

Syntax analyzer

20174158 김두성

20170768 이영석

Syntax analyzer를 만들기 위해서는 CFG G를 non-ambiguous 하게 표현해야 합니다. 주어진 G는 아래와 같습니다.

- 01: $CODE \rightarrow VDECL\ CODE \mid FDECL\ CODE \mid CDECL\ CODE \mid \epsilon$
- 02: $VDECL \rightarrow vtype\ id\ semi \mid vtype\ ASSIGN\ semi$
- 03: $ASSIGN \rightarrow id\ assign\ RHS$
- 04: $RHS \rightarrow EXPR \mid literal \mid character \mid boolstr$
- 05: $EXPR \rightarrow EXPR\ addsub\ EXPR \mid EXPR\ multdiv\ EXPR$
- 06: $EXPR \rightarrow lparen\ EXPR\ rparen \mid id \mid num$
- 07: $FDECL \rightarrow vtype\ id\ lparen\ ARG\ rparen\ lbrace\ BLOCK\ RETURN\ rbrace$
- 08: $ARG \rightarrow vtype\ id\ MOREARGS \mid \epsilon$
- 09: $MOREARGS \rightarrow comma\ vtype\ id\ MOREARGS \mid \epsilon$
- 10: $BLOCK \rightarrow STMT\ BLOCK \mid \epsilon$
- 11: $STMT \rightarrow VDECL \mid ASSIGN\ semi$
- 12: $STMT \rightarrow if\ lparen\ COND\ rparen\ lbrace\ BLOCK\ rbrace\ ELSE$
- 13: $STMT \rightarrow while\ lparen\ COND\ rparen\ lbrace\ BLOCK\ rbrace$
- 14: $COND \rightarrow COND\ comp\ COND \mid boolstr$
- 15: $ELSE \rightarrow else\ lbrace\ BLOCK\ rbrace \mid \epsilon$
- 16: $RETURN \rightarrow return\ RHS\ semi$
- 17: $CDECL \rightarrow class\ id\ lbrace\ ODECL\ rbrace$
- 18: $ODECL \rightarrow VDECL\ ODECL \mid FDECL\ ODECL \mid \epsilon$

다른 부분에서는 ambiguous이 없는 것으로 보였으나 5번과 14번에서 ambiguous를 발견하였습니다. 14번에서는 left recursive도 발견되었습니다. 이것을 해결하기 위해서 식을 다시 정의하도록 하였습니다.

5번 식은

$EXPR \rightarrow T\ addsub\ EXPR \mid T$
 $T \rightarrow F\ multdiv\ T \mid F$
 $F \rightarrow lparen\ EXPR\ rparen \mid id \mid num$

와 같이 바꿈으로 non-ambiguous하게 만들 수 있었습니다.

14번 식은

$COND \rightarrow A\ comp\ COND \mid A$
 $A \rightarrow boolstr$

와 같이 바꿈으로 ambiguous도 없애고 left recursive도 사라졌습니다.

그리고 위의 내용을 바탕으로 SLR Parser Generator에 아래와 같은 값을 넣어서 결과를 살펴보았습니다

다. 그 결과 정상적인 SLR-parsing table을 출력할 수 있었습니다.

```
(0) S -> CODE
(1) CODE -> VDECL CODE
(2) CODE -> FDECL CODE
(3) CODE -> CDECL CODE
(4) CODE -> ''
(5) VDECL -> vtype id semi
(6) VDECL -> vtype ASSIGN semi
(7) ASSIGN -> id assign RHS
(8) RHS -> EXPR
(9) RHS -> literal
(10) RHS -> character
(11) RHS -> boolstr
(12) EXPR -> T addsub EXPR
(13) EXPR -> T
(14) T -> F multdiv T
(15) T -> F
(16) F -> lparen EXPR rparen
(17) F -> id
(18) F -> num
(19) FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
(20) ARG -> vtype id MOREARGS
(21) ARG -> ''
(22) MOREARGS -> comma vtype id MOREARGS
(23) MOREARGS -> ''
(24) BLOCK -> STMT BLOCK
(25) BLOCK -> ''
(26) STMT -> VDECL
(27) STMT -> ASSIGN semi
(28) STMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE
(29) STMT -> while lparen COND rparen lbrace BLOCK rbrace
(30) COND -> A comp COND
(31) COND -> A
(32) A -> boolst
(33) ELSE -> else lbrace BLOCK rbrace
(34) ELSE -> ''
(35) RETURN -> return RHS semi
(36) CDECL -> class id lbrace ODECL rbrace
(37) ODECL -> VDECL ODECL
(38) ODECL -> FDECL ODECL
(39) ODECL -> ''
```

아래는 SLR parsing table의 Action 부분입니다.

[illegible]

State	ACTION																							
	vtype	id	semi	assign	literal	character	boolstr	addsub	multdiv	lparen	rparen	num	lbrace	rbrace	comma	if	while	comp	boolst	else	return	class	\$	
67											r31							s73						
68											r32							r32						
69											s74													
70											r22													
71														r35										
72												s75												
73																			s68					
74												s77												
75	s53	s54												r25		s51	s52				r25			
76											r30													
77	s53	s54												r25		s51	s52				r25			
78														s80										
79														s81										
80	r34	r34												r34		r34	r34			s83	r34			
81	r29	r29												r29		r29	r29				r29			
82	r28	r28												r28		r28	r28				r28			
83													s84											
84	s53	s54												r25		s51	s52				r25			
85														s86										
86	r33	r33												r33		r33	r33				r33			

아래는 SLR parsing table의 Goto 부분입니다.

[illegible]

[illegible]

이제 저희의 코드를 설명드리겠습니다. 저희 코드는 python으로 작성되었습니다. 이전 lexical Analyzer의 결과값을 받아와서 LAcod에 저장하였습니다.

```
test - Windows 메모장
파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)
<CLASS><IDENTIFIER, myclass><LBRACE><INT><IDENTIFIER, i><ASSIGNMENT><INTEGER,
5><SEMI><RBRACE><BOOL><IDENTIFIER, abc><ASSIGNMENT><BOOLEAN,
true><SEMI><INT><IDENTIFIER, i><ASSIGNMENT><IDENTIFIER, a><MULTIFLY><LPAREN><INTEGER,
4><PLUS><IDENTIFIER, i><RPAREN><SEMI>
```

이전 결과값은 위와 같은 형태이므로 더 쉽게 나타내기 위해서 형식을 조금 바꾸었습니다. 문자열의 형태로 되어 있는 상태에서 List 구조로 바꾸었고 꺾쇠 괄호를 없앴습니다.

```
1 import sys
2
3 A = sys.argv[1]
4 file = open(A,"rt")
5 LAcod = file.read() # lexical_analyzer의 결과를 input으로 받아 LAcod에 저장합니다
6 file.close
7 if len(LAcod) > 1: # LAcod1 에 하나라도 들어있다면 양 끝의 '<', '>'를 없애줍니다
8     LAcod = LAcod[1:-1]
9 SAcod = LAcod.split('><') # '><'를 기준으로 토큰별로 나눠서 SAcod에 저장합니다
10 SAcod.append('$') # 마지막에 '$' 토큰을 추가합니다
11 SAcod_save = SAcod[:] # 에러가 발생했을 때 위치를 알려주기 위해 따로 저장을 해둡니다
12
```

그리고 그에 맞추어서 SAcod에서 사용할 terminals로 변환하기 위해 dictionary 자료형으로 만들어서 SAcod 리스트에 입력하였습니다.

```
13 # SAcod에서 주어진 Terminals로 변환하기 위한 dictionary 자료형입니다
14 SAcodtoTerminals = {'INT':'vtype',
15 'CHAR':'vtype',
16 'STR':'vtype',
17 'BOOL':'vtype',
18 'INTEGER':'num',
19 'CHARACTER':'character',
20 'BOOLEAN':'boolstr',
21 'STRING':'literal',
22 'IDENTIFIER':'id',
23 'COMPARE':'comp',
24 'PLUS':'addsub',
25 'MINUS':'addsub',
26 'MULTIFLY':'multdiv',
27 'DIVISION':'multdiv',
28 'ASSIGNMENT':'assign',
29 'SEMI':'semi',
30 'LBRACE':'lbrace',
31 'RBRACE':'rbrace',
32 'LPAREN':'lparen',
33 'RPAREN':'rparen',
34 # 'LBRACKET':'',
35 # 'RBRACKET':'',
36 'COMMA':'comma',
37 'IF':'if',
38 'ELSE':'else',
39 'WHILE':'while',
40 'CLASS':'class',
41 'RETURN':'return'
42
```

```

43 # '$'가 나오기 전까지 SAcodeterminals로 변환합니다
44 for i in range(len(SAcodeterminals)-1):
45     SAcodeterminals[i] = SAcodeterminals[i].split(',')[0]
46     if SAcodeterminals[i] in SAcodeterminals:
47         SAcodeterminals[i] = SAcodeterminals[SAcodeterminals[i]]
48     else:
49         print('err : ' + SAcodeterminals[i]) # Terminals로 변환하지 못하는 경우에 에러를 출력해줍니다
50

```

그리고 SLR parsing table의 내용을 저장하기 위해서 dictionary 자료형을 사용하였습니다. 형식은 (state, terminal or nonterminal): 'action' 과 같은 형태입니다.

```

51 # SLR parsing table에 대한 dictionary 자료형입니다
52 LRtable = {(0, 'vtype'): 's5',
53 (2, 'vtype'): 's5',
54 (3, 'vtype'): 's5',
55 (4, 'vtype'): 's5',
56 (13, 'vtype'): 'r5',
57 (14, 'vtype'): 's19',
58 (16, 'vtype'): 'r6',
59 (17, 'vtype'): 's5',
60 (31, 'vtype'): 's5',
61 (32, 'vtype'): 's5',
62 (38, 'vtype'): 'r36',
63 (41, 'vtype'): 's53',
64 (43, 'vtype'): 's55',
65 (48, 'vtype'): 's53',
66 (49, 'vtype'): 'r26',
67 (59, 'vtype'): 'r27',
68 (64, 'vtype'): 'r19',
69 (75, 'vtype'): 's53',
70 (77, 'vtype'): 's53',
71 (80, 'vtype'): 'r34',
72 (81, 'vtype'): 'r29',
73 (82, 'vtype'): 'r28',
74 (84, 'vtype'): 's53',
75 (86, 'vtype'): 'r33',
76
77 (5, 'id'): 's10',
78 (6, 'id'): 's12',
79 (13, 'id'): 'r5',
80 (15, 'id'): 's28',
81 (16, 'id'): 'r6',
82 (17, 'id'): 's5',
83 (31, 'id'): 's5',
84 (32, 'id'): 's5',
85 (38, 'id'): 'r36',
86 (41, 'id'): 's53',
87 (43, 'id'): 's55',
88 (48, 'id'): 's53',
89 (49, 'id'): 'r26',
90 (59, 'id'): 'r27',
91 (64, 'id'): 'r19',
92 (75, 'id'): 's53',
93 (77, 'id'): 's53',
94 (80, 'id'): 'r34',
95 (81, 'id'): 'r29',
96 (82, 'id'): 'r28',
97 (84, 'id'): 's53',
98 (86, 'id'): 'r33',
99

```

CFG G도 list 형태로 저장하기 위해 우선 문자열로 저장한 후에 list로 변환하였습니다. 아래와 같이 SLR_grammer 라는 2차원 배열로 저장하였습니다.

```
383 # ambiguity를 없앤 CFG입니다 (''는  $\epsilon$  입니다)
384 SLR_grammer = ""S -> CODE
385 CODE -> VDECL CODE
386 CODE -> FDECL CODE
387 CODE -> CDECL CODE
388 CODE -> ''
389 VDECL -> vtype id semi
390 VDECL -> vtype ASSIGN semi
391 ASSIGN -> id assign RHS
392 RHS -> EXPR
393 RHS -> literal
394 RHS -> character
395 RHS -> boolstr
396 EXPR -> T addsub EXPR
397 EXPR -> T
398 T -> F multdiv T
399 T -> F
400 F -> lparen EXPR rparen
401 F -> id
402 F -> num
403 FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
404 ARG -> vtype id MOREARGS
405 ARG -> ''
```

```
424
425 # CFG를 나눠서 저장합니다
426 SLR_grammer = SLR_grammer.split('\n')
427 for i in range(len(SLR_grammer)):
428     SLR_grammer[i] = SLR_grammer[i].split('->')
429     SLR_grammer[i][1] = SLR_grammer[i][1].split()
430
```

```

431 # syntax analyze
432 step, state, i = 1, 0, 0 # step: parsing(action) 단계, state: 현재 상태
433 stack = []
434 action = '0'
435 while True:
436     # action을 수행합니다
437     # action 이 reduce 일 때
438     if action[0] == 'r':
439         reduce = SLR_grammar[int(action[1:]))[1][:] # 여기서 reduce는 action의 CFG에서 -> 뒤에 있는 부분입니다
440         if reduce != ['']: # reduce가 ε 이 아닐 때 해당하는 stack을 지워줍니다
441             if stack[-1] == reduce[-1]:
442                 stack = stack[:-len(reduce)*2+1]
443             else:
444                 stack = stack[:-len(reduce)*2]
445         stack.append(SLR_grammar[int(action[1:]))[0]) # CFG에서 -> 앞에 있는 부분을 stack에 추가해줍니다
446     # action 이 shift and goto 일 때
447     elif action[0] == 's':
448         stack.append(SAcode[i]) # SAcode의 첫번째 데이터를 stack으로 옮겨줍니다
449         i = i + 1
450         state = int(action[1:]) # state를 갱신시키고 stack에 넣어줍니다
451         stack.append(state)
452     # action 이 accept 일 때
453     elif action == 'acc':
454         print('accept') # accept를 출력해주고 프로그램이 끝납니다
455         break
456     # action 이 goto 일 때
457     else:
458         state = int(action) # state를 갱신시키고 stack에 넣어줍니다
459         stack.append(state)
460

```

다음은 실행 부분입니다. state에 따른 action을 수행합니다. action의 경우에는 4가지가 있는데 reduce, shift and goto, acc, goto입니다. reduce의 경우에는 CFG와 stack 안에 있는 내용이 같으면 상위 node로 전환됩니다. 이 내용은 SLR parsing table 안에 있기 때문에 확인해보지 않고 진행합니다.

shift and goto의 경우에는 SAcode의 첫번째 데이터를 stack에 push합니다. 그 후에 현재 state를 저장합니다.

acc의 경우에는 문법적으로 올바르다고 판단한 경우이므로 더 이상 진행 할 필요가 없습니다. 따라서 반복문을 깨고 바깥으로 나갑니다.

goto의 경우에는 현재 state를 저장하고 stack에 넣습니다.

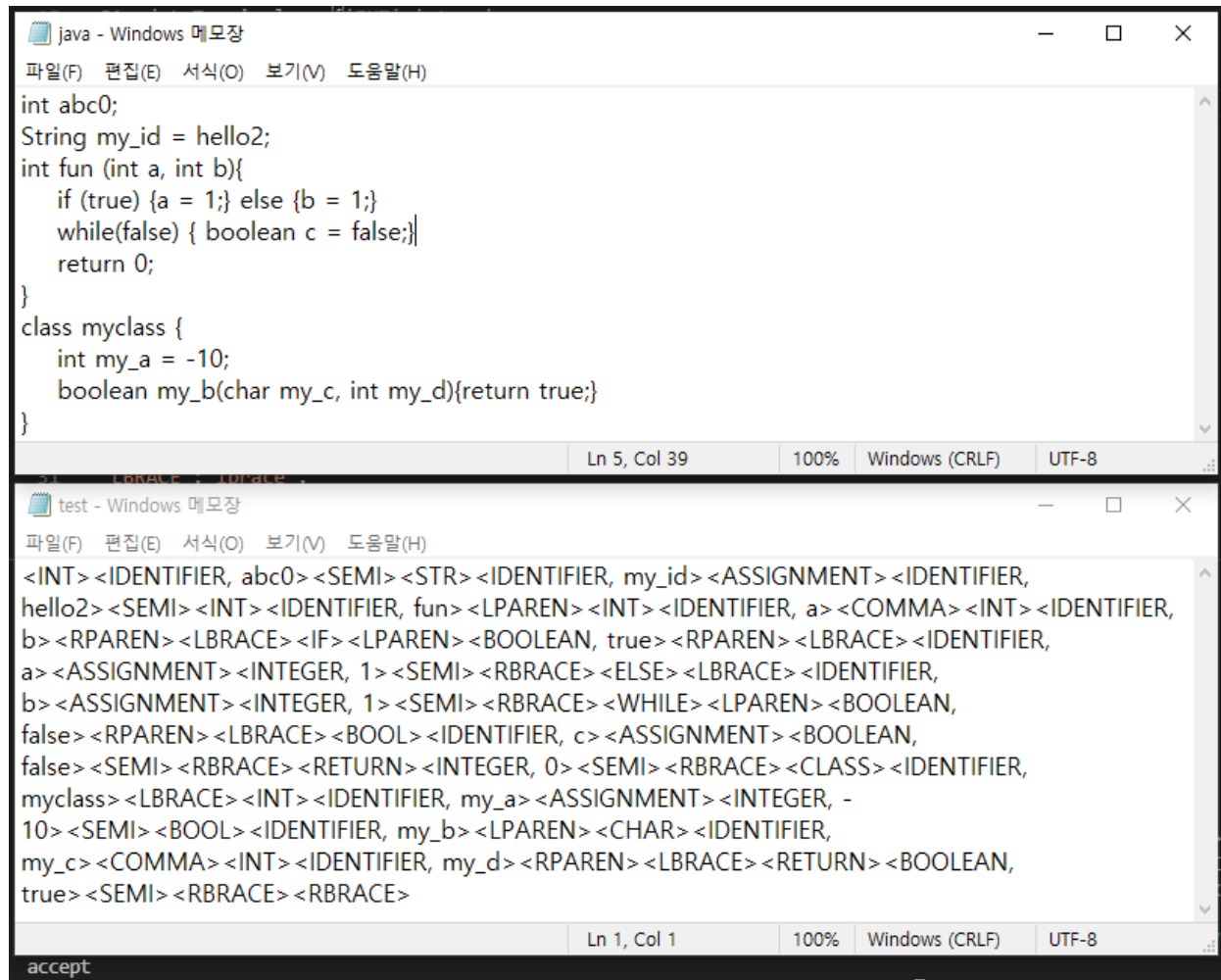
그리고 다음 action을 결정해야합니다. stack의 제일 위에 state가 있을 때와 없을 때로 나누어 SLR parsing table에 따라 다음 action을 정합니다. 만약 table에 없다면 문법적으로 문제가 있는 것이므로 에러의 위치와 이유를 알려주게 됩니다. 에러의 이유는 SLR parsing table에 따라서 다음에 올 token 으로 기대되는 것이 오지 않았기 때문이라고 서술하게 됩니다.

```

476     else: # stack의 마지막이 state가 아닐 때
477         # LRtable에 있으면 action을 갱신합니다
478         if (stack[-2], stack[-1].strip()) in LRtable:
479             action = LRtable[(stack[-2], stack[-1].strip())]
480         # LRtable에 없으면 reject를 출력해주고 에러의 위치와 이유를 알려줍니다
481         else:
482             print('reject')
483             print('에러가 {} 번째 토큰인 <{}> 에서 발생하였습니다'.format(i+1, SAcodesave[i]))
484             expect = [] # 에러가 나온 토큰 자리에 기대되는 토큰 배열입니다
485             for key_ in LRtable.keys():
486                 if key_[0]==stack[-2]:
487                     expect.append(key_[1])
488             print('<{}> 이 아닌 {} 가 올 것으로 기대됩니다'.format(SAcodesave[i], expect))
489             break
490
491     step = step + 1
492     # print('{} {} {} {}'.format(step, stack, SAcodesave[i:], action)) # parsing 과정을 보여줍니다
493

```

아래가 실행 결과입니다. 제일 위의 코드가 자바 코드이고 그 다음이 lexical analyzer를 통과한 코드입니다. 그 아래에 accept 라는 올바른 코드의 실행 결과를 볼 수 있습니다.

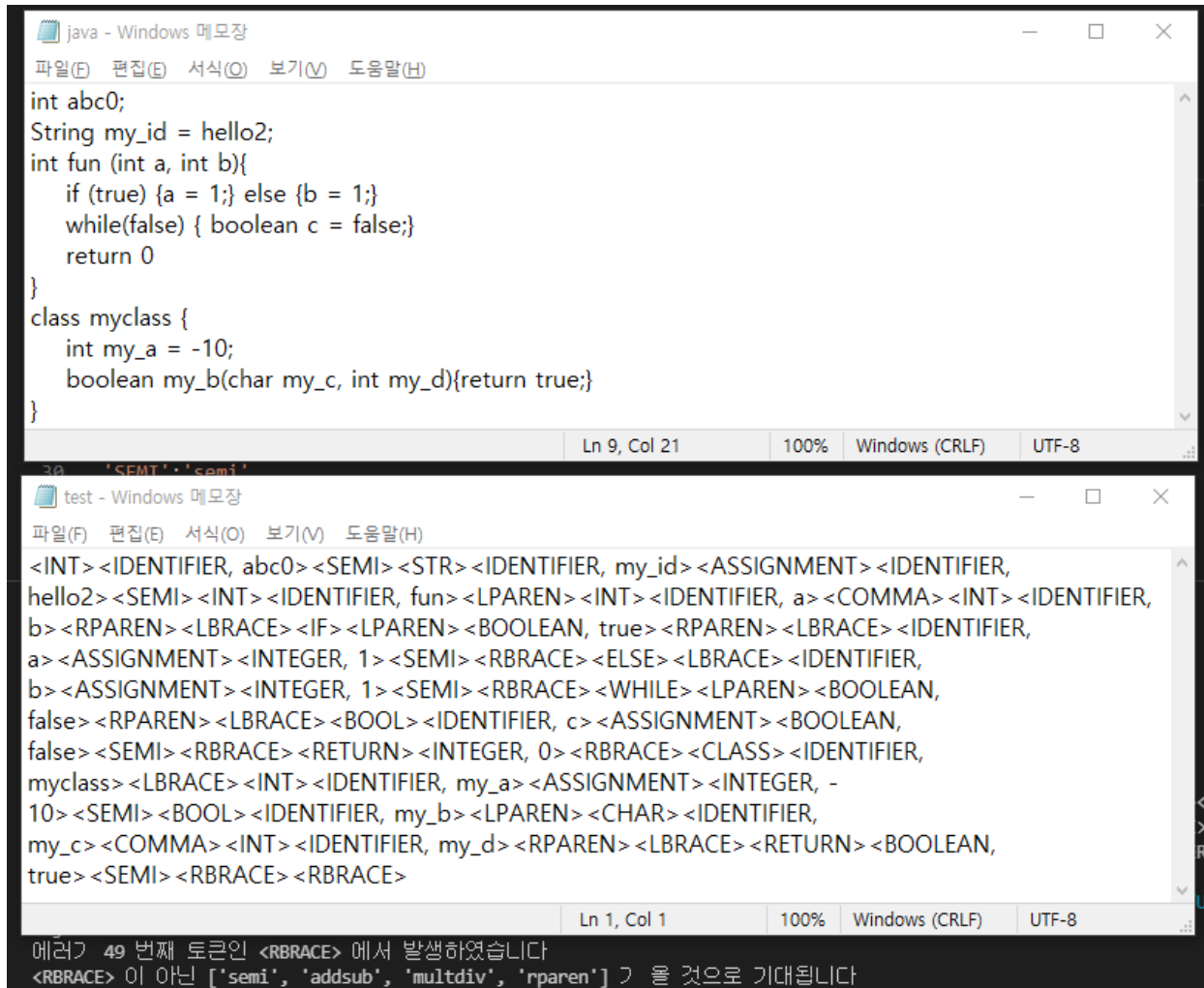


The image shows two screenshots of a Windows Notepad window. The top window, titled 'java - Windows 메모장', contains the following Java code:

```
int abc0;
String my_id = hello2;
int fun (int a, int b){
    if (true) {a = 1;} else {b = 1;}
    while(false) { boolean c = false;}
    return 0;
}
class myclass {
    int my_a = -10;
    boolean my_b(char my_c, int my_d){return true;}
}
```

The bottom window, titled 'test - Windows 메모장', shows the lexical analysis output of the code above. The output is a single line of tokens separated by spaces, enclosed in angle brackets. The tokens are: <INT> <IDENTIFIER, abc0> <SEMI> <STR> <IDENTIFIER, my_id> <ASSIGNMENT> <IDENTIFIER, hello2> <SEMI> <INT> <IDENTIFIER, fun> <LPAREN> <INT> <IDENTIFIER, a> <COMMA> <INT> <IDENTIFIER, b> <RPAREN> <LBRACE> <IF> <LPAREN> <BOOLEAN, true> <RPAREN> <LBRACE> <IDENTIFIER, a> <ASSIGNMENT> <INTEGER, 1> <SEMI> <RBRACE> <ELSE> <LBRACE> <IDENTIFIER, b> <ASSIGNMENT> <INTEGER, 1> <SEMI> <RBRACE> <WHILE> <LPAREN> <BOOLEAN, false> <RPAREN> <LBRACE> <BOOL> <IDENTIFIER, c> <ASSIGNMENT> <BOOLEAN, false> <SEMI> <RBRACE> <RETURN> <INTEGER, 0> <SEMI> <RBRACE> <CLASS> <IDENTIFIER, myclass> <LBRACE> <INT> <IDENTIFIER, my_a> <ASSIGNMENT> <INTEGER, -10> <SEMI> <BOOL> <IDENTIFIER, my_b> <LPAREN> <CHAR> <IDENTIFIER, my_c> <COMMA> <INT> <IDENTIFIER, my_d> <RPAREN> <LBRACE> <RETURN> <BOOLEAN, true> <SEMI> <RBRACE> <RBRACE>. The status bar at the bottom of the window shows 'Ln 1, Col 1', '100%', 'Windows (CRLF)', and 'UTF-8'. The word 'accept' is visible at the very bottom of the image.

반대로 문법적으로 올바르지 못한 코드의 실행 결과는 아래와 같습니다. return 0 이후의 세미콜론을 빼 보았습니다. 어떤 토큰이 틀렸는지 알려주고 있고 틀린 이유로 올 수 있는 다른 토큰들을 제시하고 있습니다.



The image shows two screenshots of a Java IDE window. The top screenshot displays a Java code snippet with a syntax error. The code is as follows:

```
int abc0;
String my_id = hello2;
int fun (int a, int b){
    if (true) {a = 1;} else {b = 1;}
    while(false) { boolean c = false;}
    return 0
}
class myclass {
    int my_a = -10;
    boolean my_b(char my_c, int my_d){return true;}
}
```

The status bar at the bottom of the window indicates "Ln 9, Col 21", "100%", "Windows (CRLF)", and "UTF-8".

The bottom screenshot shows the same code snippet tokenized. The tokens are displayed as follows:

```
<INT> <IDENTIFIER, abc0> <SEMI> <STR> <IDENTIFIER, my_id> <ASSIGNMENT> <IDENTIFIER,
hello2> <SEMI> <INT> <IDENTIFIER, fun> <LPAREN> <INT> <IDENTIFIER, a> <COMMA> <INT> <IDENTIFIER,
b> <RPAREN> <LBRACE> <IF> <LPAREN> <BOOLEAN, true> <RPAREN> <LBRACE> <IDENTIFIER,
a> <ASSIGNMENT> <INTEGER, 1> <SEMI> <RBRACE> <ELSE> <LBRACE> <IDENTIFIER,
b> <ASSIGNMENT> <INTEGER, 1> <SEMI> <RBRACE> <WHILE> <LPAREN> <BOOLEAN,
false> <RPAREN> <LBRACE> <BOOL> <IDENTIFIER, c> <ASSIGNMENT> <BOOLEAN,
false> <SEMI> <RBRACE> <RETURN> <INTEGER, 0> <RBRACE> <CLASS> <IDENTIFIER,
myclass> <LBRACE> <INT> <IDENTIFIER, my_a> <ASSIGNMENT> <INTEGER, -
10> <SEMI> <BOOL> <IDENTIFIER, my_b> <LPAREN> <CHAR> <IDENTIFIER,
my_c> <COMMA> <INT> <IDENTIFIER, my_d> <RPAREN> <LBRACE> <RETURN> <BOOLEAN,
true> <SEMI> <RBRACE> <RBRACE>
```

The status bar at the bottom of the window indicates "Ln 1, Col 1", "100%", "Windows (CRLF)", and "UTF-8".

Below the tokenized code, there is a message in Korean: "예러가 49 번째 토큰인 <RBRACE> 에서 발생하였습니다" (An error occurred at the 49th token, which is <RBRACE>). Below this message, there is a list of tokens that are not expected at this position: "<RBRACE> 이 아닌 ['semi', 'addsub', 'multdiv', 'rparen']" (Not <RBRACE>, but ['semi', 'addsub', 'multdiv', 'rparen']).

CFG G에 자바의 전체 문법을 모두 담아내기에는 주어진 G의 양이 적고 만약 그만큼 주어진다고 하면 SLR parsing table도 복잡해지고 시간도 그만큼 더 오래 걸릴 것이기 때문에 주어진 문법에 맞추어서 input 코드를 작성하는 것이 매우 한정적이었습니다. 다음에 기회가 된다면 좀 더 많은 문법을 작성하여 추가로 구문을 분석할 수 있도록 파서를 만들어 보도록 하겠습니다.

감사합니다.