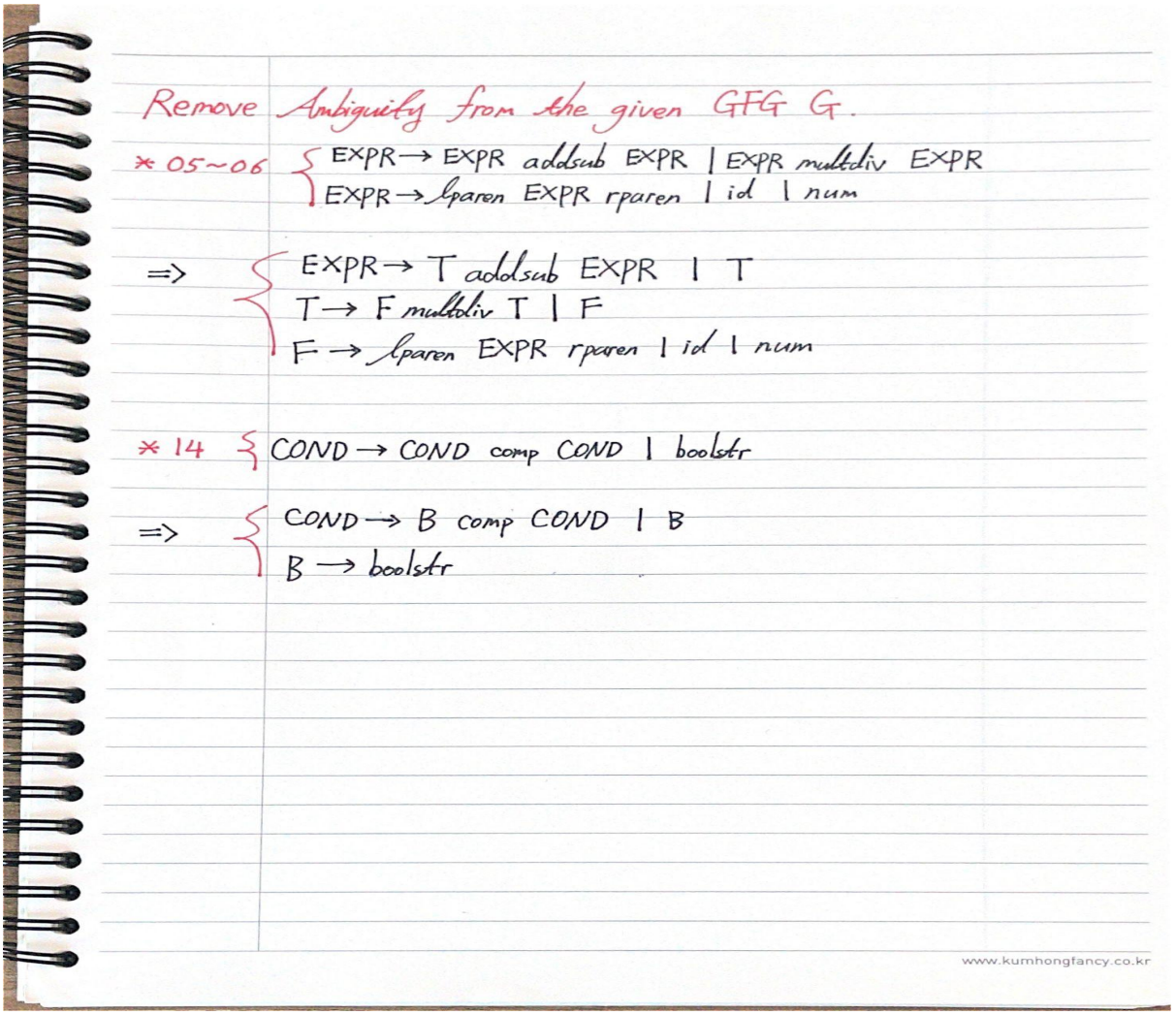
Compiler
Syntax Analyzer Report

Team 24.
20176959 김영권

1. Non-ambiguous CFG G

1	CODE -> VDECL CODE | FDECL CODE | CDECL CODE | ε
2	VDECL -> vtype id semi | vtype ASSIGN semi
3	ASSIGN -> id assign RHS
4	RHS -> EXPR | literal | character | boolstr
5	EXPR -> T addsub EXPR | T
6	T -> F multdiv T | F
7	F -> lparen EXPR rparen | id | num
8	FDECL -> vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace
9	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 -> B comp COND | B
16	B -> 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

- Removing the ambiguity of the given CFG G



2. SLR Parsing Table

Please visit the link below.

(I've set it to be rendered on the browser's document)

https://htmlpreview.github.io/?https://github.com/youngkwon02/Simplified-Java-Compiler/blob/main/doc/SLR_parsing_table.html

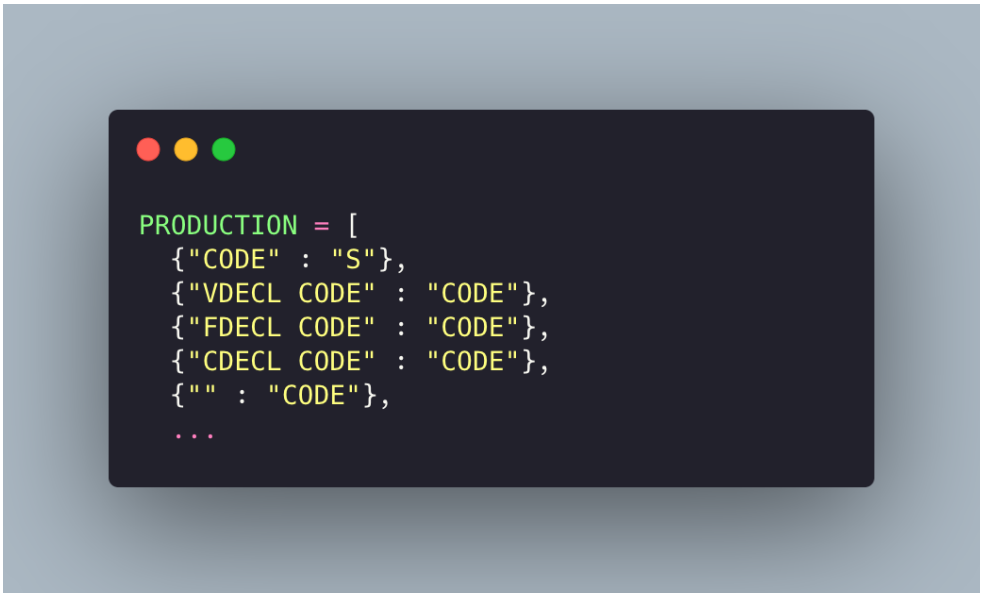
3. Some functions for successful syntax analyzing

- Token parser(Function) & Token convert table(Dataset)

I defined a token_parser function in the token_parser.py file.
Because I've used my own token name for the output of the lexical analyzer, I should have converted it to given token names like vtype, semi, etc..
So, the function token_parser refactor the output of the lexical analyzer to fit the input of the syntax analyzer referring to the token convert table.

- Production (Dataset)

I made a production.py file and defined an array named PRODUCTION.
The form of the PRODUCTION array is below.



```
PRODUCTION = [
  {"CODE" : "S"},
  {"VDECL CODE" : "CODE"},
  {"FDECL CODE" : "CODE"},
  {"CDECL CODE" : "CODE"},
  {"" : "CODE"},
  ...
```

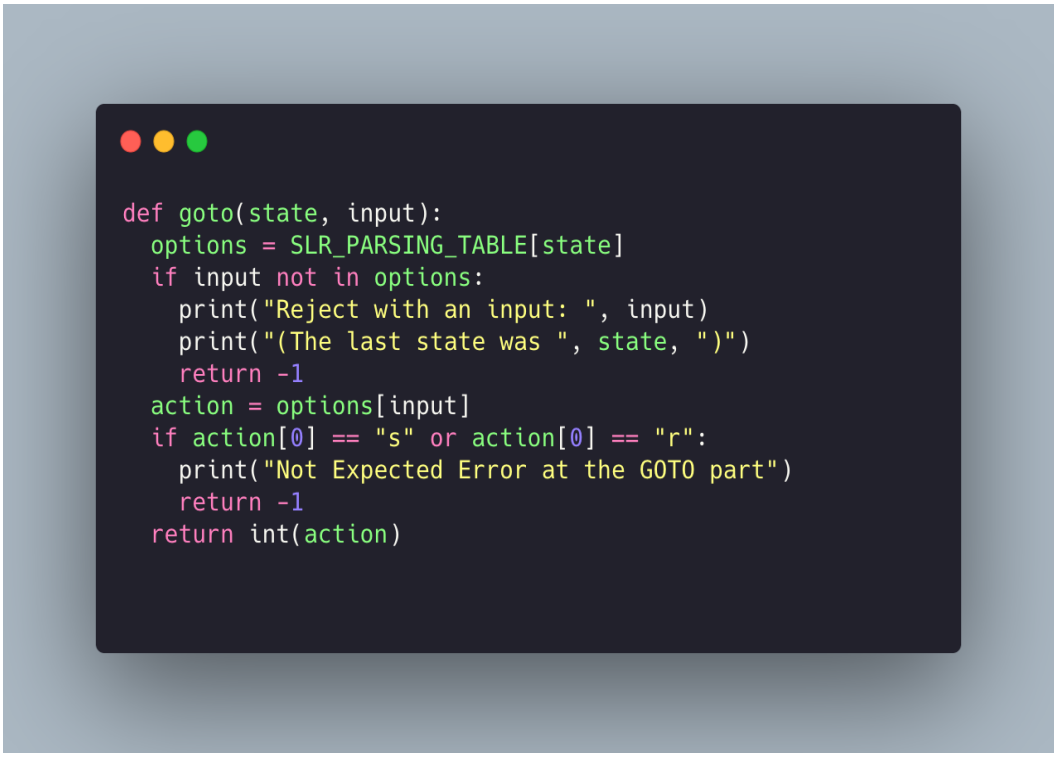
Like this, it has all non-ambiguous CFG G as a form of an array.
For example, the first element of the array is {"CODE" : "S"} and it means that there's an non-ambiguous CFG "S->CODE".
This PRODUCTION array is referred to during the reduction step.

- **SLR_PARSING_TABLE (Dataset)**

```
SLR_PARSING_TABLE = [
  # State 0
  {
    "vtype": "s5",
    "class": "s6",
    "$": "r4",
    "CODE": "1",
    "VDECL": "2",
    "FDECL": "3",
    "CDECL": "4"
  },
  # State 1
  {
    "$": "acc"
  },
  # State 2
  {
    "vtype": "s5",
    "class": "s6",
    "$": "r4",
    "CODE": "7",
    "VDECL": "2",
    "FDECL": "3",
    "CDECL": "4"
  },
  ...
```

I've expressed the created SLR parsing table as an upper object list.
The n th object of the list means the State n.
And each key of the object means the possible options for each state.
For example, at the state 0, if the next input is class, the value of the SLR_PARSING_TABLE[0]['class'] is s6, so we shift and change the state 6.
Instead, if the input was FDECL, the value of the …[0]['FDECL'] is 3, so we just change the state without any shift or reduction.

- **GOTO (Function)**



```
def goto(state, input):
  options = SLR_PARSING_TABLE[state]
  if input not in options:
    print("Reject with an input: ", input)
    print("(The last state was ", state, ")")
    return -1
  action = options[input]
  if action[0] == "s" or action[0] == "r":
    print("Not Expected Error at the GOTO part")
    return -1
  return int(action)
```

After doing reduction, we should check the GOTO(state, input).
This goto function will help the step.
If something is wrong, the function will print an error message and return -1.
But if all things are fine, it will return the next state to the main function.

- **Main (Function)**

In advance, all variables would be initialized in the main function such as state_stack and next_pointer.
We can guess that current state is the top value of the state stack, so for each step, we set the current state to state_stack[-1].
After deciding the current state, we should get all possible input for the current state. So referring to the dataset SLR_Parsing_Table, we can get all choices and we can define what to do according to the next_input.

We set the value of action to what to do.
If the action value's first character is s or r, it means we have to do shift or reduction. So by the if condition, I've separated the both cases.

1. *If the action[0] == “s”*
   This means we have to do shifts.
   So we change the next_pointer value to point to the next token.
   After increasing the next_pointer, we have to change the state to action[1:]
   (e.g. if the action is “s32”, the value of action[1:] would be 32)

2. *If the action[0] == “r”*
   This means we have to do reduction.
   So first, we have to decide which production rule we need.
   If the action value is “r5”, it means we will use fifth production for the reduction.
   So by accessing the PRODUCTION array, we can get the production that we need.
   Also, we should pop the state stack’s values as much as the length of the right side of the production. So by measuring the length of the production's right side, I also let it pop the stack.
   After the upper steps, we should check if the result is correct using the pre-defined goto function.

3. *If the action == “acc”*
   This means the given tokens are acceptable, so we print the accept message on the console and finish the program.

4. *If the given tokens are not acceptable...*
   This case would be detected during the options checking.
   In detail, we get all possible options referring to the SLR_Parsing_Table using the current state.
   After getting it, we would decide what we have to do according to the next_input.
   The options are given as an object form, so we can check if the next_input is acceptable or not by checking if the next_input is in the keys of the object.
   If the next_input is not in it, the error message will be printed on the console with the unacceptable input token.

```python
def main():
  state_stack = [0] # State를 누적 보관하는 Stack
  left_side = []
  next_pointer = 0 # 처리할 Input의 index 값 보관
  next_input = TOKEN_LIST[next_pointer] # next_pointer의 값을 통해 처리할 Input token에 접근
  counter = 1 # Error message 구체화를 위한 counter
  while(True):
    current_state = state_stack[-1] # Current State는 State_stack의 top value
    options = SLR_PARSING_TABLE[current_state] # SLR Parsing Table에서 모든 option을 읽어옴
    if next_input not in options: # 하지만 next_input이 처리할 수 없는 input일 경우
      print("Reject with an", counter, "th input: ", next_input) # 에러메시지 출력
      return -1
    action = options[next_input] # next_input이 current_state에서 처리 가능한 input이라면 action에 저장
    if action[0] == "s": # action이 s로 시작한다면, shift and goto
      left_side.append(next_input) # shift
      next_pointer += 1 # shift
      next_input = TOKEN_LIST[next_pointer] # shift
      counter += 1
      state_stack.append(int(action[1:])) # State Change
    elif action[0] == "r": # action이 r로 시작한다면, reduction
      prod = PRODUCTION[int(action[1:])] # 처리할 Production number를 prod에 저장
      alpha = list(prod.keys())[0] # Reduction할 Target terminal(or non-terminal)
      pop_num = len(alpha.split(' ')) # T->a라는 Production에서 |a|만큼 state를 pop하므로 pop할 횟수 저장
      if alpha == "": # |a| == 0인 예외적인 경우 처리
        pop_num = 0
      for pop in range(pop_num): # Pop 처리
        left_side.pop()
        state_stack.pop()
      left_side.append(list(prod.values())[0]) # Reduction result update
      new_state = goto(state_stack[-1], left_side[-1]) # Reduction 이후 GOTO 과정을 통해 correct한지 확
인하는 과정
      if new_state == -1: # GOTO 결과가 정상이 아니라면
        print("Reject with an", counter, "th input: ", next_input) # 에러메시지로 출력
        return -1
      state_stack.append(new_state) # new_state를 state_stack에 update
    elif action == "acc": # ACCEPT
      print("Accept")
      return 0
    else: # action이 s 혹은 r로 시작하지 않는다면 only state change
      state_stack.append(int(action)) # state_stack update
```

- DEMO

1. Acceptable case (test.java file)

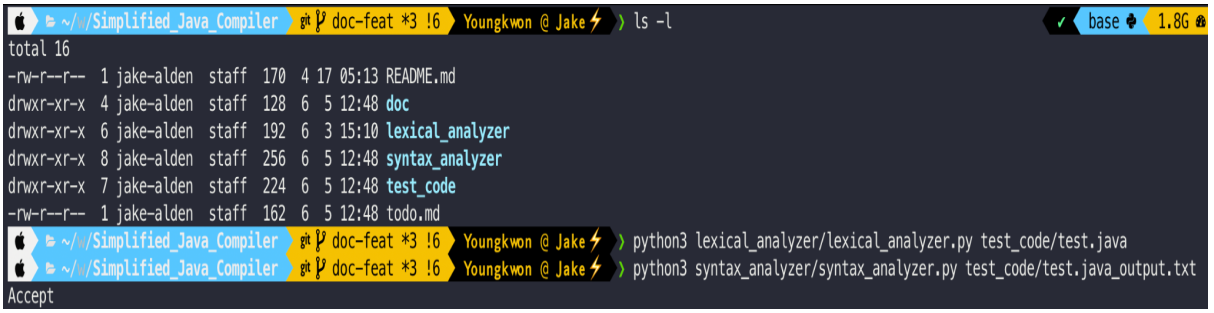
```
class test {
    String my_name = "Youngkwon Kim";
    int age = 25;
    char bloodType = "B";
    String univ = "Chung-Ang Univ";

    String getName(){
        return my_name;
    }

    int getAge(){
        if(true){
            int newAge = age + 1;
        }
        return age;
    }

    char getBloodType(){
        return bloodType;
    }

    String getUniv(){
        return univ;
    }
}
```



```
~/…/Simplified_Java_Compiler  doc-feat *3 !6  Youngkwon @ Jake  ls -l                    base  1.8G
total 16
-rw-r--r--  1 jake-alden  staff  170  4 17 05:13 README.md
drwxr-xr-x  4 jake-alden  staff  128  6  5 12:48 doc
drwxr-xr-x  6 jake-alden  staff  192  6  3 15:10 lexical_analyzer
drwxr-xr-x  8 jake-alden  staff  256  6  5 12:48 syntax_analyzer
drwxr-xr-x  7 jake-alden  staff  224  6  5 12:48 test_code
-rw-r--r--  1 jake-alden  staff  162  6  5 12:48 todo.md
~/…/Simplified_Java_Compiler  doc-feat *3 !6  Youngkwon @ Jake  python3 lexical_analyzer/lexical_analyzer.py test_code/test.java
~/…/Simplified_Java_Compiler  doc-feat *3 !6  Youngkwon @ Jake  python3 syntax_analyzer/syntax_analyzer.py test_code/test.java_output.txt
Accept
```

2. Unacceptable Case
   (There's no Left-Brace at the getName() method definition)

```
class error {
  String my_name = "Youngkwon Kim";
  int age = 25;
  char bloodType = "B";
  String univ = "Chung-Ang Univ";

  String getName()
    return my_name
  }

  int getAge(){
      if(true){
          int newAge = age + 1;
      }
      return age;
  }

  char getBloodType(){
      return bloodType;
  }

  String getUniv(){
      return univ;
  }
}
```

*** The error message means, "return" appeared at the position of the omitted left_brace.**

Thanks for reading my submission.
I will attach my Git-Hub repository link of this Lexer and Syntaxer project below.

For this semester, your totally perfect and clear lecture was so helpful for my self development.
Thanks for your great explanation about the overall compiler course!