



## Compiler Term Project

### (Syntax Analyzer)

Subject	Compiler
Class No.	01
Professor Name	Hyosu Kim
Team No.	11
Team Members	Hyomin Kim, Sanghwa Lee
Date	2021.06.05

## < Index >

<b>A. CFG .....</b>	<b>(3~4)</b>
0) Given CFG .....	(3)
1) Our CFG .....	(4)
 <b>B. SLR Parsing Table.....</b>	<b>(5~7)</b>
 <b>C. Code Implementation .....</b>	<b>(8~15)</b>
0) Overall Flow .....	(8)
1) lexical_analyzer.py (Revised) .....	(8)
2) SLR.py .....	(8~11)
3) syntax_analyzer.py .....	(12~15)
 <b>D. Result Screenshots .....</b>	<b>(16~19)</b>

## A. CFG

### 0) Given CFG ( non-left recursive but ambiguous )

01:  $\text{CODE} \rightarrow \text{VDECL CODE} \mid \text{FDECL CODE} \mid \text{CDECL CODE} \mid \epsilon$

02:  $\text{VDECL} \rightarrow \text{vtype id semi} \mid \text{vtype ASSIGN semi}$

03:  $\text{ASSIGN} \rightarrow \text{id assign RHS}$

04:  $\text{RHS} \rightarrow \text{EXPR} \mid \text{literal} \mid \text{character} \mid \text{boolstr}$

05:  $\text{EXPR} \rightarrow \text{EXPR addsub EXPR} \mid \text{EXPR multdiv EXPR}$

06:  $\text{EXPR} \rightarrow \text{lparen EXPR rparen} \mid \text{id} \mid \text{num}$

07:  $\text{FDECL} \rightarrow \text{vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace}$

08:  $\text{ARG} \rightarrow \text{vtype id MOREARGS} \mid \epsilon$

09:  $\text{MOREARGS} \rightarrow \text{comma vtype id MOREARGS} \mid \epsilon$

10:  $\text{BLOCK} \rightarrow \text{STMT BLOCK} \mid \epsilon$

11:  $\text{STMT} \rightarrow \text{VDECL} \mid \text{ASSIGN semi}$

12:  $\text{STMT} \rightarrow \text{if lparen COND rparen lbrace BLOCK rbrace ELSE}$

13:  $\text{STMT} \rightarrow \text{while lparen COND rparen lbrace BLOCK rbrace}$

14:  $\text{COND} \rightarrow \text{COND comp COND} \mid \text{boolstr}$

15:  $\text{ELSE} \rightarrow \text{else lbrace BLOCK rbrace} \mid \epsilon$

16:  $\text{RETURN} \rightarrow \text{return RHS semi}$

17:  $\text{CDECL} \rightarrow \text{class id lbrace ODECL rbrace}$

18:  $\text{ODECL} \rightarrow \text{VDECL ODECL} \mid \text{FDECL ODECL} \mid \epsilon$

## 1) Our CFG ( non-ambiguous )

01:  $\text{CODE} \rightarrow \text{VDECL CODE} \mid \text{FDECL CODE} \mid \text{CDECL CODE} \mid \epsilon$

02:  $\text{VDECL} \rightarrow \text{vtype id semi} \mid \text{vtype ASSIGN semi}$

03:  $\text{ASSIGN} \rightarrow \text{id assign RHS}$

04:  $\text{RHS} \rightarrow \text{EXPR} \mid \text{literal} \mid \text{character} \mid \text{boolstr}$

05:  $\text{EXPR} \rightarrow \text{TERM addsub EXPR} \mid \text{TERM}$

06:  $\text{TERM} \rightarrow \text{FACTOR multdiv TERM} \mid \text{FACTOR}$

07:  $\text{FACTOR} \rightarrow \text{lparen EXPR rparen} \mid \text{id} \mid \text{num}$

08:  $\text{FDECL} \rightarrow \text{vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace}$

09:  $\text{ARG} \rightarrow \text{vtype id MOREARGS} \mid \epsilon$

10:  $\text{MOREARGS} \rightarrow \text{comma vtype id MOREARGS} \mid \epsilon$

11:  $\text{BLOCK} \rightarrow \text{STMT BLOCK} \mid \epsilon$

12:  $\text{STMT} \rightarrow \text{VDECL} \mid \text{ASSIGN semi}$

13:  $\text{STMT} \rightarrow \text{if lparen COND rparen lbrace BLOCK rbrace ELSE}$

14:  $\text{STMT} \rightarrow \text{while lparen COND rparen lbrace BLOCK rbrace}$

15:  $\text{COND} \rightarrow \text{COND' comp COND} \mid \text{COND'}$

16:  $\text{COND'} \rightarrow \text{boolstr}$

17:  $\text{ELSE} \rightarrow \text{else lbrace BLOCK rbrace} \mid \epsilon$

18:  $\text{RETURN} \rightarrow \text{return RHS semi}$

19:  $\text{CDECL} \rightarrow \text{class id lbrace ODECL rbrace}$

20:  $\text{ODECL} \rightarrow \text{VDECL ODECL} \mid \text{FDECL ODECL} \mid \epsilon$

## B. SLR parsing table

Through our newly defined CFG which is non-ambiguous we constructed SLR parsing table at website : ( [SLR Parser Generator \(sourceforge.net\)](https://sourceforge.net/projects/slr-parser-generator/) ).

### [ Input ]

CODE' -> CODE

CODE -> VDECL CODE

CODE -> FDECL CODE

CODE -> CDECL CODE

CODE -> "

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 -> "

MOREARGS -> comma vtype id MOREARGS

MOREARGS -> "

BLOCK -> STMT BLOCK

BLOCK -> "

STMT -> VDECL

STMT -> ASSIGN semi

STMT -> if lparen COND rparen lbrace BLOCK rbrace ELSE

STMT -> while lparen COND rparen lbrace BLOCK rbrace

COND -> COND' comp COND

COND -> COND'

COND' -> boolstr

ELSE -> else lbrace BLOCK rbrace

ELSE -> "

RETURN -> return RHS semi

CDECL -> class id lbrace ODECL rbrace

ODECL -> VDECL ODECL

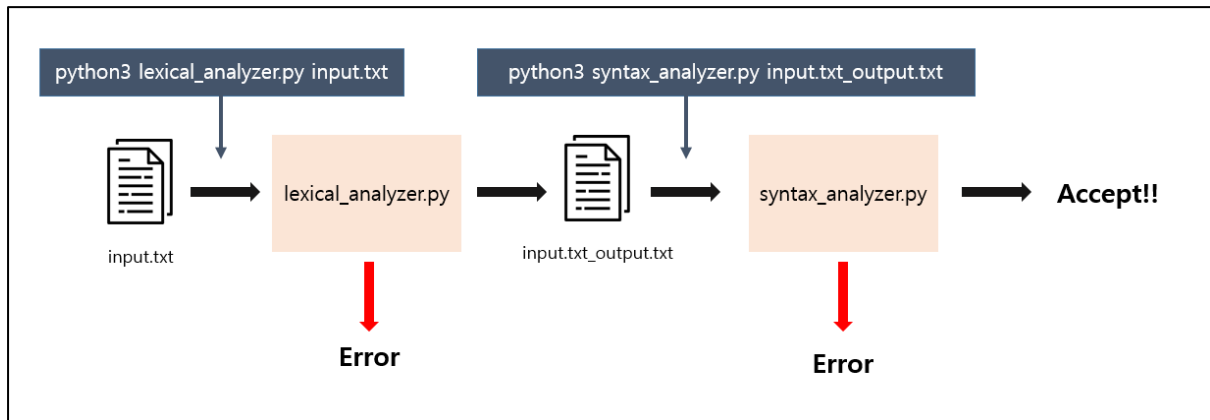
ODECL -> FDECL ODECL

ODECL -> "

[illegible]

## C. Code Implementation

### 0) Overall Flow



### 1) lexical\_analyzer.py (Revised)

```
40         if token == '$':                               # finish when meeting end of string
41             break
```

[ Line 40~41 ]

: Case for end symbol (\$) is added to finish lexical analyzing.

### 2) SLR.py

```
class SLRGrammar:
    def __init__(self, grammar):
        self.grammar = grammar
```

Class for CFG is defined. Our non-ambiguous CFG is added in form of 2D-list at `self.grammar`.



```
class SLRTable:
    def __init__(self, action, goto):
        self.action = action
        self.goto = goto
```

Class for SLR parsing table is defined. We used the table through website : ( [SLR Parser Generator \(sourceforge.net\)](https://sourceforge.net/projects/slr-parser-generator/) ) (image of SLR parsing table is at **B. SLR parsing table.** ) We used two variables for action table and goto table.

```
slr_grammar = SLRGrammar([["CODE", 'CODE'],
                           ['CODE', 'VDECL CODE'],
                           ['CODE', 'FDECL CODE'],
                           ['CODE', 'CDECL CODE'],
                           ['CODE', ''],
                           ['VDECL', 'VTYPE ID SEMI'],
                           ['VDECL', 'VTYPE GOTO_ASSIGN SEMI'],
                           ['GOTO_ASSIGN', 'ID ASSIGN RHS'],
                           ['RHS', 'EXPR'],
                           ['RHS', 'STRING'],
                           ['RHS', 'CHAR'],
                           ['RHS', 'BOOL'],
                           ['EXPR', 'TERM ADDSUB EXPR'],
                           ['EXPR', 'TERM'],
                           ['TERM', 'FACTOR MULTDIV TERM'],
                           ['TERM', 'FACTOR'],
                           ['FACTOR', 'LPAREN EXPR RPAREN'],
                           ['FACTOR', 'ID'],
                           ['FACTOR', 'INTEGER'],
                           ['FDECL', 'VTYPE ID LPAREN ARG RPAREN LBRACE BLOCK GOTO_RETURN RBRACE'],
                           ['ARG', 'VTYPE ID MOREARGS'],
                           ['ARG', ''],
                           ['MOREARGS', 'COMMA VTYPE ID MOREARGS'],
                           ['MOREARGS', ''],
                           ['BLOCK', 'STMT BLOCK'],
                           ['BLOCK', ''],
                           ['STMT', 'VDECL'],
                           ['STMT', 'GOTO_ASSIGN SEMI'],
```

```
['STMT', 'IF LPAREN COND RPAREN LBRACE BLOCK RBRACE GOTO_ELSE'],
['STMT', 'WHILE LPAREN COND RPAREN LBRACE BLOCK RBRACE'],
['COND', 'COND COMP COND'],
['COND', 'COND'],
['COND', 'BOOL'],
['GOTO_ELSE', 'ELSE LBRACE BLOCK RBRACE'],
['GOTO_ELSE', ''],
['GOTO_RETURN', 'RETURN RHS SEMI'],
['CDECL', 'CLASS ID LBRACE ODECL RBRACE'],
['ODECL', 'VDECL ODECL'],
['ODECL', 'FDECL ODECL'],
['ODECL', '']])
```

Our CFG is added in form of 2D list. Derivation and derivative are initialized as string value.

(※ ASSIGN, ELSE, RETURN non-terminal are included in both action table and goto table. Thus we named GOTO\_ASSIGN, GOTO\_ELSE, GOTO\_RETURN for goto table non-terminals and ASSIGN, ELSE, RETURN for action table non-terminals. )

**Form : [ [ 'Derivation', 'Derivative' ] ]**

**EX)** CODE -> CDECL CODE : [['CODE', 'CDECL CODE']]

CODE ->  $\epsilon$  : [['CODE', '']]

```
slr_table = SLRTable([[[[0, 'VTYPE', 's5'], [0, 'CLASS', 's6'], [0, '$', 'r4']],
[[1, '$', 'acc']],
[[2, 'VTYPE', 's5'], [2, 'CLASS', 's6'], [2, '$', 'r4']],
[[3, 'VTYPE', 's5'], [3, 'CLASS', 's6'], [3, '$', 'r4']],
[[4, 'VTYPE', 's5'], [4, 'CLASS', 's6'], [4, '$', 'r4']],
[[5, 'ID', 's10']],
[[6, 'ID', 's12']],
[[7, '$', 'r1']],
[[8, '$', 'r2']],
[[9, '$', 'r3']],
[[10, 'SEMI', 's13'], [10, 'ASSIGN', 's15'], [10, 'LPAREN', 's14']],
[[11, 'SEMI', 's16']],
[[12, 'LBRACE', 's17']],
[[13, 'VTYPE', 'r5'], [13, 'ID', 'r5'], [13, 'RBRACE', 'r5'], [13, 'IF', 'r5'], [13, 'WHILE', 'r5'], [13, 'RETURN', 'r5'], [13, 'CLASS', 'r5'], [13, 'VTYPE', 's19'], [14, 'RPAREN', 'r21']],
[[14, 'VTYPE', 's19'], [14, 'RPAREN', 'r21']],
[[15, 'ID', 's20'], [15, 'STRING', 's22'], [15, 'CHAR', 's23'], [15, 'BOOL', 's24'], [15, 'LPAREN', 's27'], [15, 'INTEGER', 's29']],
[[16, 'VTYPE', 'r6'], [16, 'ID', 'r6'], [16, 'RBRACE', 'r6'], [16, 'IF', 'r6'], [16, 'WHILE', 'r6'], [16, 'RETURN', 'r6'], [16, 'CLASS', 'r6'], [16, 'VTYPE', 's5'], [17, 'RPAREN', 'r39']],
[[17, 'VTYPE', 's5'], [17, 'RPAREN', 'r39']],
[[18, 'RPAREN', 's33']],
[[19, 'ID', 's34']],
[[20, 'SEMI', 'r7']],
[[21, 'SEMI', 'r8']],
[[22, 'SEMI', 'r9']],
[[23, 'SEMI', 'r10']],
[[24, 'SEMI', 'r11']],
[[25, 'SEMI', 'r13'], [25, 'ADDSUB', 's35'], [25, 'RPAREN', 'r13']],
[[26, 'SEMI', 'r15'], [26, 'ADDSUB', 'r15'], [26, 'MULTDIV', 's36'], [26, 'RPAREN', 'r15']],
[[27, 'ID', 's28'], [27, 'LPAREN', 's27'], [27, 'INTEGER', 's29']],
[[28, 'SEMI', 'r17'], [28, 'ADDSUB', 'r17'], [28, 'MULTDIV', 'r17'], [28, 'RPAREN', 'r17']]]])
```

SLR parsing table (Action table)

```
[[[0, 'CODE', 1], [0, 'VDECL', 2], [0, 'FDECL', 3], [0, 'CDECL', 4]],
[],
[[2, 'CODE', 7], [2, 'VDECL', 2], [2, 'FDECL', 3], [2, 'CDECL', 4]],
[[3, 'CODE', 8], [3, 'VDECL', 2], [3, 'FDECL', 3], [3, 'CDECL', 4]],
[[4, 'CODE', 9], [4, 'VDECL', 2], [4, 'FDECL', 3], [4, 'CDECL', 4]],
[[5, 'GOTO_ASSIGN', 11]],
[],
[],
[],
[],
[],
[],
[],
[],
[[14, 'ARG', 18]],
[[15, 'RHS', 20], [15, 'EXPR', 21], [15, 'TERM', 25], [15, 'FACTOR', 26]],
[],
[[17, 'VDECL', 31], [17, 'FDECL', 32], [17, 'ODECL', 30]],
```

SLR parsing table (Goto table)

SLR parsing table has lots of value so we captured it partially. We used 3D list to initialize value of SLRTable.

**Action table Form : [[[State\_num, 'State', 'Action']]]**

**EX)**

State	
	vtype
0	s5

: [[[0, 'VTYPE', 's5']]]

**Goto table Form : [[[State\_num, 'State', goto\_state]]]**

**EX)** (row at state '0')

GOTO			
	FDECL	ARG	MORE
3			

: [[[0, 'FDECL', 3]]]

### 3) syntax\_analyzer.py

```
1  import argparse
2  from SLR import *
3
4
5  def syntax_analyzer(data):
6      data = data.split()          # split output of lexical analyzer by blank and reset to analyze easier
7      del data[-1]
8      data.append('$')
9      data.append('$')
10     index = 1                    # initialize index of starting token
11     state_stack = [0]           # stack that stores states and last element is current state
12     left_substring = []         # store left substring that already shifted before
13     while True:
14         if data[index] == 'OP':    # distinguish +, - and *, / operators
15             if data[index + 1] == '+' or data[index + 1] == '-':
16                 data[index] = 'ADDSUB'
17             else:
18                 data[index] = 'MULTDIV'
19         for action in slr_table.action[state_stack[-1]]:
20             if data[index] == action[1]:    # find action corresponding to terminal value
21                 if action[2][0] == 's':    # shift and goto decision
22                     state_stack.append(int(action[2][1]))
23                     left_substring.append(data[index])
24                     index += 2
25                 elif action[2][0] == 'r':    # reduce decision
26                     for non_terminal in reversed(slr_grammar.grammar[int(action[2][1:]))[1].split()):
27                         if non_terminal == left_substring[-1]:
28                             del left_substring[-1]
29                             state_stack.pop()
30                         else:
31                             print(f"There's no goto at state {state_stack[-1]} for {left_substring}")
32                             return
33
34                     left_substring.append(slr_grammar.grammar[int(action[2][1:]))[0])
35                     for goto in slr_table.goto[state_stack[-1]]:    # find goto corresponding to non-terminal value
36                         if left_substring[-1] == goto[1]:
37                             state_stack.append(goto[2])
38                         else:
39                             print("Accept")
40                             return
41                             break
42                     else:
43                         print("reject")
44                         print(f"There's no action at state {state_stack[-1]} for {data[index]}")
45                         return
46
47     parser = argparse.ArgumentParser()
48     parser.add_argument("input_file_name", help="file name of your input code")
49     args = parser.parse_args()
50     fr = open(args.input_file_name, 'r')
51     lexical_output = fr.read()
52     syntax_analyzer(lexical_output)
```

```

5  def syntax_analyzer(data):
6      data = data.split()           # split output of lexical analyzer by blank and reset to analyze easier
7      del data[-1]
8      data.append('$')
9      data.append('$')
10     index = 1                     # initialize index of starting token
11     state_stack = [0]             # stack that stores states and last element is current state
12     left_substring = []           # store left substring that already shifted before

```

Syntax\_analyzer function gets **data** argument, a result of *lexical\_analyzer.py*.

First, let's see example of *lexical\_analyzer.py* result.

	Token	Lexeme
1	-----	
2	CLASS	class
3	ID	Test
4	LBRACE	{
5	VTYPE	int
6	ID	sum
7	LPAREN	(
8	VTYPE	int
9	ID	a
10	COMMA	,
11	VTYPE	int
12	ID	b
13	RPAREN	)

230	SEMI	;
231	RBRACE	}
232	RBRACE	}
233	-----	

This is example result of lexical\_analyzer.py. If we use data.split(), it will derive following value.

data[0] = -----

data[1] = CLASS

data[2] = class

data[3] = ID

data[4] = Test

.....

data[-1] = -----

Therefore, we deleted value at data[-1] at line 7.

Then we appended '\$' twice, one for token and one for lexeme.

Finally we initialized index variable, stack list, which will be used for bottom-up parsing, and left substring list.

```

13     while True:
14         if data[index] == 'OP':                # distinguish +, - and *, / operators
15             if data[index + 1] == '+' or data[index + 1] == '-':
16                 data[index] = 'ADDSUB'
17             else:
18                 data[index] = 'MULTDIV'

```

Now we start bottom-up parsing from line 13. First, we check whether token is 'OP' or not. Our lexical analyzer determines every +, -, \*, / operators as 'OP'. Therefore, we need additional process to distinguish +,- operator and \*,/ operator to determine operator priority. To do this, after 'OP' is checked, we check lexeme value( data[index+1] ) to distinguish whether it is 'ADDSUB' or 'MULTDIV'.

```

19     for action in slr_table.action[state_stack[-1]]:
20         if data[index] == action[1]:            # find action corresponding to terminal value
21             if action[2][0] == 's':            # shift and goto decision
22                 state_stack.append(int(action[2][1]))
23                 left_substring.append(data[index])
24                 index += 2
25             elif action[2][0] == 'r':          # reduce decision
26                 for non_terminal in reversed(slr_grammar.grammar[int(action[2][1:]))[1].split()):
27                     if non_terminal == left_substring[-1]:
28                         del left_substring[-1]
29                         state_stack.pop()
30                 else:
31                     print(f"There's no goto at state {state_stack[-1]} for {left_substring}")
32                     return
33                 left_substring.append(slr_grammar.grammar[int(action[2][1:]))[0])
34                 for goto in slr_table.goto[state_stack[-1]]:    # find goto corresponding to non-terminal value
35                     if left_substring[-1] == goto[1]:
36                         state_stack.append(goto[2])
37             else:
38                 print("Accept")
39                 return
40             break
41         else:
42             print("reject")
43             print(f"There's no action at state {state_stack[-1]} for {data[index]}")
44             return

```

#### [ Line 19~44 ]

We check the last value of state\_stack list and find it at slr\_table's action table.

At first, state\_stack[-1] is 0, so it checks state 0 of action table.

#### [ Line 20~40 ]

If data[index] value (terminal value) is same with action[1], such as [0, 'VTYPE', 's5'], this value, we check action[2][0] [0, 'VTYPE', 's5'] to determine whether to shift or reduce.

#### [ Line 21~24 ]

If `a[2][0]` is 's', it means 'shift'. Therefore we append `a[2][1]` `[0, 'VTYPE', 's5']` at `state_stack` list and append current token at `left_substring` list. Then increase index value by 2 to skip current lexeme and check next token.

#### [ Line 25~36 ]

If `a[2][0]` is 'r', it means 'reduce'. Therefore we get value of `int(action[2][1:])` `[20, 'SEMI', 'r7']` and check `slr_grammar.grammar[int(action[2][1:])[1].split(), derivatives of CFG such as` `['CODE', 'VDECL', 'CODE'],`

#### [ Line 27~29 ]

If last value of left substring is in chosen derivatives, we delete last value of left substring and pop value of state stack. This process will be repeated as the length of derivatives.

#### [ Line 30~32 ]

If last value of left substring are not in chosen derivatives, it prints error report that there are no corresponding states and stops parsing.

#### [ Line 33 ]

Finally we append the reduced non-terminal value such as `['CODE', 'VDECL', 'CODE'],`

#### [ Line 34~36 ]

Finally we check goto table at `state_stack`'s last value and check if last value of `left_substring` is in `goto[1]` such as `[0, 'CODE', 1]`. And if it is true we append the value of `goto[2]` such as `[0, 'CODE', 1]` at `state_stack` list.

#### [ Line 37~39 ]

If `action[2][0]` is neither 's' or 'r', it means case such as `[1, '$', 'acc']` which means "Accept".

#### [ Line 41~44 ]

If there is no corresponding action table, it prints reject error.

## D. Result Screenshots

[ Every possible test case without any error ]

### Command Line

```
os161@ubuntu:~/compiler_term_project/compiler-main$ python3 lexical_analyzer.py input.txt
os161@ubuntu:~/compiler_term_project/compiler-main$ python3 syntax_analyzer.py input.txt_output.txt
Accept
```

### Input File ( Input.txt )

```
class Test {
    int sum(int a, int b) {
        return a+b;
    }
    int noArg(){
        return 0;
    }
    string returnString(){
        return "temp";
    }
    int returnInt(){
        return 2;
    }
    char returnChar(){
        return 'a';
    }
    boolean returnTrue(){
        return true;
    }
    boolean returnFalse(){
        return false;
    }
}

int main(string s) {
    int a = 3;
    int b = 4;
    int c;
    char d;
    char e = 'a';
    boolean f;
    boolean g = true;
    boolean h = false;
    string i;
    string j = "temp";
    if (true >= false){
        a = 2;
    }
    else{
        if(true > true){
            b=3;
        }
        else{
        }
    }

    if(true==false){
    }
    if(true!=false){
    }
    if(true<false){
    }
    if(true<=false){
    }
    c = a+b;
    c = a-b;
    c = a/b;
    c = a*b;
    while(true){

    }
    while(false<true){
        c = 2;
    }
    return 'a';
}
```



## Output File ( input.txt\_output.txt )

```
-----  
CLASS      class  
ID          Test  
LBRACE     {  
VTYPE      int  
ID          sum  
LPAREN     (  
VTYPE      int  
ID          a  
COMMA      ,  
VTYPE      int  
ID          b  
RPAREN     )  
LBRACE     {  
RETURN     return  
ID          a  
OP          +  
ID          b  
SEMI       ;  
RBRACE     }  
VTYPE      int  
ID          noArg  
LPAREN     (  
RPAREN     )  
LBRACE     {  
RETURN     return  
INTEGER    0  
SEMI       ;  
RBRACE     }  
VTYPE      string  
ID          returnString  
LPAREN     (  
RPAREN     )  
LBRACE     {  
RETURN     return  
STRING     "temp"  
SEMI       ;  
RBRACE     }  
VTYPE      int  
ID          returnInt  
LPAREN     (  
RPAREN     )  
LBRACE     {  
RETURN     return  
INTEGER    2  
SEMI       ;  
RBRACE     }  
VTYPE      char
```

```
ID          e  
ASSIGN     =  
CHAR       'a'  
SEMI       ;  
VTYPE      boolean  
ID          f  
SEMI       ;  
VTYPE      boolean  
ID          g  
ASSIGN     =  
BOOL       true  
SEMI       ;  
VTYPE      boolean  
ID          h  
ASSIGN     =  
BOOL       false  
SEMI       ;  
VTYPE      string  
ID          i  
SEMI       ;  
VTYPE      string  
ID          j  
ASSIGN     =  
STRING     "temp"  
SEMI       ;  
IF          if  
LPAREN     (  
BOOL       true  
COMP       >=  
BOOL       false  
RPAREN     )  
LBRACE     {  
ID          a  
ASSIGN     =  
INTEGER    2  
SEMI       ;  
RBRACE     }  
ELSE       else  
LBRACE     {  
IF          if  
LPAREN     (  
BOOL       true  
COMP       >  
BOOL       true  
RPAREN     )  
LBRACE     {
```

```
IF          if  
LPAREN     (  
BOOL       true  
COMP       <  
BOOL       false  
RPAREN     )  
LBRACE     {  
IF          if  
LPAREN     (  
BOOL       true  
COMP       <=  
BOOL       false  
RPAREN     )  
LBRACE     {  
RBRACE     }  
ID          c  
ASSIGN     =  
ID          a  
OP          +  
ID          b  
SEMI       ;  
ID          c  
ASSIGN     =  
ID          a  
OP          -  
ID          b  
SEMI       ;  
ID          c  
ASSIGN     =  
ID          a  
OP          /  
ID          b  
SEMI       ;  
ID          c  
ASSIGN     =  
ID          a  
OP          *  
ID          b  
SEMI       ;  
WHILE      while  
LPAREN     (  
BOOL       true  
RPAREN     )  
LBRACE     {  
RBRACE     }
```

```
RPAREN    )
LBRACE    {
RBRACE    }
WHILE     while
LPAREN    (
BOOL      false
COMP      <
BOOL      true
RPAREN    )
LBRACE    {
ID        c
ASSIGN    =
INTEGER   2
SEMI      ;
RBRACE    }
RETURN    return
CHAR      'a'
SEMI      ;
RBRACE    }
RBRACE    }
```

-----

[ Input with an error ]

## Command Line

```
os161@ubuntu:~/compiler_term_project/compiler-main$ python3 lexical_analyzer.py wrong_input.txt
os161@ubuntu:~/compiler_term_project/compiler-main$ python3 syntax_analyzer.py wrong_input.txt_output.txt
reject
There's no action at state 41 for BOOL
```

Input File ( wrong\_input.txt )

```
int main(){
    false = false+1;
}
```

Output File ( wrong\_input.txt\_output.txt )

```
-----
VTYPE      int
ID          main
LPAREN     (
RPAREN     )
LBRACE     {
BOOL       false
ASSIGN     =
BOOL       false
OP         +
INTEGER    1
SEMI       ;
RBRACE     }
-----
```