project 2 : SLR parser

20172675 이해인
20171666 고재원

< contents >

1. Find & Correct ambiguous CFG

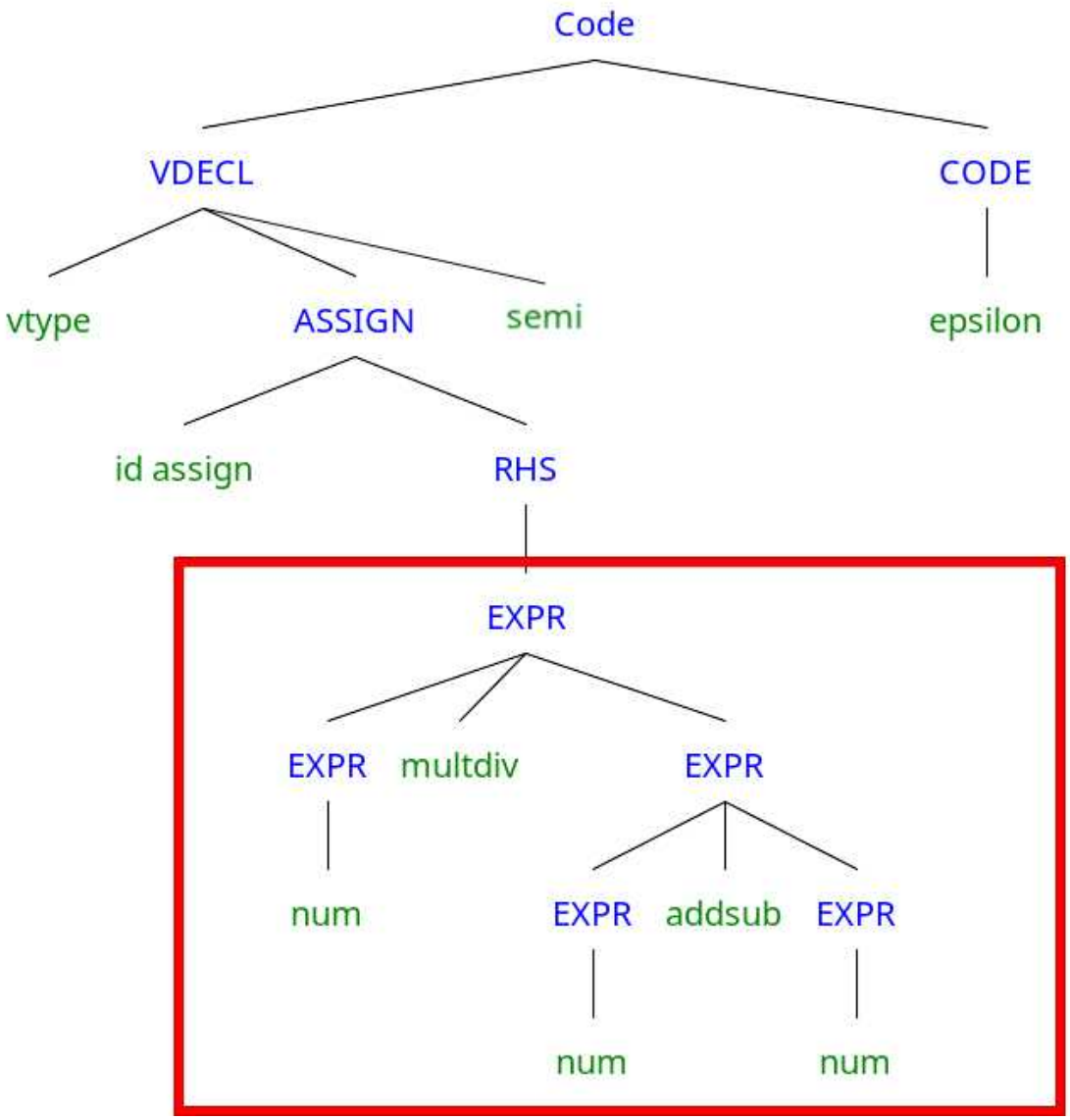
In the given CFG, there are two ambiguous syntaxes.

1. 05: EXPR→EXPR addsub EXPR | EXPR multdiv EXPR

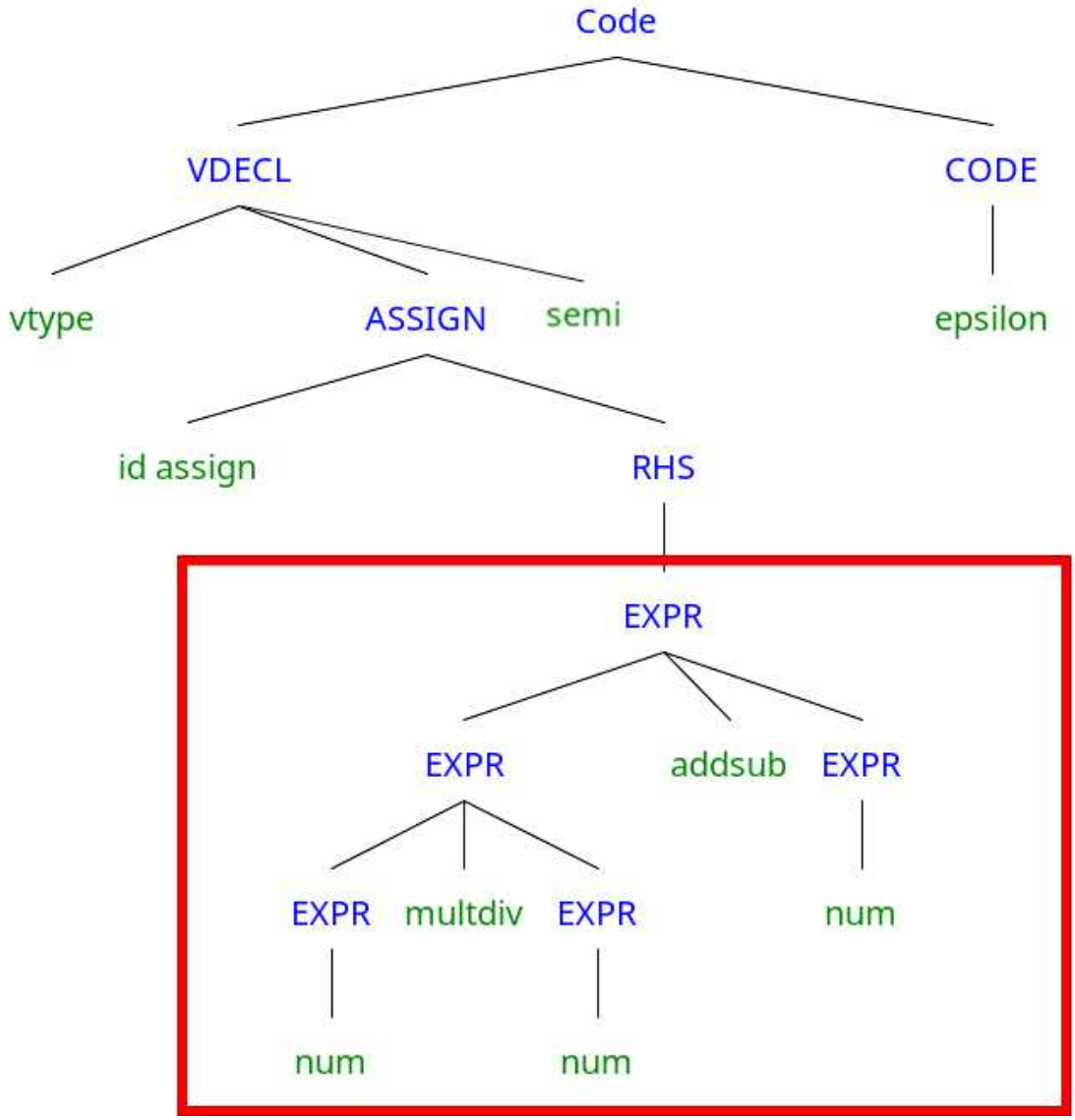
if input string is : **int y = 5 / 1 + 4;**

then using CFG, we can draw some parse trees. But, we can see two trees are made. (please ignore the id assign part)

<1st parse tree of int y = 5 / 1 + 4; >



< 2nd parse tree of int y = 5 / 1 + 4; >

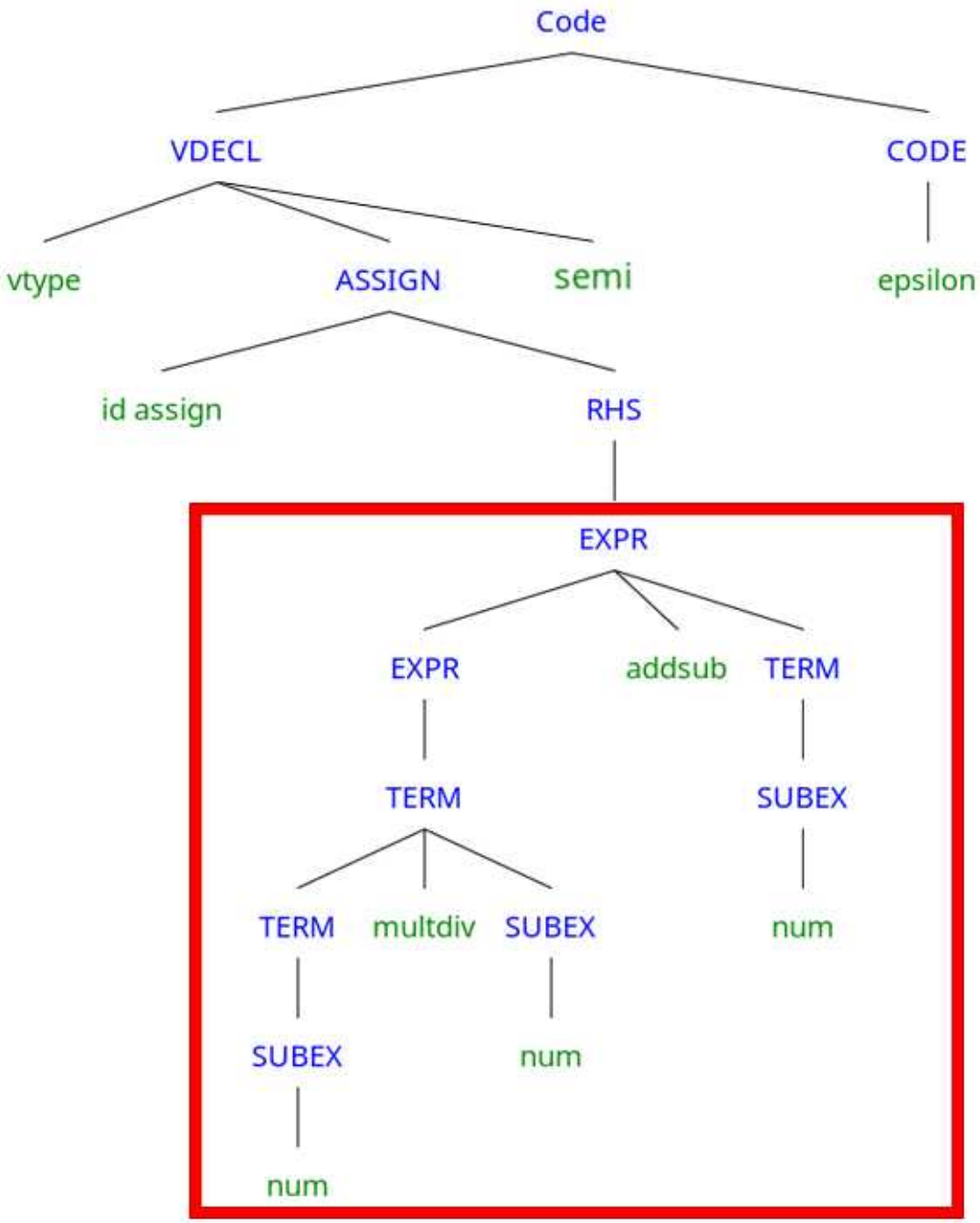


As we can see, for one string, there are two different trees. So by the rule, its syntax is ambiguous. We have to change it.

< before >
05: EXPR→EXPR addsub EXPR | EXPR multdiv EXPR
06: EXPR→lparen EXPR rparen | id | num
< after >
05: EXPR→EXPR addsub TERM | TERM
06: TERM→TERM multdiv SUBEX | SUBEX **(calculate mult/div first)**
07: SUBEX→lparen EXPR rparen | id | num

So, by using a changed CFG, we can make a CORRECT tree.

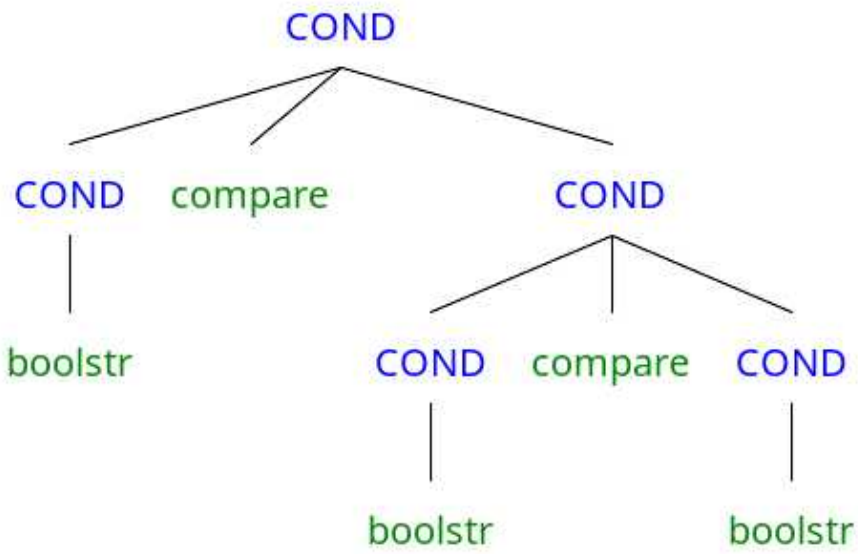
< correct parse tree >



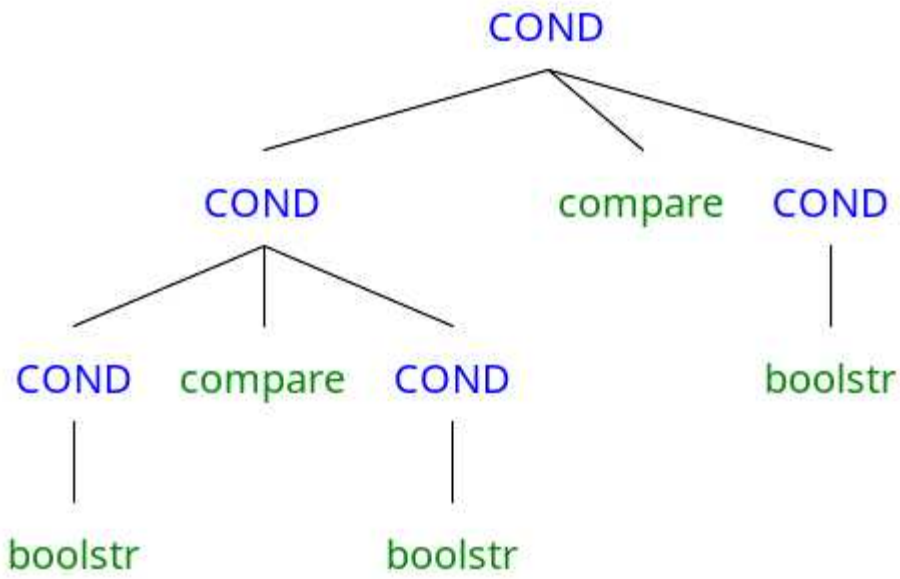
2. 14: COND→COND comp COND | boolstr

if input string is : **if(true == false != true){ BLOCK } ELSE**
also, using the CFG, we can make some parse trees. We just show the COND part and omit the rest.

< 1st parse tree >



< 2nd parse tree >

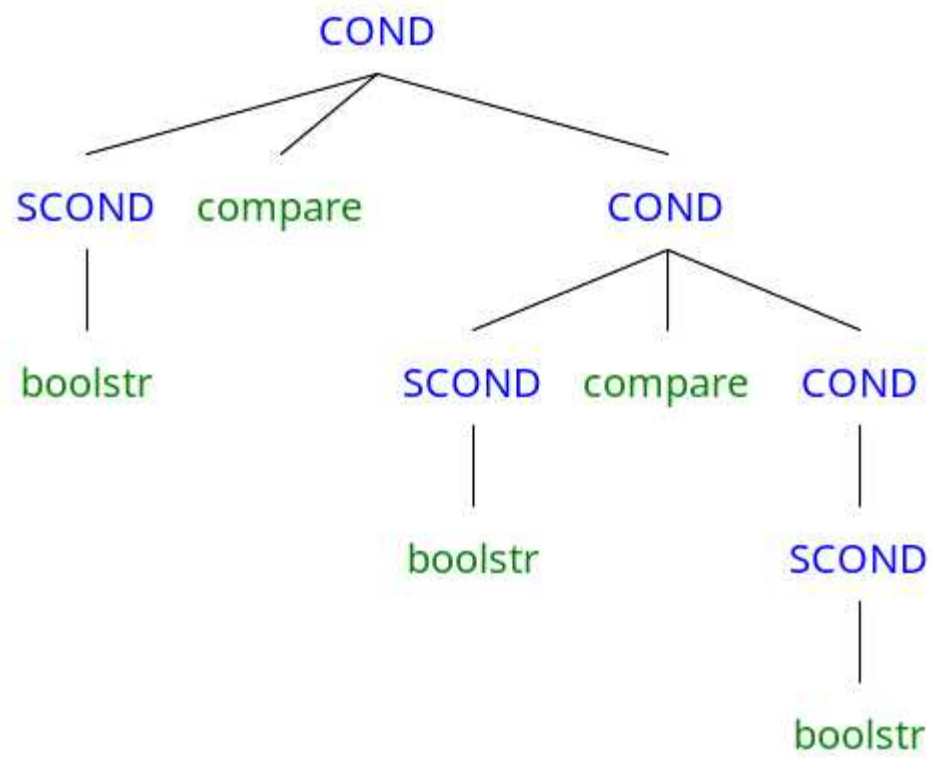


As we can see, this line of CFG also has some ambiguity. So we have to change it.

< before >
14: COND→COND comp COND | boolstr

< after >
14 COND→ SCOND comp COND | SCOND
15 SCOND→ boolstr

< correct parse tree >



So, the changed & correct CFG is this :

CFG G:

01: CODE→VDECL CODE | FDECL CODE | CDECL CODE| ϵ
02: VDECL→vtype id semi | vtype ASSIGN semi
03: ASSIGN→id assign RHS
04: RHS→EXPR | literal | character | boolstr
05: EXPR→EXPR addsub TERM | TERM
06: TERM→TERM multdiv SUBEX | SUBEX
07: SUBEX→lparen EXPR rparen | id | num
08: FDECL→vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
09: ARG→vtype id MOREARGS | ϵ
10: MOREARGS→comma vtype id MOREARGS | ϵ
11: BLOCK→STMT BLOCK | ϵ
12: STMT→VDECL | ASSIGN semi
13: STMT→if lparen COND rparen lbrace BLOCK rbrace ELSE
14: STMT→while lparen COND rparen lbrace BLOCK rbrace
15 COND→ SCOND comp COND | SCOND
16 SCOND→ boolstr
17: ELSE→else lbrace BLOCK rbrace | ϵ
18: RETURN→return RHS semi
19: CDECL→class id lbrace ODECL rbrace
20: ODECL→VDECL ODECL | FDECL ODECL | ϵ

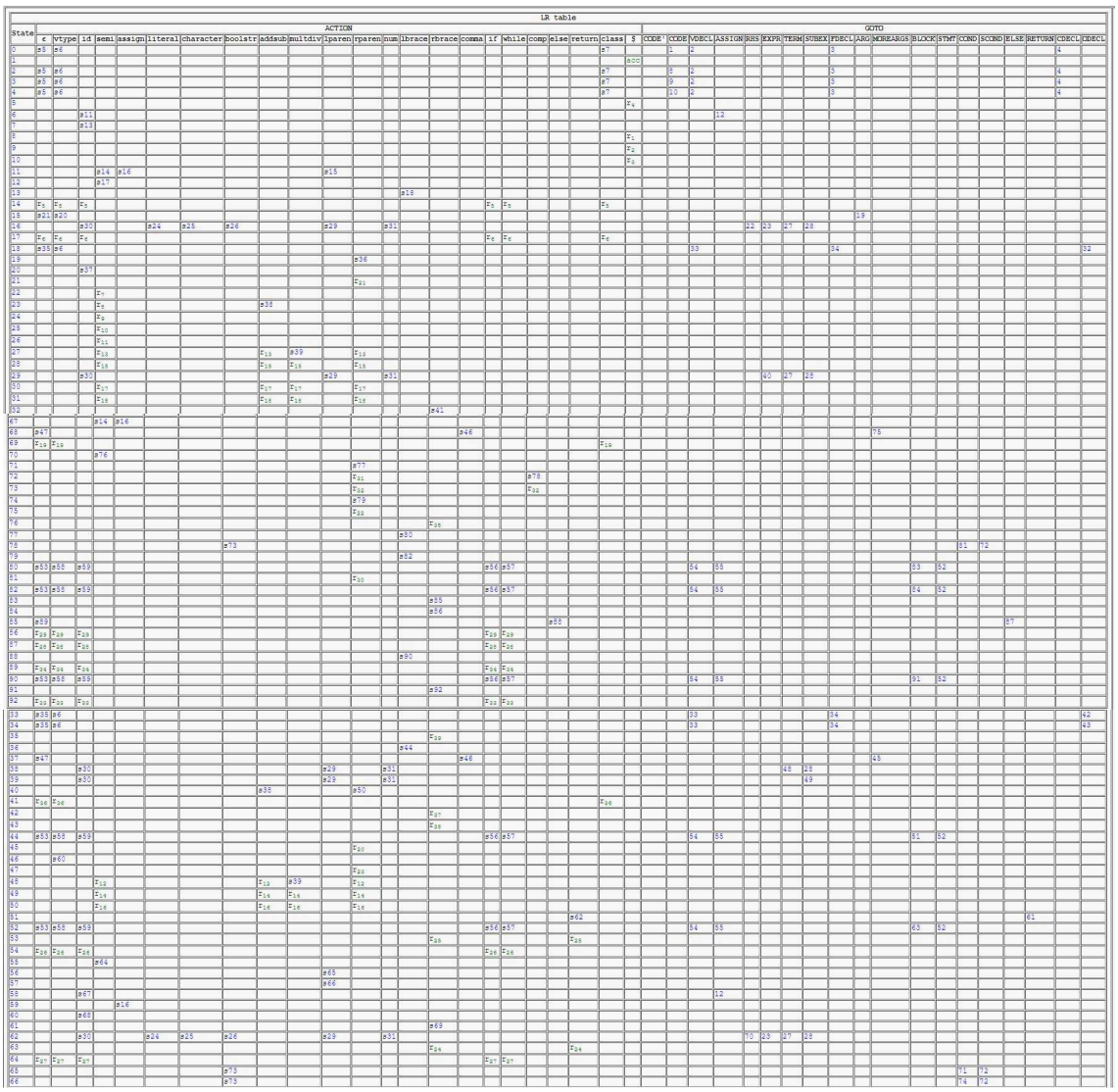
2.SLR table using CFG

To make SLR table, we have to put one line of CFG.

CODE' -> CODE

By using the web site, we can make SLR table.

< SLR table of unambiguous CFG >



(if you want to see detail, We prefer you to zoom it)

3. Some Changes in Lexical Analyzer

For the syntax analyzer, we have to change some Tokens in the lexical analyzer.

[Token name]

< before >		**< after >**
vtype		vtype
s_integer		num
s_char		character
boolstr		boolstr
lit_str		literal
ident		id
if/else/while/return/class	->	if/else/while/return/class
operator		addsub/multdiv
assign		assign
compare		comp
terminate		comma
lparen		lparen
rparen		rparen
lcb		lbrace
rcb		rbrace
lsb		lsb ([ )
rsb		rsb ( ])

But, it's not enough. In CFG, **there is epsilon**, so we have to put epsilon in the lexical analyzer. epsilon appears at a special moment.

< when epsilon appears >

01: CODE→VDECL CODE | FDECL CODE | CDECL CODE| ϵ
09: ARG→vtype id MOREARGS | ϵ
10: MOREARGS→comma vtype id MOREARGS | ϵ
11: BLOCK→STMT BLOCK | ϵ
17: ELSE→else lbrace BLOCK rbrace | ϵ
20: ODECL→VDECL ODECL | FDECL ODECL | ϵ

From this CFGs, we can make some if-statement to put epsilon.

if input str is : .. **( )**... then token is : ..lparen **epsilon** rparen...
if input str is : ..**{ return**.. then token is : ..lbrace **epsilon** return..
if input str is : ..**{ }**.. then token is : ..lbrace **epsilon** rbrace..
if input str is : **} }** then token is : ..rbrace **epsilon** rbrace..
if input str is : **} not else** then token is : ..lbrace **epsilon** notelse...
if input str is : **} return** then token is : ..rbrace **epsilon** return..
if input str is : **id)** then token is : .. id **epsilon** rparen.....

But, when it comes to **} not else,** there are few things to think.

1. not else means that the token of that lexeme is not else. It can be anything except ‘else’.

2. ‘If’ have to already appears in the input str.

3. in } not else, } is the first rbrace that appear in input str after ‘if’ appears.

So, we make some variable checkIF = 0, and when id detected, checkIF is 1, and when checkIF is 1 and } appear, checkIF is 2.

Therefore, when checking 'not else' in "{ not else", when checkIF is 2, we put epsilon and checkIF is 0. if not, we just get token of not else.

rest of a if statement follows the similar way.

Lastly, To make some change, we change the token name and put some if-statement to add epsilon.

< detect / add Epsilon >

```
case '}':
    if (checkIF == 1) {
        checkIF++;  // if and } id detected!     checkIF: 1 -> 2
    }
    if (nextToken == rbrace)
        addEpsilon();
    //  input str is : ... { }.... >> { + epsilon + }
    if (nextToken == lbrace)
        addEpsilon();
    //  input str is : ... ; }.... >> ; + epsilon + }
    if (nextToken == semi) {
        if (checkRETURN == 0)
            addEpsilon();
        else
            checkRETURN == 0;
    }
```

4. SLR parser in program

Then, here comes the main problem: How to make SLR parser in the program?

4-1. we have to gather up some tokens in one string from the output of the lexical analyzer.

the output of the lexical analyzer has a form kind of <token, value>, so if we want to get all of the tokens, we just have to extract the token from the <token, value>. We make some function to implement this:

```
================================================================
void extractToken(){
        in string, set all values to 0
        while file reach an EOF:
                get one line from file
                find the position of ‘,’
                and copy str from position of (<+1) to (,-1) = token
                attach to string and also attach ‘ ’ (blank)
}
================================================================
```

4-2. We have to make SLR table and stack

the SLR table : int table[][], but different from table, all blanks in table turns in to ER in table[][].

stack : to store some data of parser, we make the normal stack.

4-3 We make some functions that get one token from the string and return the number that matches to token.

```
================================================================
# define TOKEN number
short int getNextToken(){
        read the string, ignore the blank
        get one lexeme from the string
        if string is some TOKEN:
                return number that matches to that token
        ....
================================================================
```

4-4. We make some functions that work exactly like when we look at the SLR table and follow the table. We implement the shift and reduce by using a switch statement.

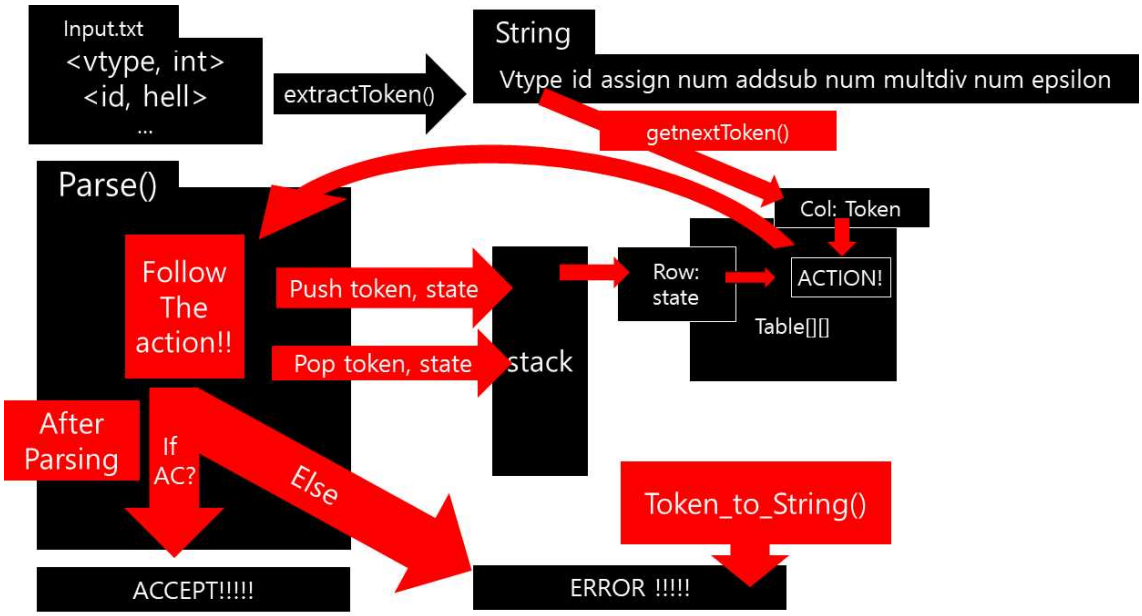
Shift : push token and Shift state(n of Sn)
Reduce : pop tokens and the |a| contents that X->a
ex) r5, 5 : COMP -> boolstr comp boolstr : 3
        >> pop(token, state) for 3 times >> pop 6 times

```
================================================================
void parse(){
        push 0 in stack
        while :
                token = getToken
                action = table[peek(&stack)][token]; // get current state
                switch(token):
```

```
                case S5:
                        push(token, &stack); // push token
                        push(5, &stack); // push state
                        break
                        ....
    if action >= 56:            // reduce operation
        while (action >= 56)
                action = table[peek(&stack)][token];
                switch (action):
                        case R1:
                           for (j = 0; j < 4; j++)
                               // R1 :  1 : CODE ->VDECL CODE
                               // pop ( token, state) for 2 times
                               // = pop 4 times
                                pop(&stack)
                           state = table[peek(&stack)][CODE]
                               // R1 :  1 : CODE ->VDECL CODE
                           if (state != ER)
                               push(CODE, &stack)
                               push(state, &stack)
                           else
                                error()
                           break
                           ...
```

==========

So, the whole execution work like this :



5. Test Result

Development Environment : Window10, visual studio
Execution Environment : WSL2, ubuntu, gcc compiler
(Assume that all files are in the 'download' folder)

< test input txt >

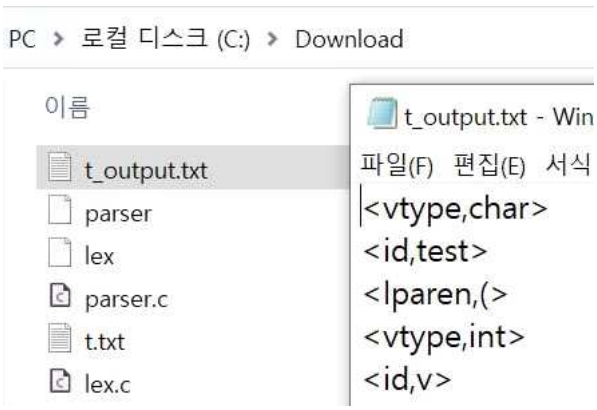
```
char test ( int v ) {
        int u;
        String str;
        char c = 'v';
        if( true == false) {
                while( true ) {
                        v = 5+1/6;
                }
        }
        return c;
}
```

< compile lexical analyzer, parser >

```
root@DESKTOP-61QKLAH:/mnt/c/download# gcc -o lex lex.c
root@DESKTOP-61QKLAH:/mnt/c/download# gcc -o parser parser.c
```

< lex result >

```
root@DESKTOP-61QKLAH:/mnt/c/download# ./lex t.txt
END Lexical Analyzer. Check the output.txt
```



< execute SLR parser - accept >

```
root@DESKTOP-61QKLAH:/mnt/c/download# ./parser t_output.txt
parsing Accept !!
```

As we can see, both lex and parser works well. But what if input has some syntax error?

< input txt which has some syntax error >

```
char test ( int v ) {
        int u;
        String str;
        char c = 'v';
        if( true == false) {
                while( true ) {
                        v = 5+1/6;
                }
        ]
        return c;
}
```

< lex, parser result >

```
root@DESKTOP-61QKLAH:/mnt/c/download# ./lex wrong.txt
END Lexical Analyzer. Check the output.txt
root@DESKTOP-61QKLAH:/mnt/c/download#  ./parser wrong_output.txt
ERROR : Wrong ] is used
 >>> END PARSER <<<
```

Because the lexical analyzer didn't know it has some syntax error, it just translates the code to tokens. But, SLR parser detects the error, and sends an error message.

< After making these programs..... >

This project is more difficult than the last project. There are some problems when implementing the SLR table. There are so much data, so when there an error, we have to check every piece of data to find the error. And there are also some details to think like add some epsilon, etc. Nevertheless, this project makes us feel the process of the SLR parser.