Compiler Term Project

Subject	Compiler
Class No.	01
Professor Name	Hyosu Kim
Team No.	11
Team Members	Hyomin Kim, Sanghwa Lee
Date	2021.04.12

< Index >

A. Definition of Tokens and Their Regular Expressions

0) Predefined Tokens

- letter :

Token name : letter

Regular Expression :

a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

- digit :

Token name : digit

Regular Expression : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- positive :

Token name : positive

Regular Expression : 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

1) Variable type

Token Name : VTYPE

Regular Expression : int | char | boolean | String

2) Signed integer

Token Name : INT

Regular Expression : 0 | ((- | ϵ) $positive^{+}$)

3) Single character

Token Name : CHAR

Regular Expression : ‘letter | digit | blank’

4) Boolean string

Token Name : BOOL

Regular Expression : true | false

5) Literal string

Token Name : STRING

Regular Expression : $“(digits\ | letter \mid blank)^{+}”$

6) An identifier of variables and functions

Token Name : ID

Regular Expression : (letters | _)(digits | letters | _)*

7) Keywords for special statements

1. Token Name : IF

Regular Expression : if

2. Token Name : ELSE

Regular Expression : else

3. Token Name : WHILE

Regular Expression : while

4. Token Name : CLASS

Regular Expression : class

5. Token Name : RETURN

Regular Expression : return

➔ Merged Regular Expression : if | else | while | class | return

8) Arithmetic operators

Token Name : OP

Regular Expression : + | - | * | /

9) Assignment operator

Token Name : ASSIGN

Regular Expression : =

10) Comparison operators

Token Name : COMP

Regular Expression : ((< | >)(= | e)) | ((! | =) =)

11) A terminating symbol of statements :

Token Name : SEMI

Regular Expression : ;

12) A pair of symbols for defining area/scope of variables and functions : { and }

Token Name : BRACE { : LBRACE } : RBRACE

Regular Expression : { | }

13) A pair of symbols for indicating a function/statement : (and)

Token Name : PAREN (: LPAREN) : RPAREN

Regular Expression : (|)

14) A pair of symbols for using an array : [and]

Token Name : BRACKET [: LBRACKET] : RBRACKET

Regular Expression : [|]

15) A symbol for separating input arguments in functions ; ,

Token Name : COMMA

Regular Expression : ,

16) Whitespaces :

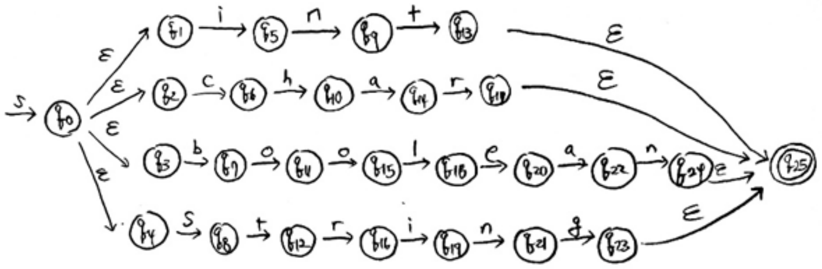
Token Name : WHITESPACE

Regular Expression : $(\backslash t \mid \backslash n \mid blank)^{+}$

B. NFA, DFA transition graph

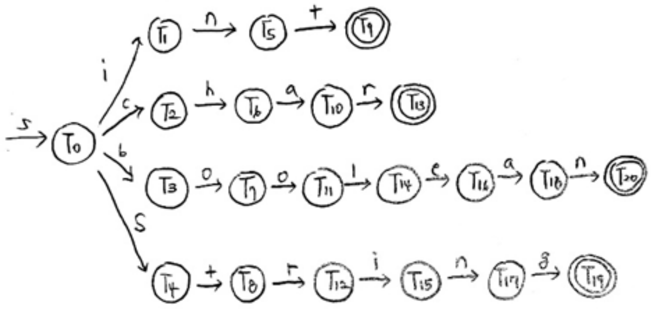
1) Variable type

TYPE (NFA)



s
q0
ε
q1 i q5 n q9 t q13
q2 c q6 h q10 a q14 r q17
q3 b q7 o q11 o q15 l q18 e q20 a q22 n q24
q4 s q8 t q12 r q16 i q19 n q21 g q23
ε
q25

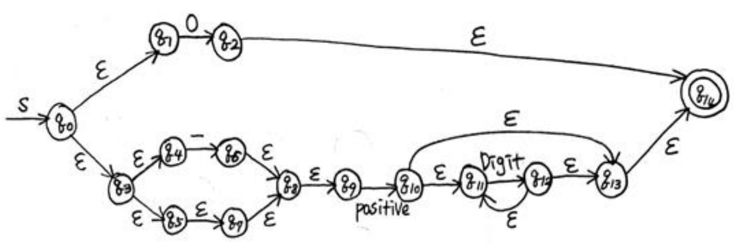
TYPE (DFA) ·



s
T0
i
T1 n T5 t T9
c
T2 h T6 a T10 r T13
b
T3 o T7 o T11 l T14 e T16 a T18 n T20
S
T4 t T8 r T12 i T15 n T17 g T19

2) Signed integer

int (NFA)



int (DFA)

$T_0 = e\text{-closure}(q_0) = \{q_0, q_1, q_3, q_4, q_5, q_7, q_8, q_9\}$

$T_1 = e\text{-closure}(\delta(T_0, 0)) = \{q_2, q_{14}\}$

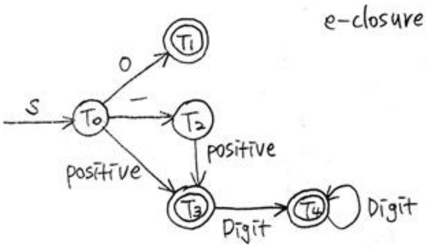
$T_2 = e\text{-closure}(\delta(T_0, -)) = \{q_6, q_8, q_9\}$

$T_3 = e\text{-closure}(\delta(T_0, \text{positive})) = \{q_{10}, q_{11}, q_{13}, q_{14}\}$

$e\text{-closure}(\delta(T_2, \text{positive})) = \{q_{10}, q_{11}, q_{13}, q_{14}\} = T_3$

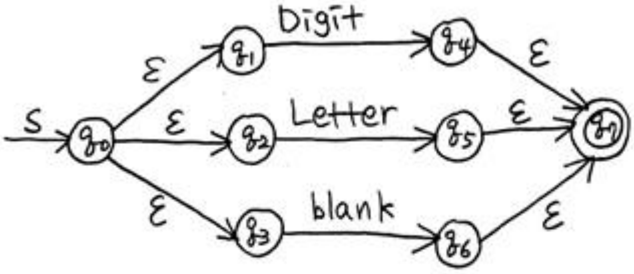
$T_4 = e\text{-closure}(\delta(T_3, \text{Digit})) = \{q_{11}, q_{12}, q_{13}, q_{14}\}$

$e\text{-closure}(\delta(T_4, \text{Digit})) = \{q_{11}, q_{12}, q_{13}, q_{14}\} = T_4$



3) Single character

char (NFA)



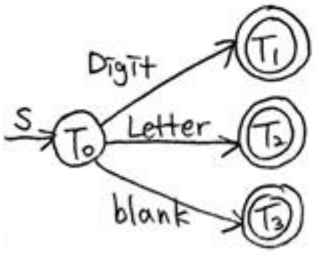
char (DFA)

$T_0 = \text{e-closure}(q_0) = \{q_0, q_1, q_2, q_3\}$

$T_1 = \text{e-closure}(\delta(T_0, \text{Digit})) = \{q_4, q_7\}$

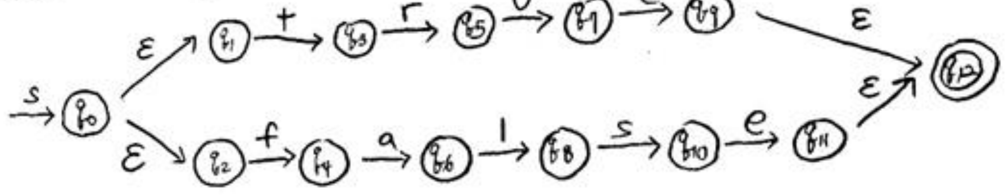
$T_2 = \text{e-closure}(\delta(T_0, \text{Letter})) = \{q_5, q_7\}$

$T_3 = \text{e-closure}(\delta(T_0, \text{blank})) = \{q_6, q_7\}$



4) Boolean string

Bool (NFA)



Bool (DFA)

$T_0 = \text{e-closure}(q_0) = \{q_1, q_2\}$

$T_1 = \text{e-closure}(\delta(T_0, t)) = \{q_3\}$

$T_2 = \text{e-closure}(\delta(T_0, f)) = \{q_4\}$

$T_3 = \text{e-closure}(\delta(T_1, r)) = \{q_5\}$

$T_4 = \text{e-closure}(\delta(T_2, a)) = \{q_6\}$

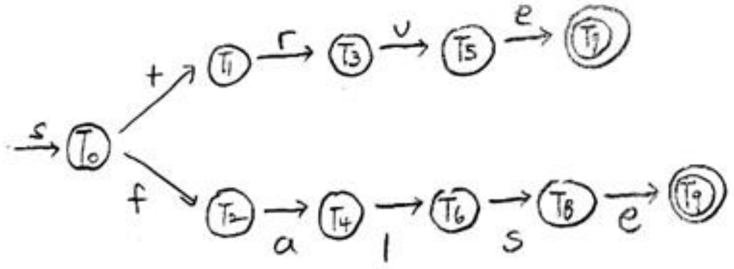
$T_5 = \text{e-closure}(\delta(T_3, u)) = \{q_7\}$

$T_6 = \text{e-closure}(\delta(T_4, l)) = \{q_8\}$

$T_7 = \text{e-closure}(\delta(T_5, e)) = \{q_9, q_{12}\}$

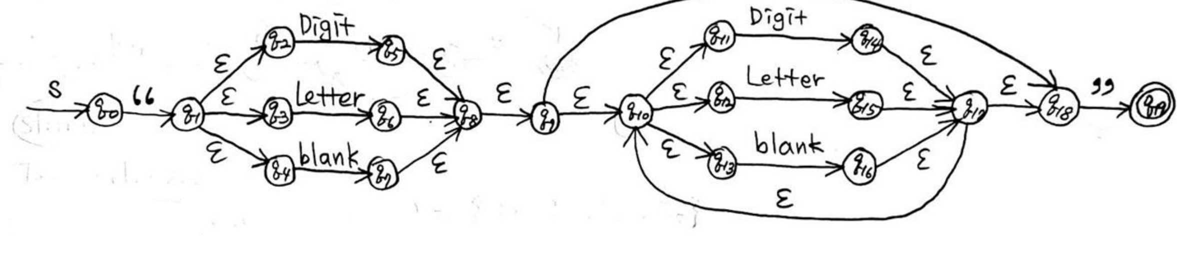
$T_8 = \text{e-closure}(\delta(T_6, s)) = \{q_{10}\}$

$T_9 = \text{e-closure}(\delta(T_8, e)) = \{q_{11}, q_{12}\}$



5) Literal String

String (NFA)



String (DFA)

$T_0 = \text{e-closure}(q_0) = \{q_0\}$

$T_1 = \text{e-closure}(\delta(T_0, “)) = \{q_1, q_2, q_3, q_4\}$

$T_2 = \text{e-closure}(\delta(T_1, \text{Digit})) = \{q_5, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{18}\}$

$T_3 = \text{e-closure}(\delta(T_1, \text{Letter})) = \{q_6, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{18}\}$

$T_4 = \text{e-closure}(\delta(T_1, \text{blank})) = \{q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{18}\}$

$T_5 = \text{e-closure}(\delta(T_2, \text{Digit})) = \text{e-closure}(\delta(T_3, \text{Digit})) = \text{e-closure}(\delta(T_4, \text{Digit}))$

$= \{q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{17}, q_{18}\}$

$T_6 = \text{e-closure}(\delta(T_2, \text{Letter})) = \text{e-closure}(\delta(T_3, \text{Letter})) = \text{e-closure}(\delta(T_4, \text{Letter}))$

$= \{q_{10}, q_{11}, q_{12}, q_{13}, q_{15}, q_{17}, q_{18}\}$

$T_7 = \text{e-closure}(\delta(T_2, \text{blank})) = \text{e-closure}(\delta(T_3, \text{blank})) = \text{e-closure}(\delta(T_4, \text{blank}))$

$= \{q_{10}, q_{11}, q_{12}, q_{13}, q_{16}, q_{17}, q_{18}\}$

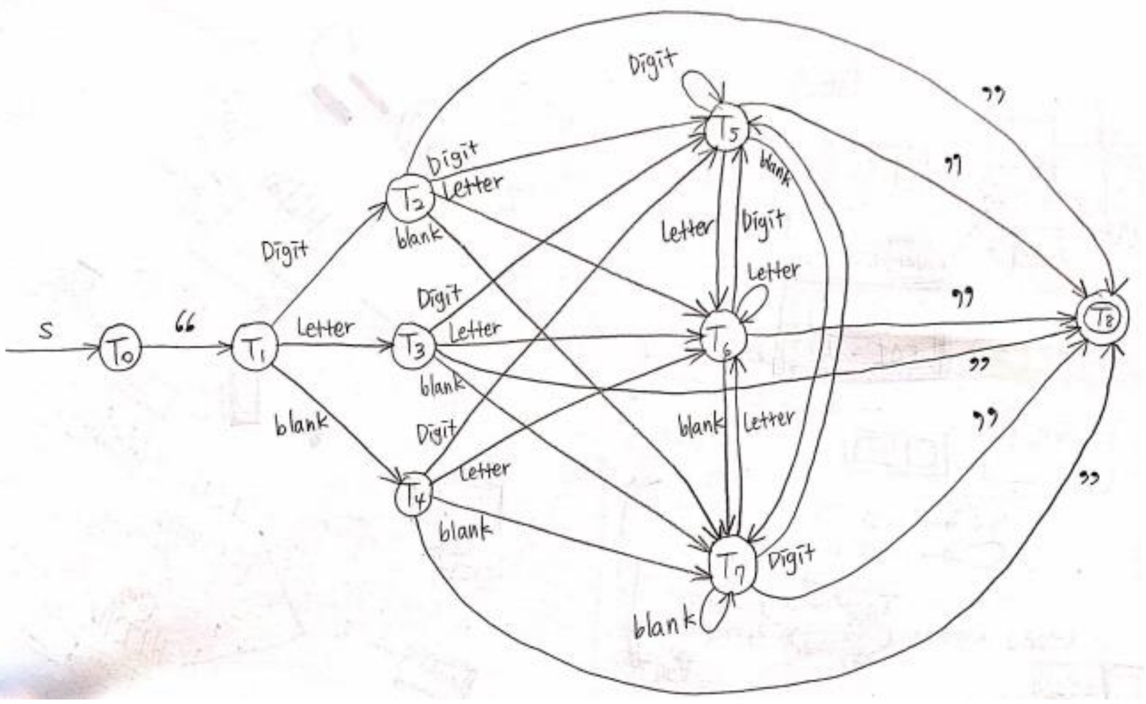
$T_8 = \text{e-closure}(\delta(T_2, ”)) = \text{e-closure}(\delta(T_3, ”)) = \text{e-closure}(\delta(T_4, ”)) = \{q_{19}\}$

$\text{e-closure}(\delta(T_5, \text{Digit})) = \text{e-closure}(\delta(T_6, \text{Digit})) = \text{e-closure}(\delta(T_7, \text{Digit})) = T_5$

$\text{e-closure}(\delta(T_5, \text{Letter})) = \text{e-closure}(\delta(T_6, \text{Letter})) = \text{e-closure}(\delta(T_7, \text{Letter})) = T_6$

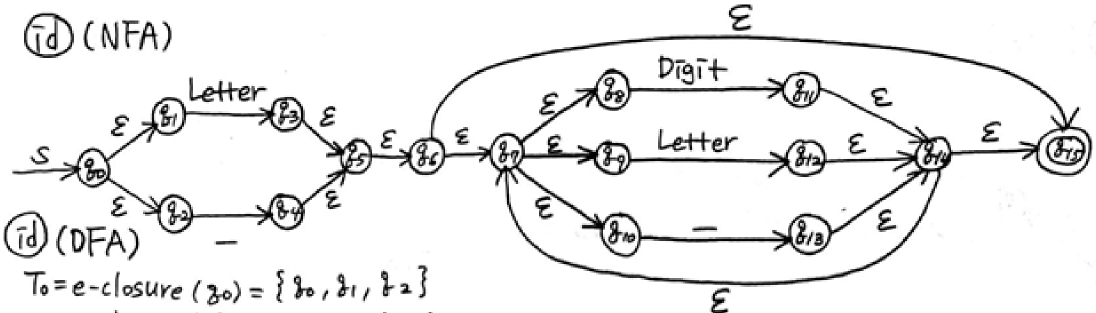
$\text{e-closure}(\delta(T_5, \text{blank})) = \text{e-closure}(\delta(T_6, \text{blank})) = \text{e-closure}(\delta(T_7, \text{blank})) = T_7$

$\text{e-closure}(\delta(T_5, ”)) = \text{e-closure}(\delta(T_6, ”)) = \text{e-closure}(\delta(T_7, ”)) = \{q_{19}\} = T_8$



6) An identifier of variables and functions

id (NFA)



id (DFA)

$T_0 = \text{e-closure}(q_0) = \{q_0, q_1, q_2\}$

$T_1 = \text{e-closure}(\delta(T_0, \text{Letter})) = \{q_3, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{15}\}$

$T_2 = \text{e-closure}(\delta(T_0, \_)) = \{q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{15}\}$

$T_3 = \text{e-closure}(\delta(T_1, \text{Digit})) = \{q_7, q_8, q_9, q_{10}, q_{11}, q_{14}, q_{15}\}$

$T_4 = \text{e-closure}(\delta(T_1, \text{Letter})) = \{q_7, q_8, q_9, q_{10}, q_{12}, q_{14}, q_{15}\}$

$T_5 = \text{e-closure}(\delta(T_1, \_)) = \{q_7, q_8, q_9, q_{10}, q_{13}, q_{14}, q_{15}\}$

$\text{e-closure}(\delta(T_2, \text{Digit})) = T_3$

$\text{e-closure}(\delta(T_2, \text{Letter})) = T_4$

$\text{e-closure}(\delta(T_2, \_)) = T_5$

$\text{e-closure}(\delta(T_3, \text{Digit})) = T_3$

$\text{e-closure}(\delta(T_3, \text{Letter})) = T_4$

$\text{e-closure}(\delta(T_3, \_)) = T_5$

$\text{e-closure}(\delta(T_4, \text{Digit})) = T_3$

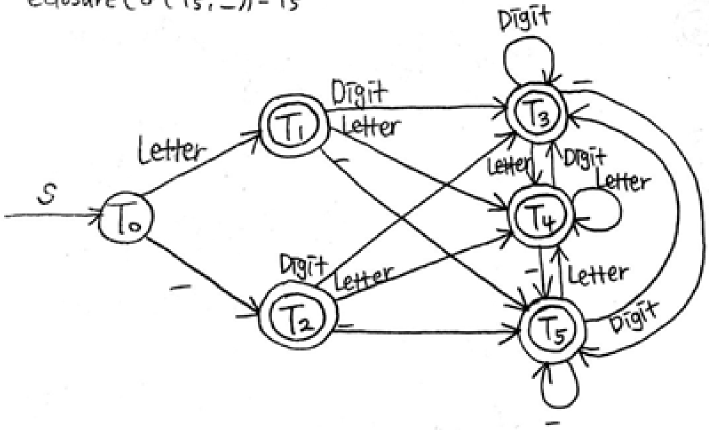
$\text{e-closure}(\delta(T_4, \text{Letter})) = T_4$

$\text{e-closure}(\delta(T_4, \_)) = T_5$

$\text{e-closure}(\delta(T_5, \text{Digit})) = T_3$

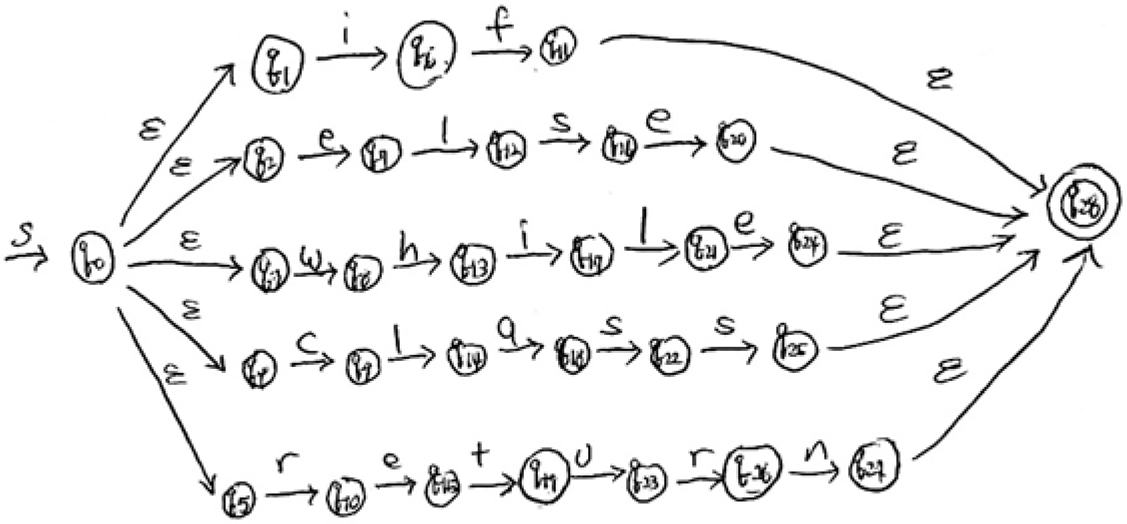
$\text{e-closure}(\delta(T_5, \text{Letter})) = T_4$

$\text{e-closure}(\delta(T_5, \_)) = T_5$



7) Keywords for special statements

keyword (NFA)



Keyword (DFA)

$T_0 = \text{e-closure}(q_0) = \{q_0, q_1, q_2, q_3, q_4, q_5\}$

$T_1 = \text{e-closure}(\delta(T_0, i)) = \{q_6\}$

$T_2 = \text{e-closure}(\delta(T_0, e)) = \{q_7\}$

$T_3 = \text{e-closure}(\delta(T_0, w)) = \{q_8\}$

$T_4 = \text{e-closure}(\delta(T_0, c)) = \{q_9\}$

$T_5 = \text{e-closure}(\delta(T_0, r)) = \{q_{10}\}$

$T_6 = \text{e-closure}(\delta(T_1, f)) = \{q_{11}, q_{28}\}$

$T_7 = \text{e-closure}(\delta(T_2, l)) = \{q_{12}\}$

$T_8 = \text{e-closure}(\delta(T_3, h)) = \{q_{13}\}$

$T_9 = \text{e-closure}(\delta(T_4, l)) = \{q_{14}\}$

$T_{10} = \text{e-closure}(\delta(T_5, e)) = \{q_{15}\}$

$T_{11} = \text{e-closure}(\delta(T_7, s)) = \{q_{16}\}$

$T_{12} = \text{e-closure}(\delta(T_8, i)) = \{q_{17}\}$

$T_{13} = \text{e-closure}(\delta(T_9, a)) = \{q_{18}\}$

$T_{14} = \text{e-closure}(\delta(T_{10}, t)) = \{q_{19}\}$

$T_{15} = \text{e-closure}(\delta(T_{11}, e)) = \{q_{20}, q_{28}\}$

$T_{16} = \text{e-closure}(\delta(T_{12}, l)) = \{q_{21}\}$

$T_{17} = \text{e-closure}(\delta(T_{13}, s)) = \{q_{22}\}$

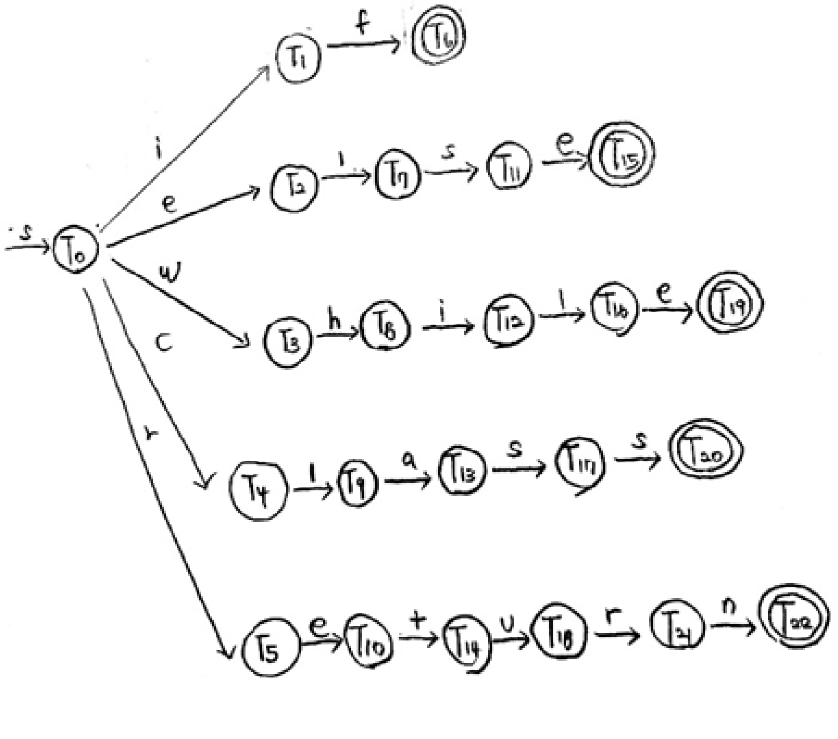
$T_{18} = \text{e-closure}(\delta(T_{14}, u)) = \{q_{23}\}$

$T_{19} = \text{e-closure}(\delta(T_{16}, e)) = \{q_{24}, q_{28}\}$

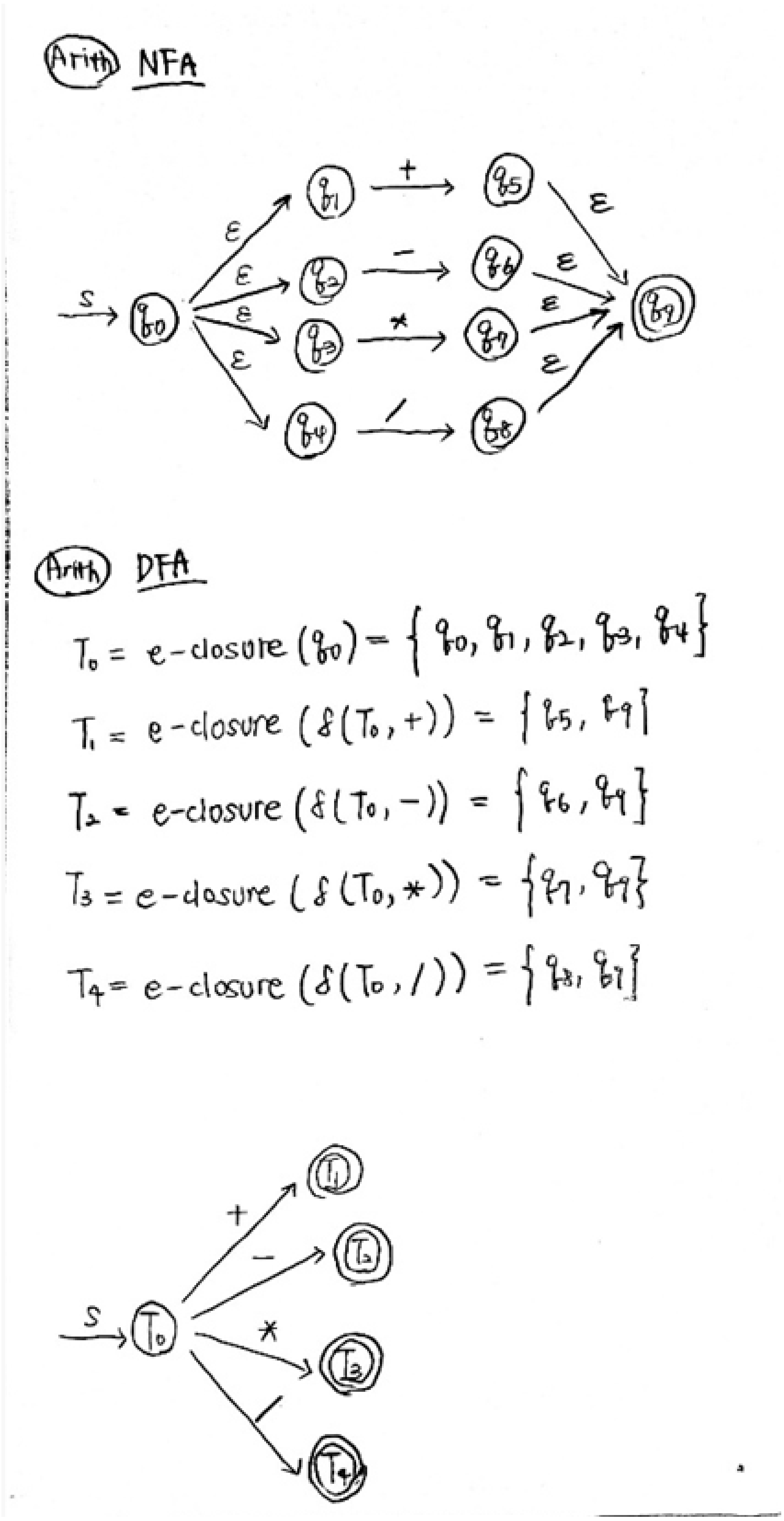
$T_{20} = \text{e-closure}(\delta(T_{17}, s)) = \{q_{25}, q_{28}\}$

$T_{21} = \text{e-closure}(\delta(T_{18}, r)) = \{q_{26}\}$

$T_{22} = \text{e-closure}(\delta(T_{21}, n)) = \{q_{27}, q_{28}\}$



8) Arithmetic operators



Arith NFA

Arith DFA

$T_0 = \text{e-closure}(q_0) = \{q_0, q_1, q_2, q_3, q_4\}$

$T_1 = \text{e-closure}(\delta(T_0, +)) = \{q_5, q_9\}$

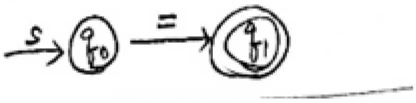
$T_2 = \text{e-closure}(\delta(T_0, -)) = \{q_6, q_9\}$

$T_3 = \text{e-closure}(\delta(T_0, *)) = \{q_7, q_9\}$

$T_4 = \text{e-closure}(\delta(T_0, /)) = \{q_8, q_9\}$

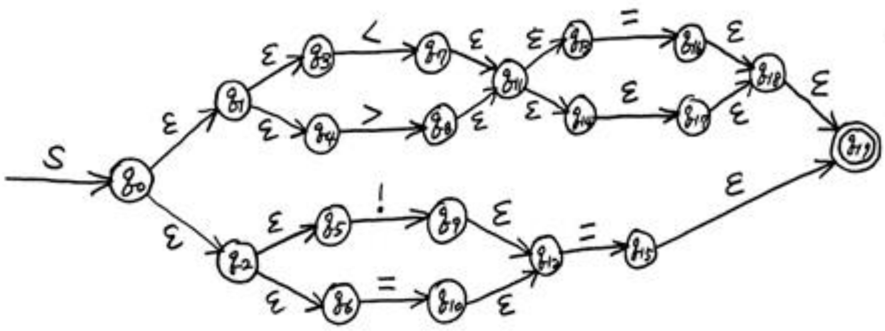
9) Assignment operator (NFA and DFA is equal)

Assign NFA , DFA



10) Comparison operators

Comp (NFA)



Comp (DFA)

T_0 = e-closure (q_0) = { q_0, q_1 , q_2, q_3 , q_4 , q_5, q_6 }

T_1 = e-closure ($\delta(T_0, <)$) = { q_7, q_{11} , q_{13} , q_{14}, q_{17} , q_{18} , q_{19} }

T_2 = e-closure ($\delta(T_0, >)$) = { q_8, q_{11} , q_{13} , q_{14}, q_{17}, q_{18} , q_{19} }

T_3 = e-closure ($\delta(T_0, !)$) = { q_9 , q_{12} }

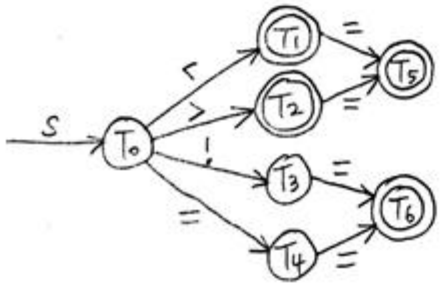
T_4 = e-closure($\delta(T_0, =)$) = { q_{10} , q_{12} }

T_5 = e-closure ($\delta(T_1, =)$) = { q_{16}, q_{18}, q_{19} }

e-closure ($\delta(T_2, =)$) = { q_{16}, q_{18}, q_{19} } = T_5

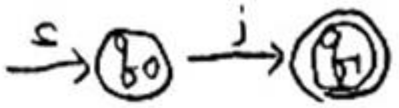
T_6 = e-closure($\delta(T_3, =)$) = { q_{15}, q_{19} }

e-closure($\delta(T_4, =)$) = { q_{15}, q_{19} } = T_6



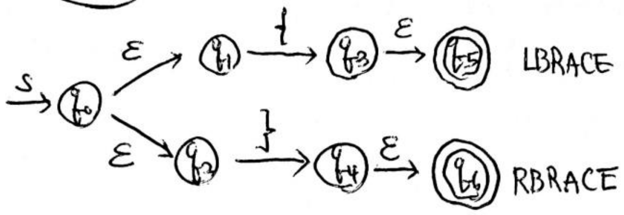
11) A terminating symbol of statements : ; (NFA and DFA is equal)

Semi NFA , DFA



12) A pair of symbols for defining area/scope of variables and functions : { and }

Braces NFA

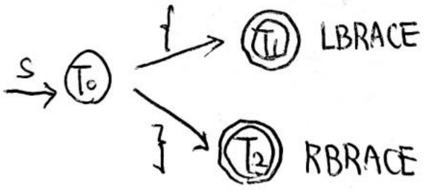


Braces DFA

T_0 = e-closure (q_0) = $\{q_1, q_2\}$

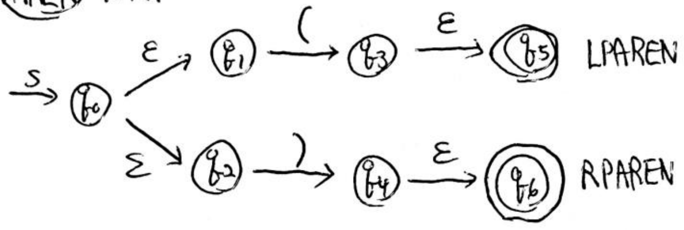
T_1 = e-closure $(\delta(T_0, \{\ ))$ = $\{q_3, q_5\}$

T_2 = e-closure $(\delta(T_0, \}\ ))$ = $\{q_4, q_6\}$



13) A pair of symbols for indicating a function / statement : (and)

PAREN NFA

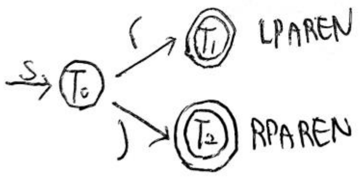


PAREN DFA

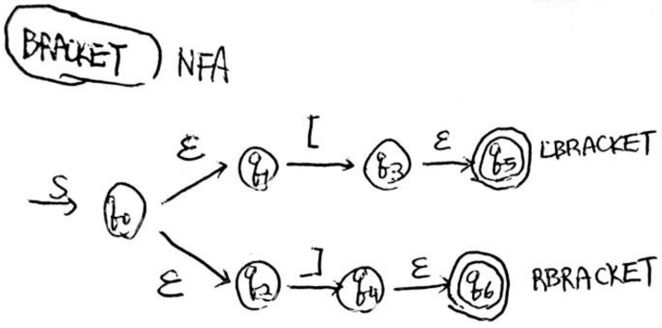
T_0 = e-closure $(q_0) = \{q_1, q_2\}$

T_1 = e-closure $(\delta(T_0, ()) = \{q_3, q_5\}$

T_2 = e-closure $(\delta(T_0,))) = \{q_4, q_6\}$



14) A pair of symbols for using an array : [and]

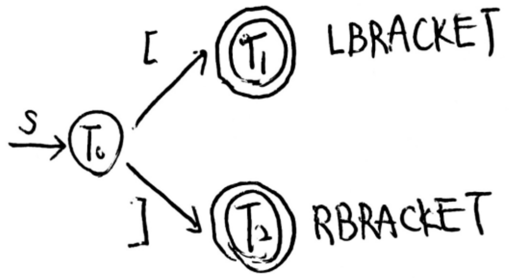


BRACKET DFA

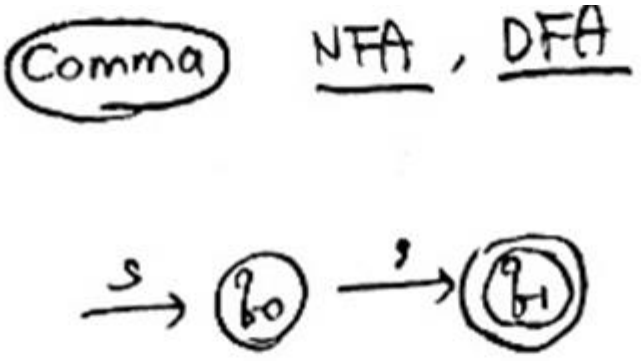
$T_0 = \text{e-closure}(q_0) = \{q_1, q_2\}$

$T_1 = \text{e-closure}(\delta(T_0, [)) = \{q_3, q_5\}$

$T_2 = \text{e-closure}(\delta(T_0,])) = \{q_4, q_6\}$

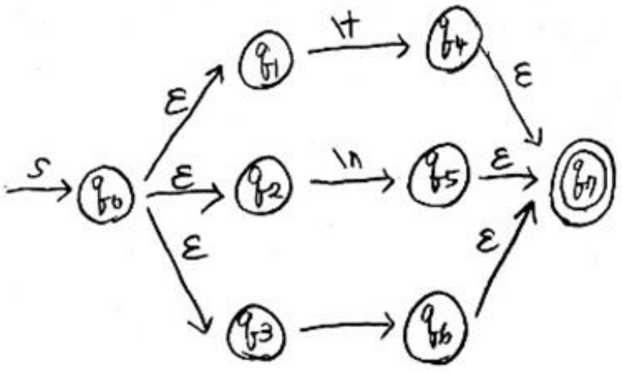


15) A symbol for separating input arguments in functions : , (NFA and DFA is equal)



16) Whitespaces

Space NFA



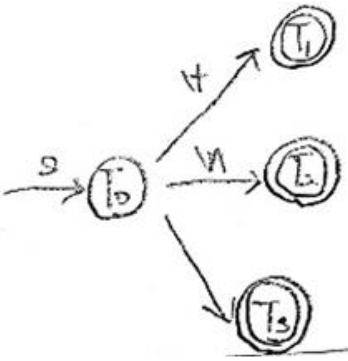
Space DFA

$T_0 = \text{e-closure}(q_0) = \{q_0, q_1, q_2, q_3\}$

$T_1 = \text{e-closure}(\delta(T_0, \backslash t)) = \{q_4, q_7\}$

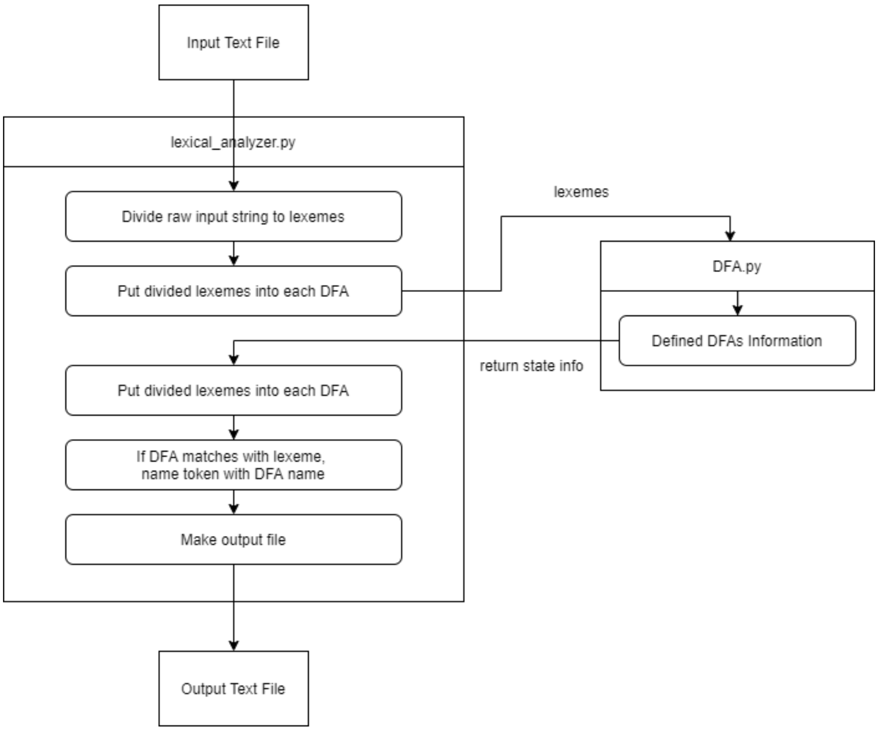
$T_2 = \text{e-closure}(\delta(T_1, \backslash n)) = \{q_5, q_7\}$

$T_3 = \text{e-closure}(\delta(T_2, \ )) = \{q_6, q_7\}$



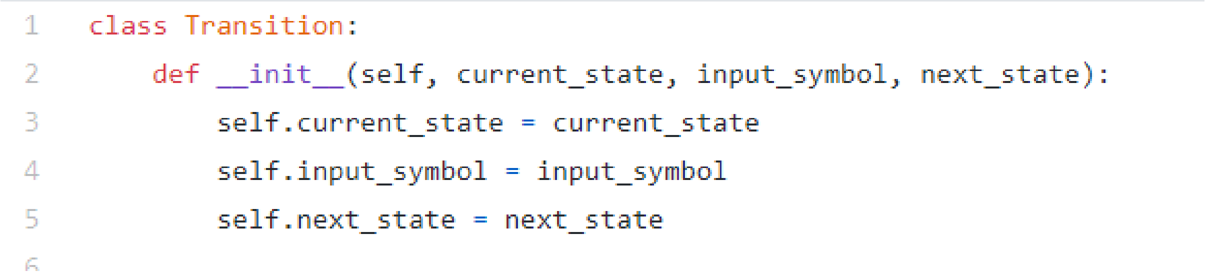
C. All About Our Lexical Analyzer

1) Overall Flow



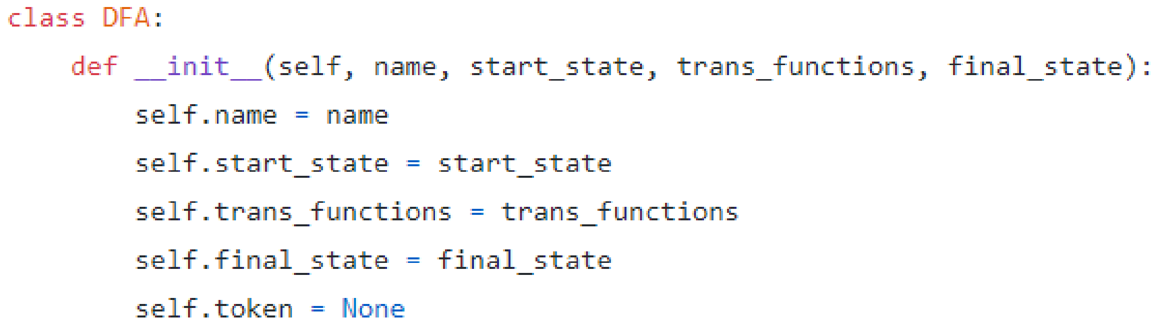
2) About Each File

1. DFA.py



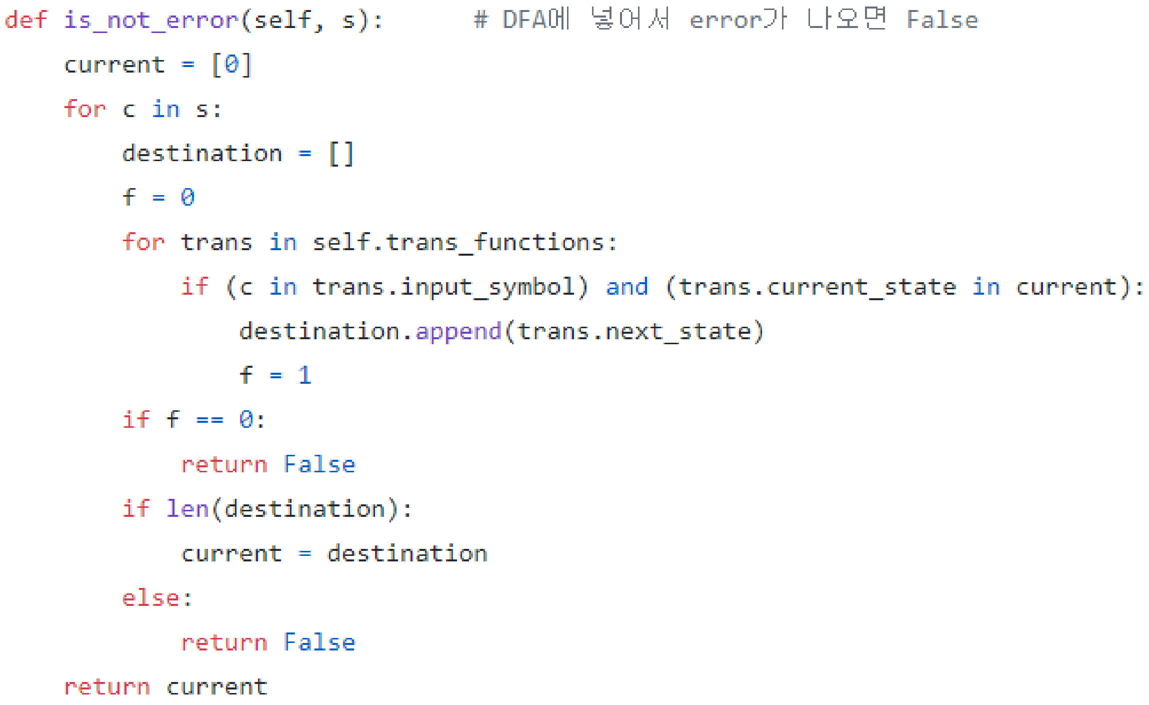
```python
class Transition:
    def __init__(self, current_state, input_symbol, next_state):
        self.current_state = current_state
        self.input_symbol = input_symbol
        self.next_state = next_state

```

Define transition function class. If input_symbol is entered current_state moves to next_state.



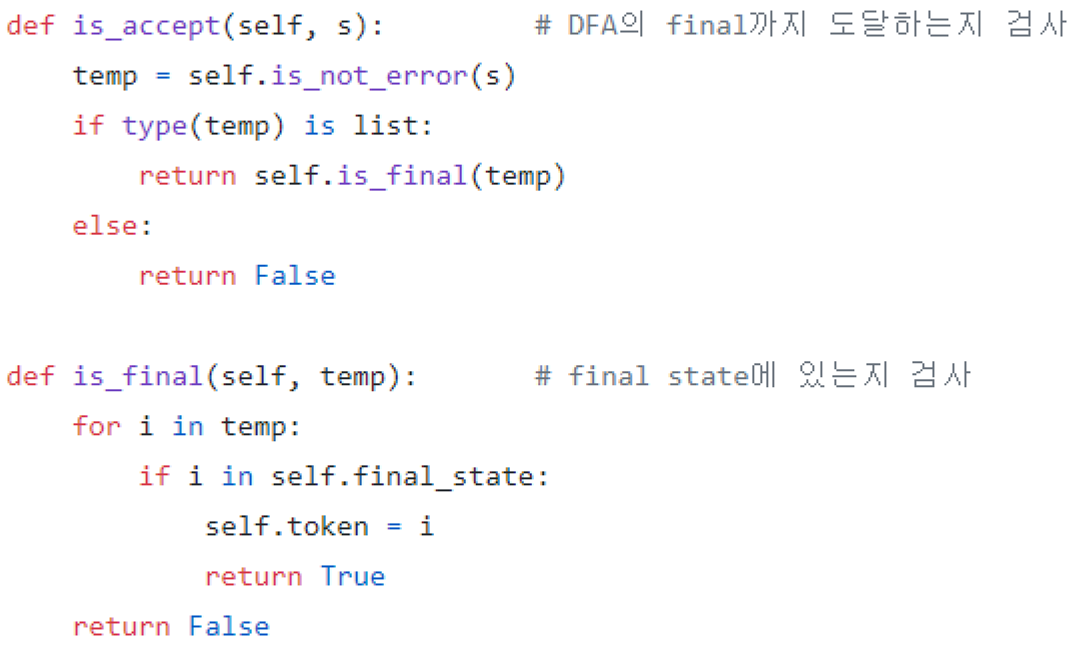
```python
class DFA:
    def __init__(self, name, start_state, trans_functions, final_state):
        self.name = name
        self.start_state = start_state
        self.trans_functions = trans_functions
        self.final_state = final_state
        self.token = None
```

DFA class initialization. It consists name, start_state, trans_functions(transition functions of each DFA), final_state. Token is initialized as “None” at first.



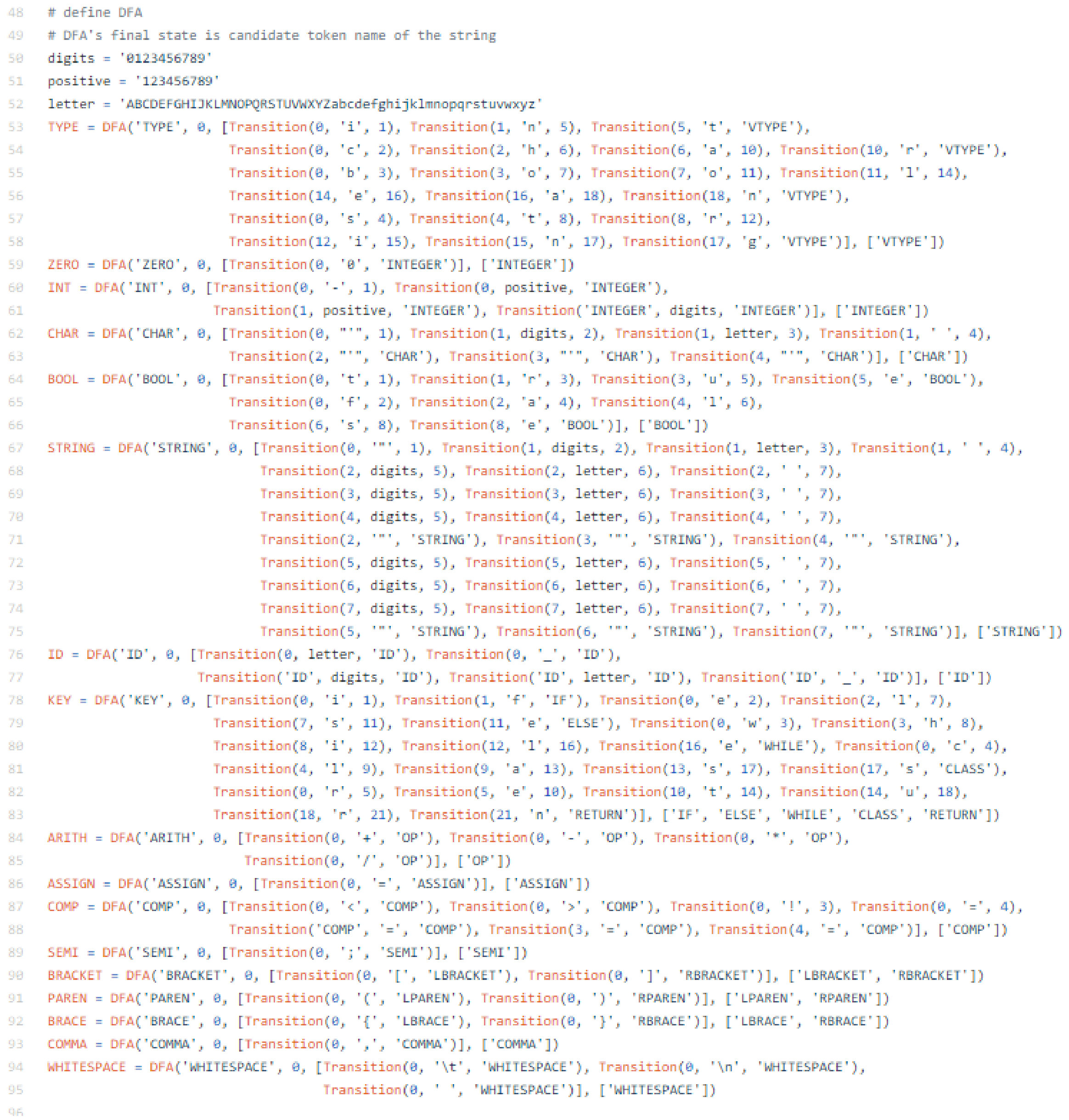
```python
def is_not_error(self, s):      # DFA에 넣어서 error가 나오면 False
    current = [0]
    for c in s:
        destination = []
        f = 0
        for trans in self.trans_functions:
            if (c in trans.input_symbol) and (trans.current_state in current):
                destination.append(trans.next_state)
                f = 1
        if f == 0:
            return False
        if len(destination):
            current = destination
        else:
            return False
    return current
```

Function to check whether current input string ‘s’ can be accepted at current DFA or not. Input string s is a part of “input.txt” string data. Variable ‘f’ is to check whether current input is accepted to current DFA. If f is 0, it returns false. If there is any input at destination list, it returns current which is a list of destination.



```
def is_accept(self, s):         # DFA의 final까지 도달하는지 검사
    temp = self.is_not_error(s)
    if type(temp) is list:
        return self.is_final(temp)
    else:
        return False

def is_final(self, temp):       # final state에 있는지 검사
    for i in temp:
        if i in self.final_state:
            self.token = i
            return True
    return False
```

Function to check if input ‘s’ reaches to current DFA’s final state. If temp is list it goes to is_final function and check if temp list has final_state of current DFA.

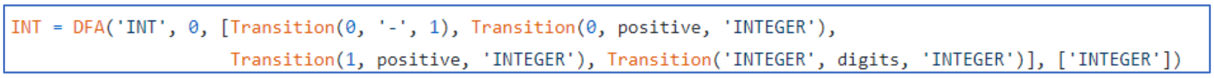


```python
# define DFA
# DFA's final state is candidate token name of the string
digits = '0123456789'
positive = '123456789'
letter = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
TYPE = DFA('TYPE', 0, [Transition(0, 'i', 1), Transition(1, 'n', 5), Transition(5, 't', 'VTYPE'),
                       Transition(0, 'c', 2), Transition(2, 'h', 6), Transition(6, 'a', 10), Transition(10, 'r', 'VTYPE'),
                       Transition(0, 'b', 3), Transition(3, 'o', 7), Transition(7, 'o', 11), Transition(11, 'l', 14),
                       Transition(14, 'e', 16), Transition(16, 'a', 18), Transition(18, 'n', 'VTYPE'),
                       Transition(0, 's', 4), Transition(4, 't', 8), Transition(8, 'r', 12),
                       Transition(12, 'i', 15), Transition(15, 'n', 17), Transition(17, 'g', 'VTYPE')], ['VTYPE'])
ZERO = DFA('ZERO', 0, [Transition(0, '0', 'INTEGER')], ['INTEGER'])
INT = DFA('INT', 0, [Transition(0, '-', 1), Transition(0, positive, 'INTEGER'),
                     Transition(1, positive, 'INTEGER'), Transition('INTEGER', digits, 'INTEGER')], ['INTEGER'])
CHAR = DFA('CHAR', 0, [Transition(0, "'", 1), Transition(1, digits, 2), Transition(1, letter, 3), Transition(1, ' ', 4),
                       Transition(2, "'", 'CHAR'), Transition(3, "'", 'CHAR'), Transition(4, "'", 'CHAR')], ['CHAR'])
BOOL = DFA('BOOL', 0, [Transition(0, 't', 1), Transition(1, 'r', 3), Transition(3, 'u', 5), Transition(5, 'e', 'BOOL'),
                       Transition(0, 'f', 2), Transition(2, 'a', 4), Transition(4, 'l', 6),
                       Transition(6, 's', 8), Transition(8, 'e', 'BOOL')], ['BOOL'])
STRING = DFA('STRING', 0, [Transition(0, '"', 1), Transition(1, digits, 2), Transition(1, letter, 3), Transition(1, ' ', 4),
                           Transition(2, digits, 5), Transition(2, letter, 6), Transition(2, ' ', 7),
                           Transition(3, digits, 5), Transition(3, letter, 6), Transition(3, ' ', 7),
                           Transition(4, digits, 5), Transition(4, letter, 6), Transition(4, ' ', 7),
                           Transition(2, '"', 'STRING'), Transition(3, '"', 'STRING'), Transition(4, '"', 'STRING'),
                           Transition(5, digits, 5), Transition(5, letter, 6), Transition(5, ' ', 7),
                           Transition(6, digits, 5), Transition(6, letter, 6), Transition(6, ' ', 7),
                           Transition(7, digits, 5), Transition(7, letter, 6), Transition(7, ' ', 7),
                           Transition(5, '"', 'STRING'), Transition(6, '"', 'STRING'), Transition(7, '"', 'STRING')], ['STRING'])
ID = DFA('ID', 0, [Transition(0, letter, 'ID'), Transition(0, '_', 'ID'),
                   Transition('ID', digits, 'ID'), Transition('ID', letter, 'ID'), Transition('ID', '_', 'ID')], ['ID'])
KEY = DFA('KEY', 0, [Transition(0, 'i', 1), Transition(1, 'f', 'IF'), Transition(0, 'e', 2), Transition(2, 'l', 7),
                     Transition(7, 's', 11), Transition(11, 'e', 'ELSE'), Transition(0, 'w', 3), Transition(3, 'h', 8),
                     Transition(8, 'i', 12), Transition(12, 'l', 16), Transition(16, 'e', 'WHILE'), Transition(0, 'c', 4),
                     Transition(4, 'l', 9), Transition(9, 'a', 13), Transition(13, 's', 17), Transition(17, 's', 'CLASS'),
                     Transition(0, 'r', 5), Transition(5, 'e', 10), Transition(10, 't', 14), Transition(14, 'u', 18),
                     Transition(18, 'r', 21), Transition(21, 'n', 'RETURN')], ['IF', 'ELSE', 'WHILE', 'CLASS', 'RETURN'])
ARITH = DFA('ARITH', 0, [Transition(0, '+', 'OP'), Transition(0, '-', 'OP'), Transition(0, '*', 'OP'),
                         Transition(0, '/', 'OP')], ['OP'])
ASSIGN = DFA('ASSIGN', 0, [Transition(0, '=', 'ASSIGN')], ['ASSIGN'])
COMP = DFA('COMP', 0, [Transition(0, '<', 'COMP'), Transition(0, '>', 'COMP'), Transition(0, '!', 3), Transition(0, '=', 4),
                       Transition('COMP', '=', 'COMP'), Transition(3, '=', 'COMP'), Transition(4, '=', 'COMP')], ['COMP'])
SEMI = DFA('SEMI', 0, [Transition(0, ';', 'SEMI')], ['SEMI'])
BRACKET = DFA('BRACKET', 0, [Transition(0, '[', 'LBRACKET'), Transition(0, ']', 'RBRACKET')], ['LBRACKET', 'RBRACKET'])
PAREN = DFA('PAREN', 0, [Transition(0, '(', 'LPAREN'), Transition(0, ')', 'RPAREN')], ['LPAREN', 'RPAREN'])
BRACE = DFA('BRACE', 0, [Transition(0, '{', 'LBRACE'), Transition(0, '}', 'RBRACE')], ['LBRACE', 'RBRACE'])
COMMA = DFA('COMMA', 0, [Transition(0, ',', 'COMMA')], ['COMMA'])
WHITESPACE = DFA('WHITESPACE', 0, [Transition(0, '\t', 'WHITESPACE'), Transition(0, '\n', 'WHITESPACE'),
                                   Transition(0, ' ', 'WHITESPACE')], ['WHITESPACE'])
```

Define each DFA.

DFA	Available Inputs	Final States (Token Name)
TYPE	i n t c h a r b o l e s g	VTYPE
ZERO	0	INTEGER
INT	0 - positive digits	INTEGER
CHAR	‘ ’ letter digits blank	CHAR
BOOL	t r u e f a l s	BOOL
STRING	“ ” letter digits blank	STRING
ID	letter _ digits	ID
KEY	i f e l s w h i c a r t u n	IF, ELSE, WHILE, CLASS, RETURN
ARITH	+ - * /	OP
ASSIGN	=	ASSIGN
COMP	< > = !	COMP
SEMI	;	SEMI
BRACKET	[ ]	LBRACKET, RBRACKET
PAREN	( )	LPAREN, RPAREN
BRACE	{ }	LBRACE, RBRACE
COMMA	,	COMMA
WHITESPACE	₩t, ₩n, blank	WHITESPACE

This table is information of our DFAs. (We added ‘ZERO’ DFA to deal with 0 integers.)



```
INT = DFA('INT', 0, [Transition(0, '-', 1), Transition(0, positive, 'INTEGER'),
                     Transition(1, positive, 'INTEGER'), Transition('INTEGER', digits, 'INTEGER')], ['INTEGER'])
```

Let's see this(↑) example.

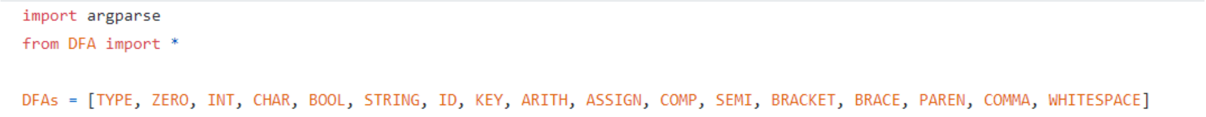
'INT' : name of DFA, which will be used as token name

0 : start state

[Transition (current_state, input_symbol, next_state)] : List of transition functions

['INT'] : list of DFA's final states

2. lexical_analyzer.py

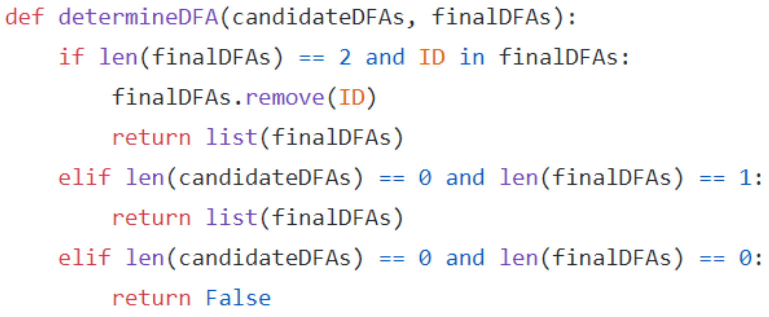


```
import argparse
from DFA import *

DFAs = [TYPE, ZERO, INT, CHAR, BOOL, STRING, ID, KEY, ARITH, ASSIGN, COMP, SEMI, BRACKET, BRACE, PAREN, COMMA, WHITESPACE]
```

Import argparse : for command argument

from DFA import * : import DFA.py

DFAs = [TYPE, INT, ….. WHITESPACE] : list of DFAs



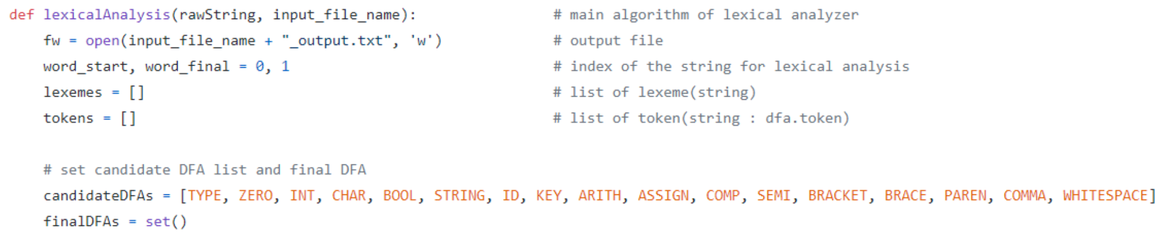
```
def determineDFA(candidateDFAs, finalDFAs):
    if len(finalDFAs) == 2 and ID in finalDFAs:
        finalDFAs.remove(ID)
        return list(finalDFAs)
    elif len(candidateDFAs) == 0 and len(finalDFAs) == 1:
        return list(finalDFAs)
    elif len(candidateDFAs) == 0 and len(finalDFAs) == 0:
        return False
```

We use two list to determine lexeme's DFA.

First Condition : If input such as "if", "true" are entered, "KEY" and "ID" DFA or "BOOL" and "ID" DFAs are remained at finalDFAs. However in that case, we need to determine them as "KEY", or "BOOL" instead of "ID". So we removed ID DFA from DFAs list and return DFAs list which only contains "KEY" or "BOOL".

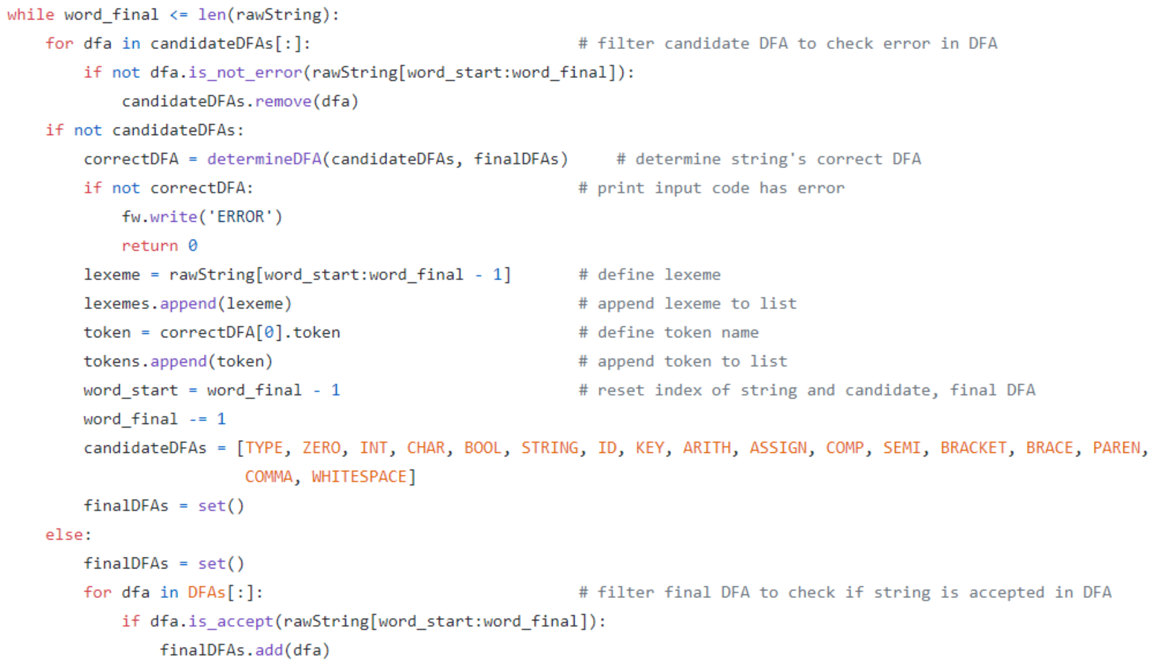
Second Condition : if candidateDFAs list is empty and finalDFAs has one composition, we use DFA in finalDFAs list.

Third Condition : if both candidateDFAs and finalDFAs are empty we don't it returns false.



```
def lexicalAnalysis(rawString, input_file_name):              # main algorithm of lexical analyzer
    fw = open(input_file_name + "_output.txt", 'w')          # output file
    word_start, word_final = 0, 1                             # index of the string for lexical analysis
    lexemes = []                                              # list of lexeme(string)
    tokens = []                                               # list of token(string : dfa.token)

    # set candidate DFA list and final DFA
    candidateDFAs = [TYPE, ZERO, INT, CHAR, BOOL, STRING, ID, KEY, ARITH, ASSIGN, COMP, SEMI, BRACKET, BRACE, PAREN, COMMA, WHITESPACE]
    finalDFAs = set()
```

Function for lexical analyzing. First create output file to write output information. “word_start” and “word_final” are index for string. We use “token” string variable, “lexemes” list to store lexemes and “tokens” list to store tokens.



```
while word_final <= len(rawString):
    for dfa in candidateDFAs[:]:                             # filter candidate DFA to check error in DFA
        if not dfa.is_not_error(rawString[word_start:word_final]):
            candidateDFAs.remove(dfa)
    if not candidateDFAs:
        correctDFA = determineDFA(candidateDFAs, finalDFAs)     # determine string's correct DFA
        if not correctDFA:                                   # print input code has error
            fw.write('ERROR')
            return 0
        lexeme = rawString[word_start:word_final - 1]        # define lexeme
        lexemes.append(lexeme)                               # append lexeme to list
        token = correctDFA[0].token                          # define token name
        tokens.append(token)                                 # append token to list
        word_start = word_final - 1                          # reset index of string and candidate, final DFA
        word_final -= 1
        candidateDFAs = [TYPE, ZERO, INT, CHAR, BOOL, STRING, ID, KEY, ARITH, ASSIGN, COMP, SEMI, BRACKET, BRACE, PAREN,
                         COMMA, WHITESPACE]
        finalDFAs = set()
    else:
        finalDFAs = set()
        for dfa in DFAs[:]:                                  # filter final DFA to check if string is accepted in DFA
            if dfa.is_accept(rawString[word_start:word_final]):
                finalDFAs.add(dfa)
```

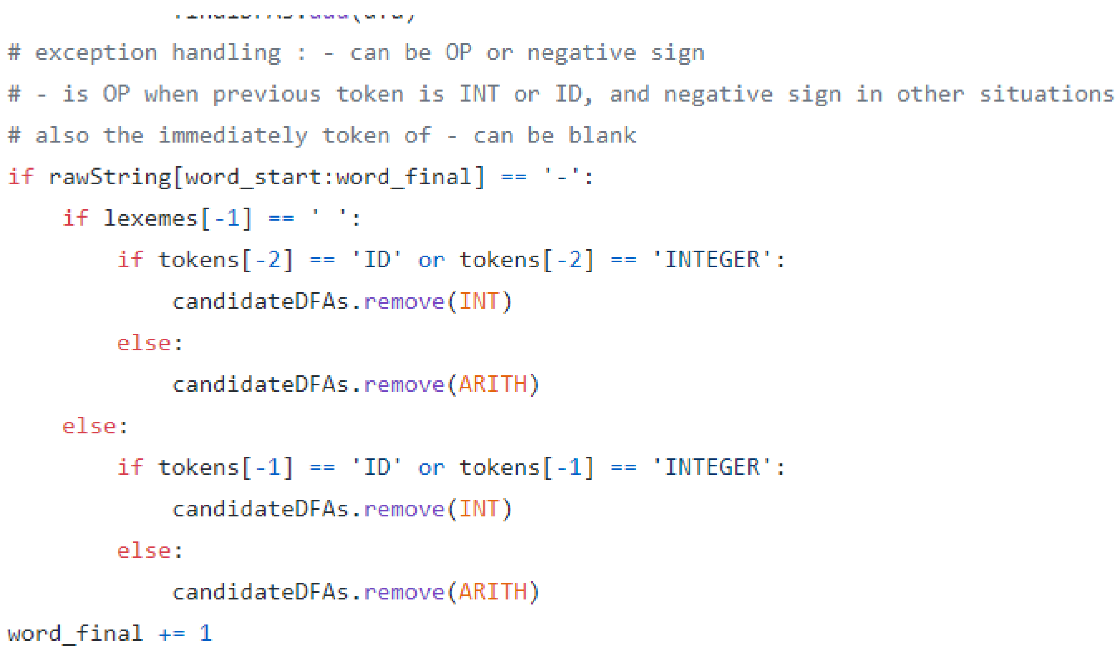
Until “word_final” index variable reaches the end of the string following codes are repeatedly executed. First we check DFAs in “candidateDFAs” and remove DFAs which don’t accept current input.

Then if “candidateDFAs” is not empty we check both “candidateDFAs” and “finalDFAs” to determine “correctDFA” through “determineDFA” function. “correctDFA” variable will be used as final DFA for current string data.

After “determineDFA”, if there aren’t any “correctDFA” it will return “ERROR”. However if “correctDFA” exists we add current string as “lexeme” and add it to “lexemes” list. Then we use “correctDFA”’s DFA name as token and add it to “tokens” list.

Then we initialize “word_start” and “word_final” index variable to the next string index. We also initialize “candidateDFAs” list and “finalDFAs” set.

However, if “candidateDFAs” is empty we use “finalDFAs” and check each DFA in “DFAs” list. If current string reaches to the final state of current DFA it will be added to “finalDFAs” list.



```
            finalDFAs.add(dfa)
# exception handling : - can be OP or negative sign
# - is OP when previous token is INT or ID, and negative sign in other situations
# also the immediately token of - can be blank
if rawString[word_start:word_final] == '-':
    if lexemes[-1] == ' ':
        if tokens[-2] == 'ID' or tokens[-2] == 'INTEGER':
            candidateDFAs.remove(INT)
        else:
            candidateDFAs.remove(ARITH)
    else:
        if tokens[-1] == 'ID' or tokens[-1] == 'INTEGER':
            candidateDFAs.remove(INT)
        else:
            candidateDFAs.remove(ARITH)
word_final += 1
```

Continuously, we used this code to deal with ‘-‘ symbol. To determine token of number with ‘-‘ symbol correctly, we need to see previous token. We divided into some cases to deal with this symbol.

1) 1 + -1 ➔ -1 (INT) , Previous token : OP

1-1) a + -1 ➔ -1 (INT) , Previous token : OP

3) a = -1 ➔ -1 (INT), Previous token : ASSIGN

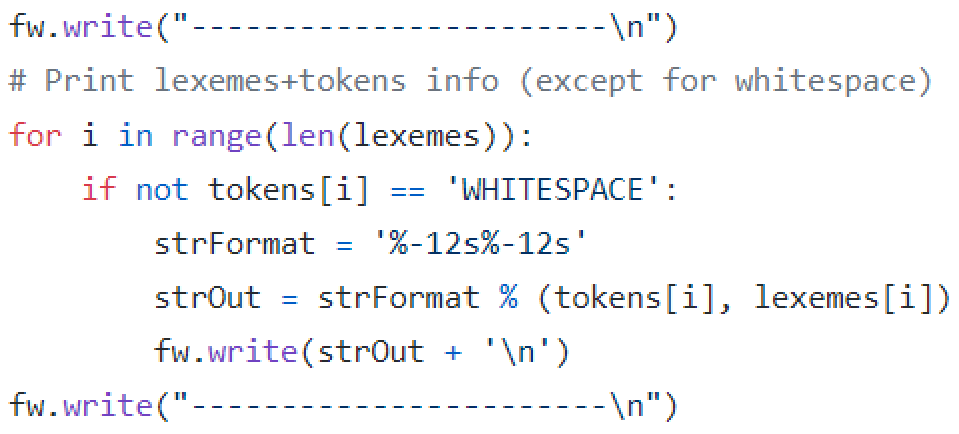
4) 1 – 1 ➔ - (OP), Previous token : INT

5) a – 1 ➔ - (OP), Previous token : ID

It means that if previous token is “INT” or “ID”, ‘-‘ symbol will be classified as “OP” token and if previous token is ***not*** “INT” or “ID”, ‘-‘ symbol will be classified as “INT” token. So we used if statements as image.

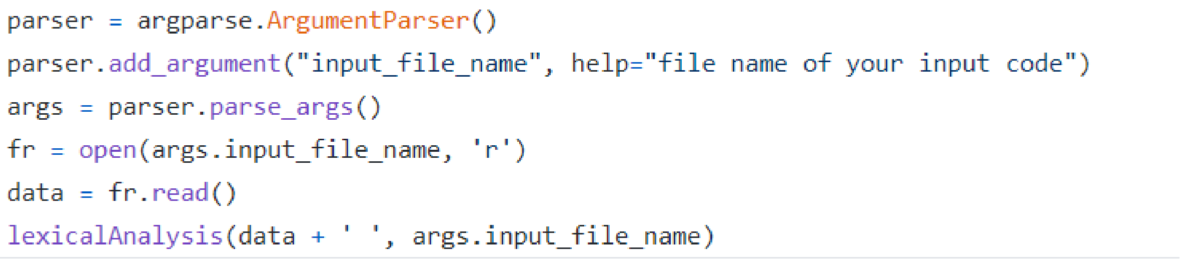
“if tokens[-1] == ‘ ‘” is to deal with blank space between minus symbol and previous token.

After that we increase “word_final” index to see the next character data of input string data.



```
fw.write("-----------------------\n")
# Print lexemes+tokens info (except for whitespace)
for i in range(len(lexemes)):
    if not tokens[i] == 'WHITESPACE':
        strFormat = '%-12s%-12s'
        strOut = strFormat % (tokens[i], lexemes[i])
        fw.write(strOut + '\n')
fw.write("-----------------------\n")
```

This part is for writing our lexeme and token result into output file.



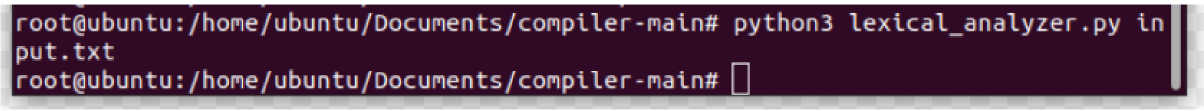
```
parser = argparse.ArgumentParser()
parser.add_argument("input_file_name", help="file name of your input code")
args = parser.parse_args()
fr = open(args.input_file_name, 'r')
data = fr.read()
lexicalAnalysis(data + ' ', args.input_file_name)
```

This part is for execution command and start point of the code.

We use " ***python3 lexical_analyzer.py input.txt*** " command to run this code.

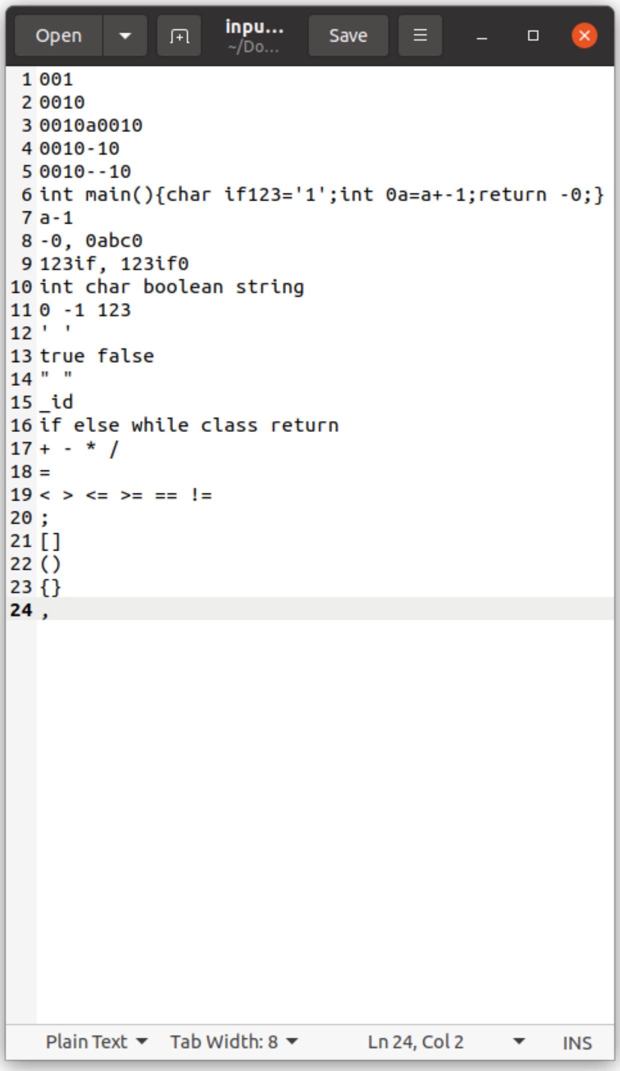
3) Result Screenshots

Command Line



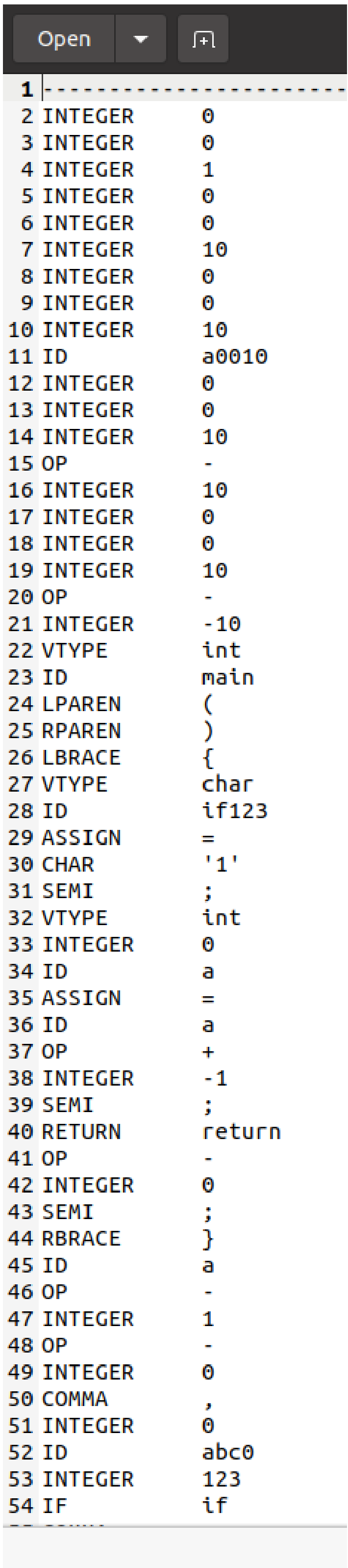
```
root@ubuntu:/home/ubuntu/Documents/compiler-main# python3 lexical_analyzer.py in
put.txt
root@ubuntu:/home/ubuntu/Documents/compiler-main# 
```

Input File (Input.txt)

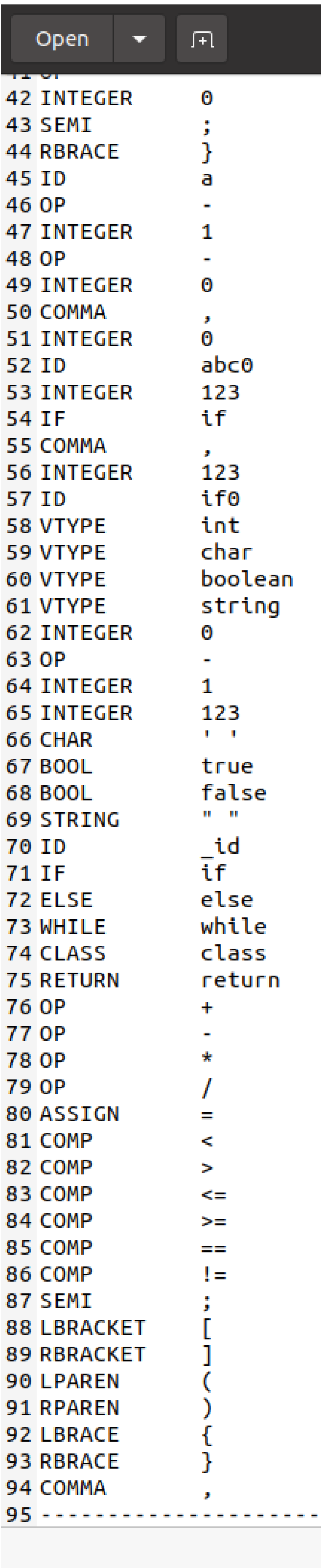


```
001
0010
0010a0010
0010-10
0010--10
int main(){char if123='1';int 0a=a+-1;return -0;}
a-1
-0, 0abc0
123if, 123if0
int char boolean string
0 -1 123
' '
true false
" "
_id
if else while class return
+ - * /
=
< > <= >= == !=
;
[]
()
{}
,
```

Output File (input.txt_output.txt)



```
----------------------
INTEGER     0
INTEGER     0
INTEGER     1
INTEGER     0
INTEGER     0
INTEGER     10
INTEGER     0
INTEGER     0
INTEGER     10
ID          a0010
INTEGER     0
INTEGER     0
INTEGER     10
OP          -
INTEGER     10
INTEGER     0
INTEGER     0
INTEGER     10
OP          -
INTEGER     -10
VTYPE       int
ID          main
LPAREN      (
RPAREN      )
LBRACE      {
VTYPE       char
ID          if123
ASSIGN      =
CHAR        '1'
SEMI        ;
VTYPE       int
INTEGER     0
ID          a
ASSIGN      =
ID          a
OP          +
INTEGER     -1
SEMI        ;
RETURN      return
OP          -
INTEGER     0
SEMI        ;
RBRACE      }
ID          a
OP          -
INTEGER     1
OP          -
INTEGER     0
COMMA       ,
INTEGER     0
ID          abc0
INTEGER     123
IF          if
```



```
INTEGER     0
SEMI        ;
RBRACE      }
ID          a
OP          -
INTEGER     1
OP          -
INTEGER     0
COMMA       ,
INTEGER     0
ID          abc0
INTEGER     123
IF          if
```

```
COMMA       ,
INTEGER     123
ID          if0
VTYPE       int
VTYPE       char
VTYPE       boolean
VTYPE       string
INTEGER     0
OP          -
INTEGER     1
INTEGER     123
CHAR        ' '
BOOL        true
BOOL        false
STRING      " "
ID          _id
IF          if
ELSE        else
WHILE       while
CLASS       class
RETURN      return
OP          +
OP          -
OP          *
OP          /
ASSIGN      =
COMP        <
COMP        >
COMP        <=
COMP        >=
COMP        ==
COMP        !=
SEMI        ;
LBRACKET    [
RBRACKET    ]
LPAREN      (
RPAREN      )
LBRACE      {
RBRACE      }
COMMA       ,
--------------------
```