-ambiguous한 표현이라고 생각하였다. )

다음은 while문에서의 true > true 조건이다.

Input (tokens): `<vtype id lparen xparen lbrace while lparen boolstr comp boolstr xparen lbrace rbrace return boolstr semi rbrace`



## CFG G

START → CODE

CODE → VDECL CODE | FDECL CODE | CDECL CODE | "

VDECL → vtype id semi | vtype ASSIGN semi

ASSIGN → id assign RHS

RHS → EXPR | literal | character | boolstr

EXPR → TERM addsub EXPR | TERM

TERM → FACTOR multdiv TERM | FACTOR

FACTOR → lparen EXPR rparen | id | num

FDECL → vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace

ARG → vtype id MOREARGS | "

MOREARGS → comma vtype id MOREARGS | "

BLOCK → STMT BLOCK | "

STMT → VDECL | ASSIGN semi

STMT → if lparen COND rparen lbrace BLOCK rbrace ELSE

STMT → while lparen COND rparen lbrace BLOCK rbrace

COND → T comp T | T

T → lparen COND rparen | boolstr

ELSE → else lbrace BLOCK rbrace | "

RETURN → return RHS semi

CDECL → class id lbrace ODECL rbrace

ODECL → VDECL ODECL | FDECL ODECL | "

\* "는 *epsilon*을 뜻함

## #2 SLR TABLE

[illegible]

이미지의 원본 크기가 너무 크므로 github에 올린 이미지의 주소(아래 url)로 접속하시면  
고화질로 확인하실 수 있습니다.

<https://user-images.githubusercontent.com/65646971/120525946-a5cc8980-c413-11eb-981a-0fee7ce50c8d.png>

## SLR grammar

rules.py

### # RULES 딥서너리 생성 ( 예시 형태 )

```

1  RULES = {'0': 'START -> CODE',
2          '1': 'CODE -> VDECL CODE',
3          '2': 'CODE -> FDECL CODE',

```

#### # 전체 rules

```

START -> CODE
CODE -> VDECL CODE
CODE -> FDECL CODE
CODE -> CDECL CODE
CODE -> epsilon
VDECL -> vtype id semi
VDECL -> vtype ASSIGN semi
ASSIGN -> id assign RHS
RHS -> EXPR
RHS -> literal
RHS -> character
RHS -> boolstr
EXPR -> TERM addsub EXPR
EXPR -> TERM
TERM -> FACTOR multdiv TERM
TERM -> FACTOR
FACTOR -> lparen EXPR rparen
FACTOR -> id
FACTOR -> num
FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
ARG -> vtype id MOREARGS
ARG -> epsilon
MOREARGS -> comma vtype id MOREARGS
MOREARGS -> epsilon
BLOCK -> STMT BLOCK
BLOCK -> epsilon
STMT -> VDECL
STMT -> ASSIGN semi
STMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE
STMT -> while lparen COND rparen lbrace BLOCK rbrace
COND -> T comp T
COND -> T
T -> lparen COND rparen
T -> boolstr
ELSE -> else lbrace BLOCK rbrace
ELSE -> epsilon
RETURN -> return RHS semi
CDECL -> class id lbrace ODECL rbrace
ODECL -> VDECL ODECL
ODECL -> FDECL ODECL
ODECL -> epsilon

```

### #3 IMPLEMENTATION

#### SLR.py

##### # SLR table 생성

```
SLR_TABLE = [
    {
        'vtype': 's5', 'class': 's6', '$': 'r4',
        'CODE': 1, 'VDECL': 2, 'FDECL': 3, 'CDECL': 4
    },
    { ... },
    ...
]
```

		LR table																	
		ACTION																	
State		vtype	id	semi	assign	literal	character	boolstr	addsub	multdiv	lparen	rparen	num	lbrace	rbrace	comma	if	while	comp
0		s5																	
1																			

		GOTO																	
START		CODE	VDECL	ASSIGN	RHS	EXPR	TERM	FACTOR	FDECL	ARG	MOREARGS	BLOCK	STMT	COND	T	ELSE	RETURN	CDECL	ODECL
		1	2						3									4	

리스트에 딕셔너리 형태로 넣었다. 리스트의 index는 state가 된다. 각 state 안에서, 딕셔너리의 키가 input symbol이 되고, 딕셔너리의 value는 shift, reduce, goto 중 3가지 종류의 상태가 된다. 총 0~89의 state가 나왔다.

#### syntax\_analyzer.py

# lexical\_analyzer.py를 실행하여 얻은 symbol table을 input으로 받음  
연속된 토큰 값을 파일 입출력을 통해서 입력으로 받는다.

```
6 # file I/O
7 f = open(sys.argv[1], 'r')
8 inputline = f.readlines()
9 filename = sys.argv[1]
```

##### # input 파일을 읽어 input symbol을 생성

```
17 terminal_list = []
18 err = 0
19 for inputStr in inputline:
20     if inputStr[0] == "E":
21         filewrite(file_out, "ERROR : input file(Lexical Analysis Result) error")
22         err = 1
23         break
24     if inputStr[0] == "[":
```

lexical analyzer는 2종류의 파일을 생성한다. 하나는 lexical analyzer의 Error report이고, 하나는 연속되는 토큰이다. 따라서 if문으로 2가지 상황을 구분하였다.



- 1) 검사할 코드가 올바른 lexical이었다면 ['TYPE', 'int']와 같은 형태가 넘어온다. 모든 토큰을 읽으면서 for문을 반복한다. syntax analyzer을 검사하기 위해 symbol table을 바탕으로 token에 따라 terminal로 구성된 input을 생성한다.

- terminal\_list 변수에 terminal input symbol들이 담긴다.

```
24     if inputStr[0] == "[":
25         list_str = inputStr.split("")
26         if list_str[1] == "IDENTIFIER":
27             terminal_list.append("id")
28         elif list_str[1] == "INTEGER":
29             terminal_list.append("num")
```

- token <OPERATOR>는 lexeme에 따라 terminal이 addsub, multdiv로 나뉜다.

```
50     elif list_str[1] == "OPERATOR":
51         if list_str[3] == "+" or list_str[3] == "-":
52             terminal_list.append("addsub")
53         elif list_str[3] == "*" or list_str[3] == "/":
54             terminal_list.append("multdiv")
```

- token <KEYWORD>는 lexeme에 따라 terminal이 if, else, while, class, return으로 나뉜다.

```
59     elif list_str[1] == "KEYWORD":
60         if list_str[3] == "if":
61             terminal_list.append("if")
62         elif list_str[3] == "else":
63             terminal_list.append("else")
64         if list_str[3] == "while":
65             terminal_list.append("while")
66         elif list_str[3] == "class":
67             terminal_list.append("class")
68         elif list_str[3] == "return":
69             terminal_list.append("return")
```

- 예외 처리) lexical analyzer 구현 시에는 [ (LARRAY) 와 ] (RARRAY) 를 올바른 token 및 lexeme 처리를 해 주었지만, syntax analyzer 구현 시에는 초기 조건에 해당 terminal이 없어서 해당 토큰에 대한 terminal을 정의하지 않았다. 검사한 java 코드가 올바른 lexical이지만 syntax에서 정의한 terminal이 아닐 때 Error를 반환하고 syntax 검사를 진행하지 않는다.

```
70     else:
71         file_out.close()
72         file_out = open(f"{filename.replace('.out', '')}_final.out", 'w')
73         printStr = "ERROR : not match type : " + list_str[1]
74         filewrite(file_out, printStr)
75         filewrite(file_out, '\n')
76         break
```

- 2) 올바르지 않았다면 ERROR: is not match type, line:1가 넘어온다. lexical에서 Error가 발생해 symbol table이 전달되지 않았다는 메시지를 나타낸다.

### # END MARK와 STACK 생성

```
78     terminal_list.append(END_MARK)
79     print(terminal_list)
80
81     now_stack = [0]
82     position = 0
```

- 1) 코드를 terminal로 치환한 input symbol의 끝에 '\$(end mark)'를 덧붙인다. 이를 통해 모든 input을 읽었음을 알 수 있다. SLR table에도 \$가 있어 acceptance를 알려주는 input으로 사용된다.
- 2) 매 과정마다 viable prefix인지 검사가 있는 SLR 파싱을 효과적으로 구현하기 위해 stack 자료구조를 사용한다. slr parsing에서 스택의 처음에는 상태 1을 나타내는 1값이 들어있어야한다. 컴퓨터 언어는 0부터 시작하므로 상태1을 나타내는 0이 들어있다.

### # SLR parsing

```
86     while (err == 0):
87         k += 1
88         print("!!!", k)
89
90         # current state
91         current_state = now_stack[-1]
92
93         # next input symbol
94         next_symbol = terminal_list[position]
95         print(next_symbol, "|", current_state)
96         # next symbol이 SLR_TABLE에 있는 지 체크
97         if next_symbol not in SLR_TABLE[current_state].keys():...
110
111
112         # shift
113         if (SLR_TABLE[current_state][next_symbol][0] == 's'):
114             position = position + 1
115             now_stack.append(int(SLR_TABLE[current_state][next_symbol][1:]))
116
117         # reduce
118         elif (SLR_TABLE[current_state][next_symbol][0] == 'r'):...
155
156         elif (SLR_TABLE[current_state][next_symbol] == 'acc'):...
```

input symbol의 terminal들을 하나씩 읽으면서 SLR 파싱을 하는 알고리즘이다.

1. 스택에서 현재 state를 pop한다.
2. 다음 입력을 확인한다.
3. SLR 테이블에서 현재 state에서 다음 입력을 보고 shift, reduce, goto 가 있는지 확인한다.

이 과정을 accept이 될 때까지 반복한다. 또는, 현재 state에서 다음 입력을 보고 취할 수 있는게 없다면 error를 발생시킨다.

### # SLR parsing -detail 1) shift

```

112     # shift
113     if (SLR_TABLE[current_state][next_symbol][0] == 's'):
114         position = position + 1
115         now_stack.append(int(SLR_TABLE[current_state][next_symbol][1:]))

```

position은 현재 terminals에서 |의 왼쪽 문자들의 suffix의 index이다. shift를 하면 다음 input을 읽기 때문에 position이 1 증가한다. 그리고 스택에는 현재의 상태에서 다음 input symbol을 읽었을 때의 state를 push하고, 다음 입력을 받는다. 다시 반복문을 돌 때는 stack에서 top이 가리키는 state로 goto한다.

## # SLR parsing -detail 2) reduce

```

116     # reduce
117     elif (SLR_TABLE[current_state][next_symbol][0] == 'r'):
118         string_check = SLR_TABLE[current_state][next_symbol][1:]
119
120         rule_check = RULES[string_check].split()
121         rule_check_len = len(rule_check) - 2
122
123         # terminal list 확인
124         for i in range(rule_check_len):
125             if (rule_check[2] != 'epsilon'):...
129         print("terminal", terminal_list)
130         if (rule_check[2] != 'epsilon'): # if not epsilon
131             position = position - rule_check_len + 1
132         else:
133             # epsilon일 때
134             position = position + 1
135         # terminal list 확인
136         terminal_list.insert(position - 1, rule_check[0])
137         current_state = now_stack[-1]
138         # print(terminal_list)
139
140         if rule_check[0] not in SLR_TABLE[current_state].keys():...
152
153         now_stack.append(SLR_TABLE[current_state][rule_check[0]])

```

만약 현재 상태에서 다음 input을 보고 SLR 테이블이 'r')이라면 reduce를 하게 된다. 즉, prefix가 viable prefix라는 것이다. r뒤에 적힌 숫자는 reduction을 하기 위해 사용되는 rule의 번호이다.

- T -> a 라는 rule이라면 a의 길이가 rule\_check\_len이다. 따라서 그 길이만큼 reduction이 됐기 때문에 스택을 pop 시킨다.

```

123     for i in range(rule_check_len):
124         if (rule_check[2] != 'epsilon'): # if not epsilon
125             # pop out from stack
126             now_stack.pop()
127             terminal_list.pop(position - i - 1)

```

- 그리고 reduction을 하기 위해 현재 terminal에서 production 룰을 거꾸로 적용시킨다. terminal\_list.pop()을 하고, 그 자리를 terminal\_list.insert()로 reduce를 한다. 즉 production rules의 RHS를 LHS로 바꾸는 것이다.

```
152 now_stack.append(SLR_TABLE[current_state][rule_check[0]])
```

- 현재 상태는 스택의 top이 가리키고 있다. 따라서 현재 상태에서 GOTO 테이블을 보고 reduction할 때 사용한 rules의 LHS가 가리키는 상태를 stack에 넣는다.

#### # SLR parsing -detail 2) acceptance

```
154 elif (SLR_TABLE[current_state][next_symbol] == 'acc'):  
155     file_out.close()  
156     file_out = open(f"{filename.replace('.out', '')}_final.out", 'w')  
157     printStr = "ACCEPT"  
158     print("ACCEPT 1")  
159     err = 1  
160     filewrite(file_out, printStr)  
161     filewrite(file_out, '\n')  
162     break
```

모든 input symbol을 다 읽고 \$를 만났을 때, acc라면 syntax가 맞다. 최종적으로 dummy start 인 START 룰로 reduction 된다.

#### # syntax\_analyzer.py를 통과한 input의 Accept/Reject 결과를 파일로 출력

```
11 file_out = open(f"{filename.replace('.out', '')}_final.out", 'w')  
12  
13 def filewrite(file, string):  
14     file.write(string)
```

주어진 symbol table의 syntax가 맞으면 Accept, 오류가 있으면 Error Report의 메시지가 남는다.

## #4 ERROR REPORT

`syntax_analyzer.py`에서 `syntax`가 `accept` 되지 않는 상황은 `error`이다.

- error 0: lexical에서 `Error`가 발생해 `symbol table`이 전달되지 않았음
- error 1: simple Java programming language 에 속하지 않는 `token`이 있음
- error 2: ACTION table에서 입력을 보고 다음 상태로 넘어갈 수 없이 끝남

`error` 중 `syntax`와 관련된 에러인 `error 2`는 `ERROR`를 띄우고 왜, 어디서 오류가 났는지를 알려준다. `while`문을 돌다가 `accept`이 되지 않은 상태에서 `state`를 이동할 수 없다면 `error`가 생긴다. 즉, SLR table에서 공백 칸을 만났을 때이다.

### # Error 2: SLR parsing) ACTION Table error message

```
103     # next symbol이 SLR_TABLE에 있는 지 체크
104     if next_symbol not in SLR_TABLE[current_state].keys():
105         file_out.close()
106         file_out = open(f"{filename.replace('.out', '')}_final.out", 'w')
107         print("REJECT")
108         err = 1
109         # ---Error message--- #
110         printStr = "ERROR 2: Token number ({}).format(token_index) + inputline[token_index-1]
111         message = "SyntaxError: Missing a terminal corresponding to one " \
112                 "of the following: (token line {}).format(token_index)
113         # 올바른 syntax가 되기 위해 필요한 아이템
114         missing_item = [item for item in list(SLR_TABLE[current_state].keys())
115                        if item != END_MARK and item.islower()]
116         missing = "{}.format(missing_item)
117
118         print(printStr, end='') # 터미널 창에 출력
119         print(message)
120         print(missing)
121
122         filewrite(file_out, printStr)~# file out
123         filewrite(file_out, message)
124         filewrite(file_out, '\n')
125         filewrite(file_out, missing)
126         filewrite(file_out, '\n')
127         # ---End of the Error message--- #
128         break
```

만약 ACTION 테이블에서 현재 상태에서 다음 입력을 보고 `shift`나 `reduction`을 하지 못하고 끝이 난다면 `error`이다. 이를 해결하기 위해 **# Error 2: Solution**의 과정을 거친다.

### Error 2: Solution

#### JAVA code

```
int a = 0
```

 문장의 마지막에 ; 이 없는 틀린 문법의 문장이다.

#### Symbol Table

```
['TYPE', 'int']
['IDENTIFIER', 'a']
['ASSIGN', '=']
['INTEGER', '0']
```

마지막에 `semi`가 없어서 `syntax error`가 나는 토큰 테이블이다.

#### SyntaxError Message

```
vtype id assign num $  
REJECT  
[REJECT] Error 2 : Token number (4) ['INTEGER', '0']  
SyntaxError: Missing a terminal corresponding to one of the following: (token line 4)  
['semi', 'addsub', 'multdiv', 'rparen']
```

[REJECT]이 뜨고, 틀린 토큰 심볼 테이블의 라인 부근에 올바른 Syntax가 되기 위해 추가해주어야 할 terminal의 종류를 알려준다(missing 된 아이템). 만약 이 에러 메시지를 보았다면 빠진 terminal을 넣어 올바른 syntax를 만들 수 있다.

일반적으로 error는 ACTION table에서 공백 칸을 만날 때 생기는 경우가 많았다. GOTO table에서 공백을 만난 경우는 symbol table이 잘못 옮겨졌거나, rules에 잘못된 문자가 있는 경우였다. 이것은 symbol table과 rules를 다시 검토하여 해결하였다.

## #5 TEST INPUT & OUTPUT

> 파일 이름

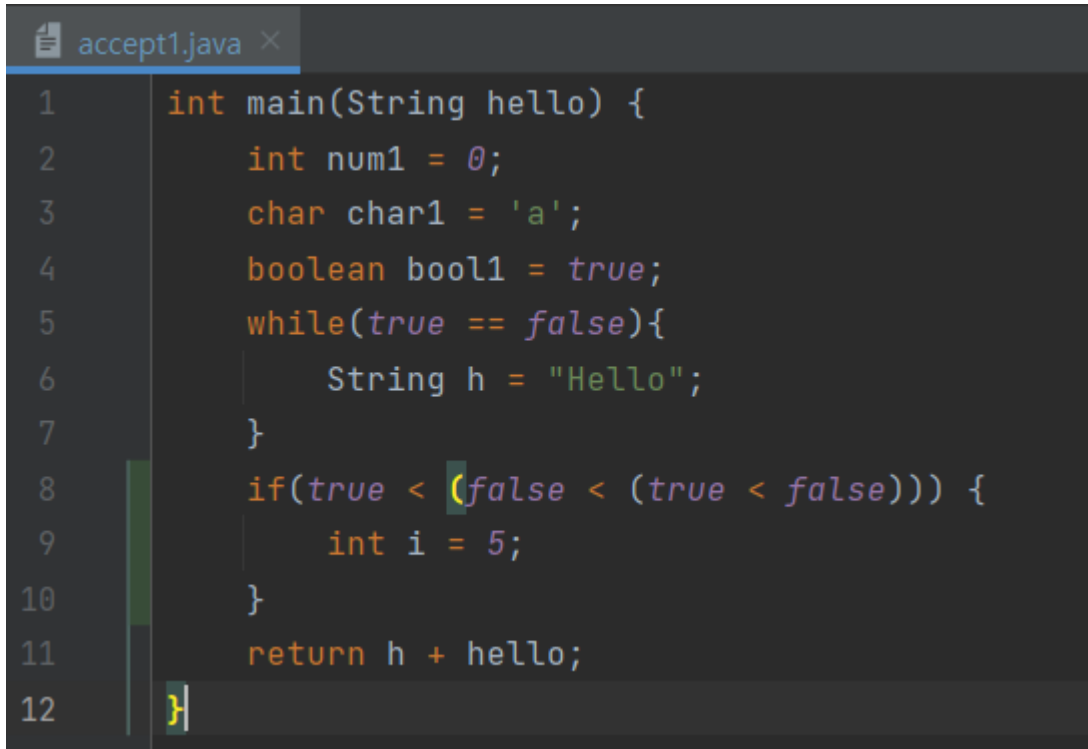
[file\_name].java 를 lexical\_analyzing 한 결과가 [file\_name].out 이고

[file\_name].out 을 syntax\_analyzing 한 결과가 [file\_name]\_final.out 이다.

### > Accept Case 1 : CODE -> FDECL 형태

파일: accept1.java => accept1.out => **accept1\_final.out**

#### 1. accept1.java



```
1  int main(String hello) {
2      int num1 = 0;
3      char char1 = 'a';
4      boolean bool1 = true;
5      while(true == false){
6          String h = "Hello";
7      }
8      if(true < (false < (true < false))) {
9          int i = 5;
10     }
11     return h + hello;
12 }
```

: if 문의 조건식으로 true < (false < (true < false)) 형태를 사용하여 비교 연산 우선순위를 괄호를 통해 지정하여 알맞은 문법으로 인식하였다. 이것은 우리가 기존 CFG를 모호하지 않은 문법으로 바꾸었기 때문이다.

#### 2. after lexical analyzing

```

accept1.java × accept1.out ×
1  ['TYPE', 'int']
2  ['IDENTIFIER', 'main']
3  ['LPAREN', '(']
4  ['TYPE', 'String']
5  ['IDENTIFIER', 'hello']
6  ['RPAREN', ')']
7  ['LBRACE', '{']
8  ['TYPE', 'int']
9  ['IDENTIFIER', 'num1']
10 ['ASSIGN', '=']
11 ['INTEGER', '0']
12 ['SEMICOLON', ';']
13 ['TYPE', 'char']
14 ['IDENTIFIER', 'char1']
15 ['ASSIGN', '=']
16 ['CHARACTER', "'a'"]
17 ['SEMICOLON', ';']
18 ['TYPE', 'boolean']
19 ['IDENTIFIER', 'bool1']
20 ['ASSIGN', '=']
21 ['BOOLEAN', 'true']
22 ['SEMICOLON', ';']
23 ['KEYWORD', 'while']
24 ['LPAREN', '(']
25 ['BOOLEAN', 'true']
26 ['RELOP', '==']
27 ['BOOLEAN', 'false']
28 ['RPAREN', ')']
29 ['LBRACE', '{']
30 ['TYPE', 'String']
31 ['IDENTIFIER', 'h']
32 ['ASSIGN', '=']
33 ['LITERAL', '"Hello"']
34 ['SEMICOLON', ';']
35 ['RBRACE', '}']
36 ['KEYWORD', 'if']
37 ['LPAREN', '(']
38 ['BOOLEAN', 'true']
39 ['RELOP', '<']
40 ['LPAREN', '(']
41 ['BOOLEAN', 'false']
42 ['RELOP', '<']
43 ['LPAREN', '(']
44 ['BOOLEAN', 'true']
45 ['RELOP', '<']
46 ['BOOLEAN', 'false']
47 ['RPAREN', ')']
48 ['RPAREN', ')']
49 ['RPAREN', ')']
50 ['LBRACE', '{']
51 ['TYPE', 'int']
52 ['IDENTIFIER', 'i']
53 ['ASSIGN', '=']
54 ['INTEGER', '5']
55 ['SEMICOLON', ';']
56 ['RBRACE', '}']
57 ['KEYWORD', 'return']
58 ['IDENTIFIER', 'h']
59 ['OPERATOR', '+']
60 ['IDENTIFIER', 'hello']
61 ['SEMICOLON', ';']
62 ['RBRACE', '}']

```

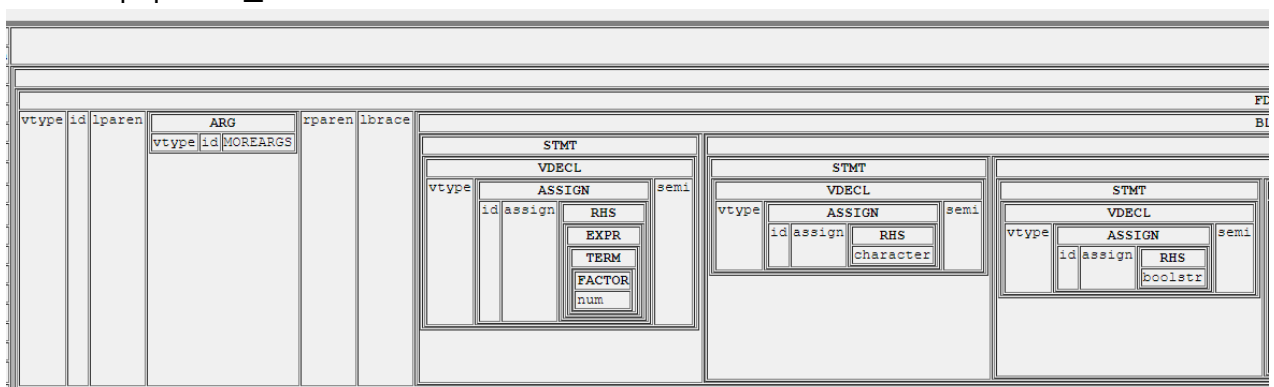
3. after syntax analyzing

```

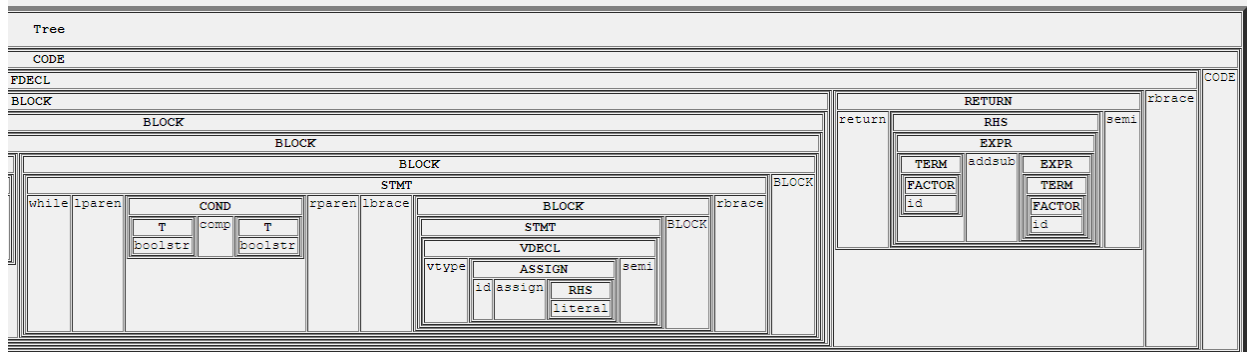
accept1.java × accept1.out × accept1_final.out ×
1  ACCEPT

```

4. 사이트로 검토







## > Accept Case 2 : CODE -> CDECL 형태

파일: accept2.java => accept2.out => **accept2\_final.out**

1. accept2.java

```
1 class Profile {
2     String name = "Compiler";
3 }
```

2. after lexical analyzing

```
1 ['KEYWORD', 'class']
2 ['IDENTIFIER', 'Profile']
3 ['LBRACE', '{']
4 ['TYPE', 'String']
5 ['IDENTIFIER', 'name']
6 ['ASSIGN', '=']
7 ['LITERAL', '"Compiler"']
8 ['SEMICOLON', ';']
9 ['RBRACE', '}']
```

3. after syntax analyzing

```
1 ACCEPT
```

4. 사이트로 검토

Input (tokens):

Maximum number of steps:

Step	Stack	Trace	Input	Action
1			class id lbrace vtype id assign literal semi rbrace \$	s6
2	0 class 6		id lbrace vtype id assign literal semi rbrace \$	s12
3	0 class 6 id 12		lbrace vtype id assign literal semi rbrace \$	s17
4	0 class 6 id 12 lbrace 17		vtype id assign literal semi rbrace \$	s5
5	0 class 6 id 12 lbrace 17 vtype 5		id assign literal semi rbrace \$	s10
6	0 class 6 id 12 lbrace 17 vtype 5 id 10		assign literal semi rbrace \$	s15
7	0 class 6 id 12 lbrace 17 vtype 5 id 10 assign 15		literal semi rbrace \$	s22
8	0 class 6 id 12 lbrace 17 vtype 5 id 10 assign 15 literal 22		semi rbrace \$	s9
9	0 class 6 id 12 lbrace 17 vtype 5 id 10 assign 15 RHS		semi rbrace \$	s20
10	0 class 6 id 12 lbrace 17 vtype 5 id 10 assign 15 RHS 20		semi rbrace \$	s7
11	0 class 6 id 12 lbrace 17 vtype 5 ASSIGN		semi rbrace \$	s11
12	0 class 6 id 12 lbrace 17 vtype 5 ASSIGN 11		semi rbrace \$	s16
13	0 class 6 id 12 lbrace 17 vtype 5 ASSIGN 11 semi 16		rbrace \$	s6
14	0 class 6 id 12 lbrace 17 VDECL		rbrace \$	s11
15	0 class 6 id 12 lbrace 17 VDECL 31		rbrace \$	s40
16	0 class 6 id 12 lbrace 17 VDECL 31 ODECL		rbrace \$	s39
17	0 class 6 id 12 lbrace 17 VDECL 31 ODECL 39		rbrace \$	s38
18	0 class 6 id 12 lbrace 17 ODECL		rbrace \$	s30
19	0 class 6 id 12 lbrace 17 ODECL 30		rbrace \$	s38
20	0 class 6 id 12 lbrace 17 ODECL 30 rbrace 38		\$	s37
21	0 CDECL		\$	s4
22	0 CDECL 4		\$	s4
23	0 CDECL 4 CODE		\$	s9
24	0 CDECL 4 CODE 9		\$	s3
25	0 CODE		\$	s1
26	0 CODE 1		\$	acc

**Tree**

파일: accept3.java => accept3.out => **accept3\_final.out**

## 1. accept3.java

2. after lexical analyzing

### 3. after syntax analyzing

#### 4. 사이트로 검토



## 1. error1.java

```
1  boolean Error() {
2      while(true < false < 10 < 3) {
3          int a = 3;
4      }
5      return "Error Report";
6  }
```

일반적으로 simple Java programming language 에서 (true < false) && (false < 10) && (10 < 3) 와 같이 비교 연산자에 대해 두 값에 대해서만 비교해 준다. 여러 개를 동시에 비교하고 싶다면 괄호를 써서 우선순위를 처리하기 때문에 우리는 앞서 CFG의 모호성을 수정하였으며 위 java파일 속 line 2의 while 조건문은 syntax error를 반환할 것이다. false 뒤에 '<' 온다는 에러가 반환되지만 실제로 우리가 구현한 syntax\_analyzer.py에서는 숫자에 대한 comp 기능은 제공하지 않으므로 숫자가 들어간 것도 에러의 요인 중 하나이다.

## 2. after lexical analyzing

```
1  ['TYPE', 'boolean']
2  ['IDENTIFIER', 'Error']
3  ['LPAREN', '(']
4  ['RPAREN', ')']
5  ['LBRACE', '{']
6  ['KEYWORD', 'while']
7  ['LPAREN', '(']
8  ['BOOLEAN', 'true']
9  ['RELOP', '<']
10 ['BOOLEAN', 'false']
11 ['RELOP', '<']
12 ['INTEGER', '10']
13 ['RELOP', '<']
14 ['INTEGER', '3']
15 ['RPAREN', ')']
16 ['LBRACE', '{']
17 ['TYPE', 'int']
18 ['IDENTIFIER', 'a']
19 ['ASSIGN', '=']
20 ['INTEGER', '3']
21 ['SEMICOLON', ';']
22 ['RBRACE', '}']
23 ['KEYWORD', 'return']
24 ['LITERAL', '"Error Report"']
25 ['SEMICOLON', ';']
26 ['RBRACE', '}']
```

## 3. after syntax analyzing

```

1 [REJECT] Error 2 : Token number (10) ['BOOLEAN', 'false']
2 SyntaxError: Missing a terminal corresponding to one of the following: (token line 10)
3 ['rparen']

```

error line 10은 error1.out파일 기준으로 ['BOOLEAN', 'false'] 이다. line 11에 해당하는 lexeme는 ['RELOP', '<'] 이다. false 다음에는 rparen이 와야하는 데 올 수 없는 lexeme으로 정의 된 comp( = RELOP ) 가 나왔으므로 Syntax error가 발생해 REJECT 되었다.

#### 4. 사이트로 검토

Input (tokens): `vtype id lparen rparen lbrace while lparen boolstr comp boolstr comp num comp`

`num rparen lbrace vtype id assign num semi rbrace return literal semi rbrace`

12	0	vtype	5	id	10	lparen	14	ARG	18	rparen	33	lbrace	41	while	52	lparen	61	T		comp	boolstr	comp	num	comp	num	rparen	lbrace	vtype	id	assign	num	semi	rbrace	return	literal	semi	rbrace	\$				
13	0	vtype	5	id	10	lparen	14	ARG	18	rparen	33	lbrace	41	while	52	lparen	61	T	67		comp	boolstr	comp	num	comp	num	rparen	lbrace	vtype	id	assign	num	semi	rbrace	return	literal	semi	rbrace	\$			
14	0	vtype	5	id	10	lparen	14	ARG	18	rparen	33	lbrace	41	while	52	lparen	61	T	67	comp	74		boolstr	comp	num	comp	num	rparen	lbrace	vtype	id	assign	num	semi	rbrace	return	literal	semi	rbrace	\$		
15	0	vtype	5	id	10	lparen	14	ARG	18	rparen	33	lbrace	41	while	52	lparen	61	T	67	comp	74	boolstr	69		comp	num	comp	num	rparen	lbrace	vtype	id	assign	num	semi	rbrace	return	literal	semi	rbrace	\$	
16	0	vtype	5	id	10	lparen	14	ARG	18	rparen	33	lbrace	41	while	52	lparen	61	T	67	comp	74	T			comp	num	comp	num	rparen	lbrace	vtype	id	assign	num	semi	rbrace	return	literal	semi	rbrace	\$	
17	0	vtype	5	id	10	lparen	14	ARG	18	rparen	33	lbrace	41	while	52	lparen	61	T	67	comp	74	T	78			comp	num	comp	num	rparen	lbrace	vtype	id	assign	num	semi	rbrace	return	literal	semi	rbrace	\$

### > Error Case 2

파일: error2.java => error2.out => **error2\_final.out**

#### 1. error2.java

```

1 String a;
2 String newStr = "Error";
3 a = a + newStr;

```

우리가 구현한 syntax\_analyzer.py에서 사용된 CFG에 따르면 변수 타입 선언 없이 Identifier가 문장의 젤 앞에 나올 수 없다. 그러므로 line 3의 a에 변수 타입이 없어 Syntax Error를 반환할 것이다.

#### 2. after lexical analyzing

```

1 ['TYPE', 'String']
2 ['IDENTIFIER', 'a']
3 ['SEMICOLON', ';']
4 ['TYPE', 'String']
5 ['IDENTIFIER', 'newStr']
6 ['ASSIGN', '=']
7 ['LITERAL', '"Error"']
8 ['SEMICOLON', ';']
9 ['IDENTIFIER', 'a']
10 ['ASSIGN', '=']
11 ['IDENTIFIER', 'a']
12 ['OPERATOR', '+']
13 ['IDENTIFIER', 'newStr']
14 ['SEMICOLON', ';']

```

#### 3. after syntax analyzing

```

1 [REJECT] Error 2 : Token number (8) ['SEMICOLON', ';']
2 SyntaxError: Missing a terminal corresponding to one of the following: (token line 8)
3 ['vtype', 'class']

```

error line 8은 error2.out파일 기준으로 ['SEMICOLON', ';'] 이다. line 9 즉, 다음에 오는 lexeme는 ['IDENTIFIER', 'a'] 이다. 세미콜론 이후에 새로운 문장을 시작할 때는 vtype이나 class가 와야하는 데 올 수 없는 lexeme으로 정의 된 id( = IDENTIFIER )가 나왔으므로 Syntax error가 발생해 REJECT 되었다.

#### 4. 사이트로 검토

Input (tokens): vtype id semi vtype id assign literal semi id assign id addsub id semi

13	0	VDECL	2	vtype	5	ASSIGN		semi id assign id addsub id semi \$	11
14	0	VDECL	2	vtype	5	ASSIGN	11	semi id assign id addsub id semi \$	s16
15	0	VDECL	2	vtype	5	ASSIGN	11 semi 16	id assign id addsub id semi \$	r <sub>6</sub>
16	0	VDECL	2	VDECL				id assign id addsub id semi \$	2
17	0	VDECL	2	VDECL	2			id assign id addsub id semi \$	

## #6 GITHUB & Google Docs & KakaoTalk & Zoom

팀플에 있어서 협업 방식으로 GITHUB와 Google Docs, KakaoTalk, Zoom 등을 이용하였다.

- ~ 20210520 : Compiler-term-project 계획 및 모호한 개념 학습
- ~ 20210528 : ambiguous -> non-ambiguous로 수정
- ~ 20210530 : 구현 원리(전체적인 flow) 논의
- ~ 20210602 : rules.py, SLR.py, syntax\_analyzer.py 구현
- ~ 20210603 : 추가 에러 수정 및 보고서 작성 및 코드 주석 추가

> 깃허브 링크 : [https://github.com/InKyeongBae/Compiler\\_syntax\\_analyzer.py](https://github.com/InKyeongBae/Compiler_syntax_analyzer.py)는 **final** 폴더에서 작업을 함.

The screenshot shows a GitHub repository page for 'Compiler\_syntax\_analyzer.py' by InKyeongBae. The repository has 3 branches and 0 tags. The main branch is 'master'. The repository contains a table of files and folders with their commit history and timestamps. The files are: .idea, compiler, final, and test. The 'final' folder is the most recent, updated 6 hours ago. The repository also has a README file and a 'Add a README' button. The right sidebar shows the repository's 'About' section, 'Releases' (no releases published), 'Packages' (no packages published), 'Contributors' (2 contributors: InKyeongBae and Jwaminju), and 'Languages' (Python 97.5%, Java 2.5%).

File/Folder	Commit Message	Timestamp
.idea	[UPDATE] nextState	2 months ago
compiler	[CREATE] final directory&files	22 hours ago
final	[UPDATE] alt tab 정리	6 hours ago
test	[STYLE] syntax->test directory 생성	15 days ago

Help people interested in this repository understand your project by adding a README. [Add a README](#)

**About**  
2021 Compiler Team Project

**Releases**  
No releases published  
[Create a new release](#)

**Packages**  
No packages published  
[Publish your first package](#)

**Contributors** (2)  
InKyeongBae INKYEONG BAE  
Jwaminju wony0120

**Languages**  
Python 97.5% Java 2.5%

## #7 LINUX

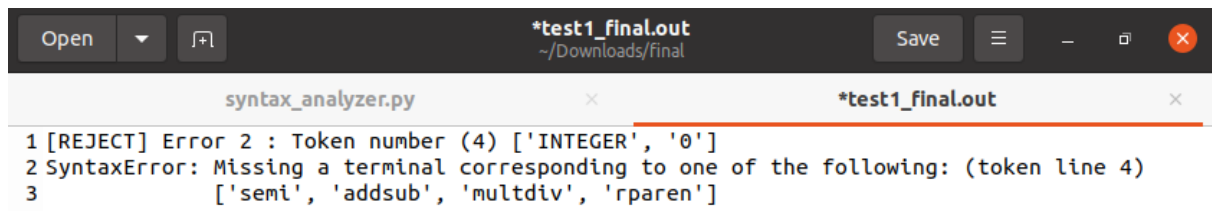
lexical\_analyzer.py와 syntax\_analyzer.py를 연달아 실행시키면, JAVA언어를 terminal sequence로 변환한 값과, Acceptance 여부를 REJECT, ACCEPT가 출력된다. 파일에도 결과가 출력된다.

```
$ python3 lexical_analyzer.py input.java
$ python3 syntax_analyzer.py input.out
```

( 이 때, lexical\_analyzer.py, DFA.py, syntax\_analyzer.py, rules.py, SLR.py 파일은 모두 같은 수준의 위치에 있어야 한다.)

### 실행 결과 1) REJECT

```
yong@ubuntu:~/Downloads/final$ python3 lexical_analyzer.py test1.java
yong@ubuntu:~/Downloads/final$ python3 syntax_analyzer.py test1.out
vtype id assign num $
REJECT
[REJECT] Error 2 : Token number (4) ['INTEGER', '0']
SyntaxError: Missing a terminal corresponding to one of the following: (token line 4)
['semi', 'addsub', 'multdiv', 'rparen']
```



### 실행 결과 2) ACCEPT

```
yong@ubuntu:~/Downloads/final$ python3 lexical_analyzer.py accept1.java
yong@ubuntu:~/Downloads/final$ python3 syntax_analyzer.py accept1.out
vtype id lparen vtype id rparen lbrace vtype id assign num semi vtype id assign character semi vtype
id assign boolstr semi while lparen boolstr comp boolstr rparen lbrace vtype id assign literal semi rb
race return id addsub id semi rbrace $
ACCEPT
```

