**Contents**

I. Find CFG’s Ambiguity

II. Regular expression

III. NFA(Non-deterministic Finite Automata) & DFA(Deterministic Finite Automata)

1. Arithmetic NFA, transition graph, DFA
2. Comparison NFA, transition graph, DFA
3. Identifier NFA, transition graph, DFA
4. Literal String NFA, transition graph, DFA
5. Singed Integer NFA, transition graph, DFA
6. Single Character NFA, transition graph, DFA

IV. Implementation

1. Developing Environment
2. Define Tokens & Symbols
3. Implement DFA
4. Check Double Quote & single Quote
5. Input Stream Preprocessing
6. Tokenize
7. Save Result
8. Entire Process Run
9. ‘–‘ Symbol issue

V. Experiment

VI. Appendix

1. NFA & DFA with Transition Table by Hand Write

**I. Find CFG’s Ambiguity**

1. 주어진 CFG :

|  |
| --- |
| 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 EXPR | EXPR multdiv EXPR  06: EXPR → lparen EXPR rparen | id | num  07: FDECL → vtype id lparen ARG rparen lbrace BLOCK RETURN rbrace  08: ARG → vtype id MOREARGS | ϵ  09: MOREARGS → comma vtype id MOREARGS | ϵ  10: BLOCK → STMT BLOCK | ϵ  11: STMT → VDECL | ASSIGN semi  12: STMT → if lparen COND rparen lbrace BLOCK rbrace ELSE  13: STMT → while lparen COND rparen lbrace BLOCK rbrace  14: COND → COND comp COND | boolstr  15: ELSE → else lbrace BLOCK rbrace | ϵ  16: RETURN → return RHS semi  17: CDECL → class id lbrace ODECL rbrace  18: ODECL → VDECL ODECL | FDECL ODECL | ϵ |

2. 모호함을 제거한 CFG :

|  |
| --- |
| 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 → TERM addsub EXPR | TERM  06: TERM → FACTOR multdiv TERM | FACTOR  07: FACTOR → lparen EXPR rparen | id | num | COND  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 → FACTOR comp FACTOR | boolstr  16: ELSE → else lbrace BLOCK rbrace | ϵ  17: RETURN → return RHS semi  18: CDECL → class id lbrace ODECL rbrace  19: ODECL → VDECL ODECL | FDECL ODECL | ϵ |

**II. Regular expression**

LETTER = ['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']

ZERO = ['0']

NON\_ZERO = ['1', '2', '3', '4', '5', '6', '7', '8', '9']

UNDER\_BAR = ['\_']

MINUS = ['-']

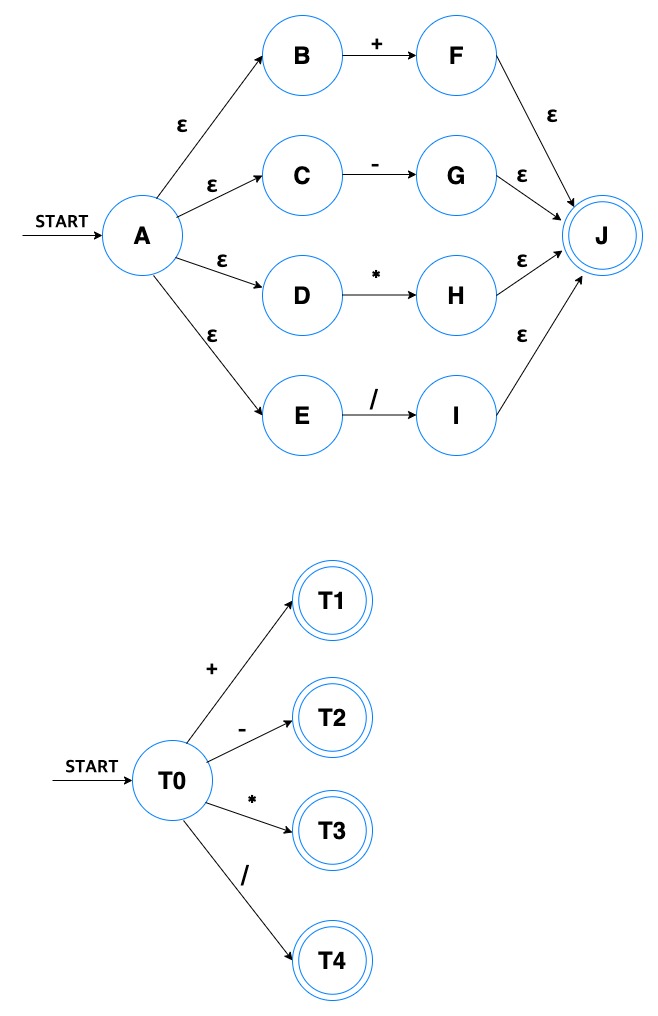
WHITE\_SPACE = [' ', '\t', '\n']

|  |  |
| --- | --- |
| Token | Regular Expression |
| VARIABLE TYPE | int | char | boolean | String | void | byte | double | float | long |
| KEYWORD | abstract | break | case | catch | class | continue | default | do | else | extends | false | finally | for | if | implements | import | instancedof | interface | native | new | null | package | private | protected | public | return | short | static | super | switch | synchronized | this | throw | throws | true | while | args | main |
| BOOLEAN STRING | true | false |
| IDENTIFIER | ( letter | under\_bar ) ( zero | non-zero | letter | under\_bar )\* |
| LITERAL STRING | “ ( zero | non-zero | letter | white\_space )\* ” |
| SIGNED INTEGER | zero | ( ( minus | ε ) non-zero ( zero | non-zero )\* ) |
| SINGLE CHARACTER | ‘ ( zero | non-zero | letter | white\_space ) ‘ |
| ARITHETIC OPERATORS | + | - | \* | / |
| ASSIGNMENT OPERATORS | = |
| COMPARISION OPERATORS | < | > | == | != | <= | >= |
| TERMINATING SYMBOL | ; |
| PAREN | { | } | ( | ) | [ | ] |
| COMMA | , |
| WHITESPACE | ( \t | | \n )\* |

**III. NFA & DFA**

**1. Arithmetic**

**- NFA graph**



**- Draw DFA**

= { A, B, C, D, E } = T0

= = { F, J } = T1

= = { G, J } = T2

= = { H, J } = T3

= = { I, J } = T4

=

=

=

=

=

=

=

=

=

=

=

=

=

=

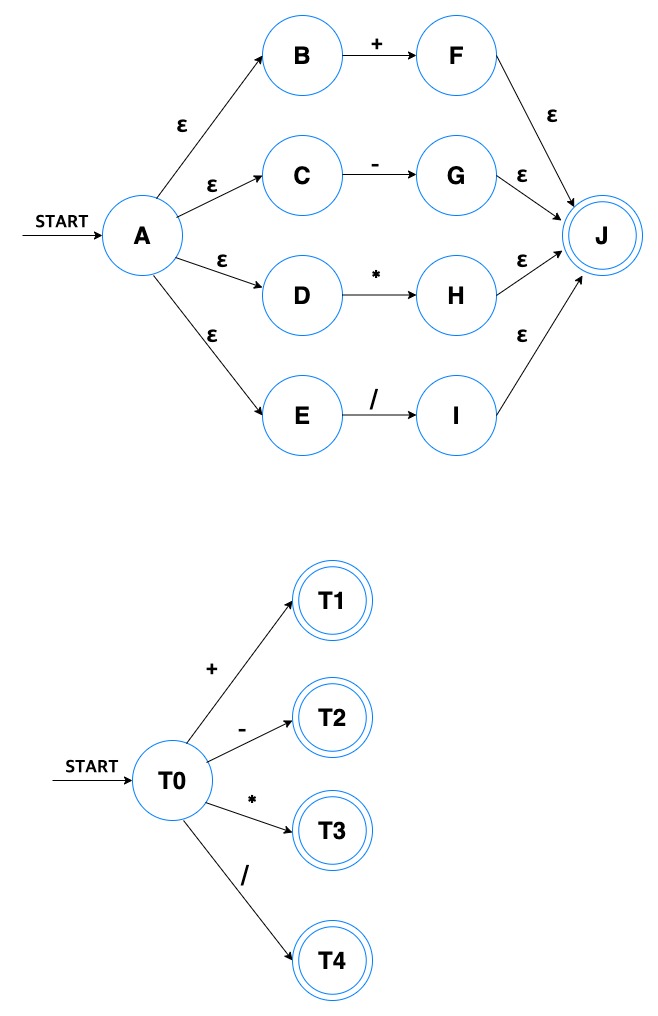
=

=

**-Transition table**

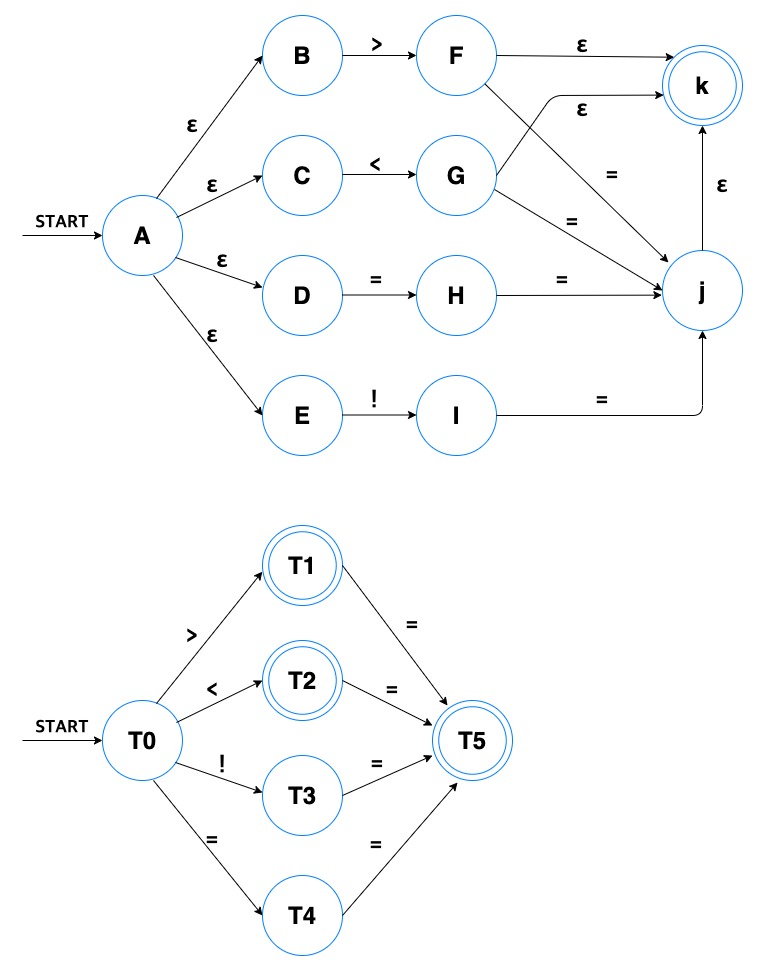
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | + | - | \* | / |
| T0 | T0 | T2 | T3 | T4 |
| T1 |  |  |  |  |
| T2 |  |  |  |  |
| T3 |  |  |  |  |
| T4 |  |  |  |  |

**- DFA graph**



**2. Comparison**

**- NFA graph**



**- DRAW DFA**

= { A, B, C, D, E } = T0

= = { F, K } = T1

= = { G, K } = T2

= = { H, K } = T3

= = { I, K } = T4

=

=

=

= = { J, K } = T5

=

=

=

= = T5

=

=

=

= = T5

=

=

=

= = T5

=

=

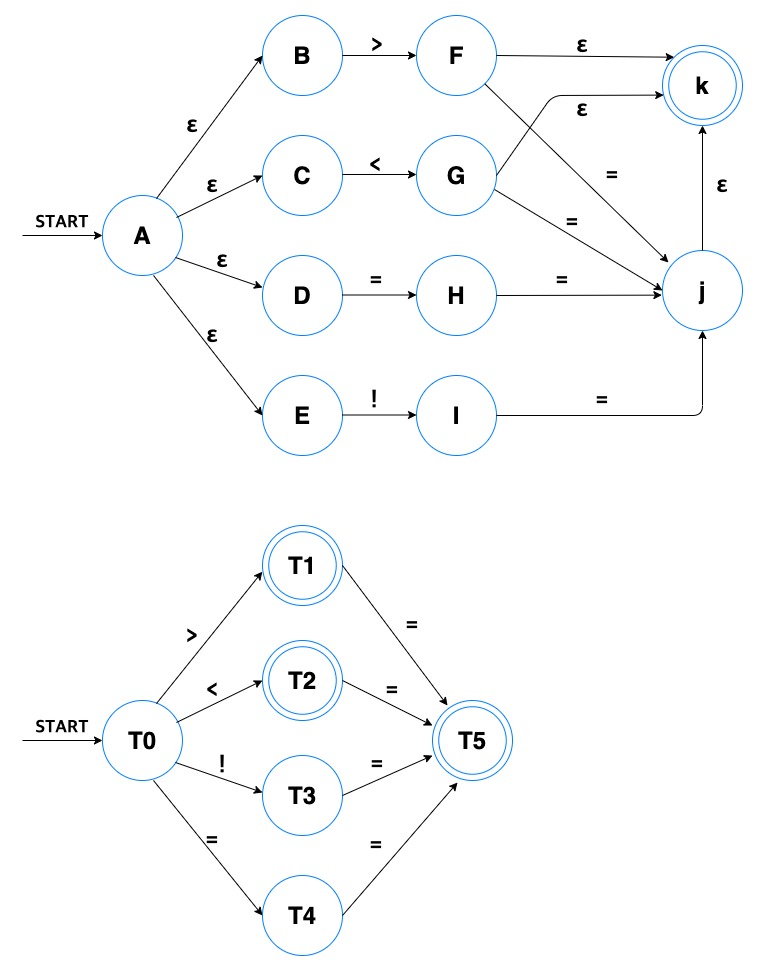
=

=

**- TRANSITION TABLE**

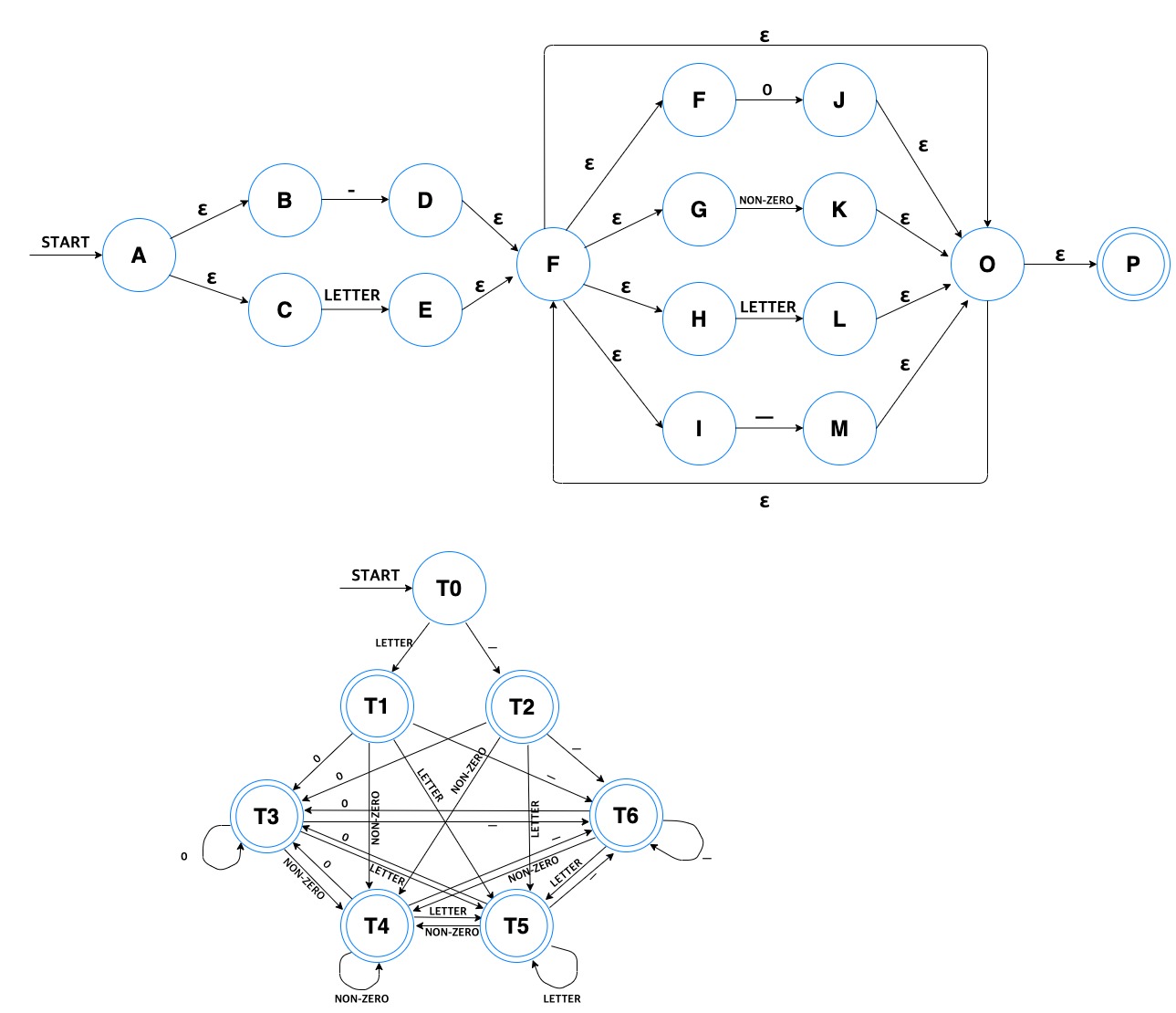
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | > | < | ! | = |
| T0 | T1 | T2 | T3 | T4 |
| T1 |  |  |  | T5 |
| T2 |  |  |  | T5 |
| T3 |  |  |  | T5 |
| T4 |  |  |  | T5 |
| T5 |  |  |  | T5 |
| T6 |  |  |  |  |

**- DFA GRAPH**



**3. Identifier**

**- DFA graph**



**- DRAW DFA**

= { A, B, C } = T0

= = { D, F, G, H, I, J, O, P } = T1

= = { E, F, G, H, I, J, O, P } = T2

=

=

= = { K, O, F, G, H, I, J, P } = T3

= = { L, O, F, G, H, I, J, P } = T4

= = { M, O, F, G, H, I, J, P } = T5

= = { N, O, F, G, H, I, J, P } = T6

= = T3

= = T4

= = T5

= =T6

= = T3

= = T4

= = T5

= =T6

= = T3

= = T4

= = T5

= =T6

= = T3

= = T4

= = T5

= =T6

= = T3

= = T4

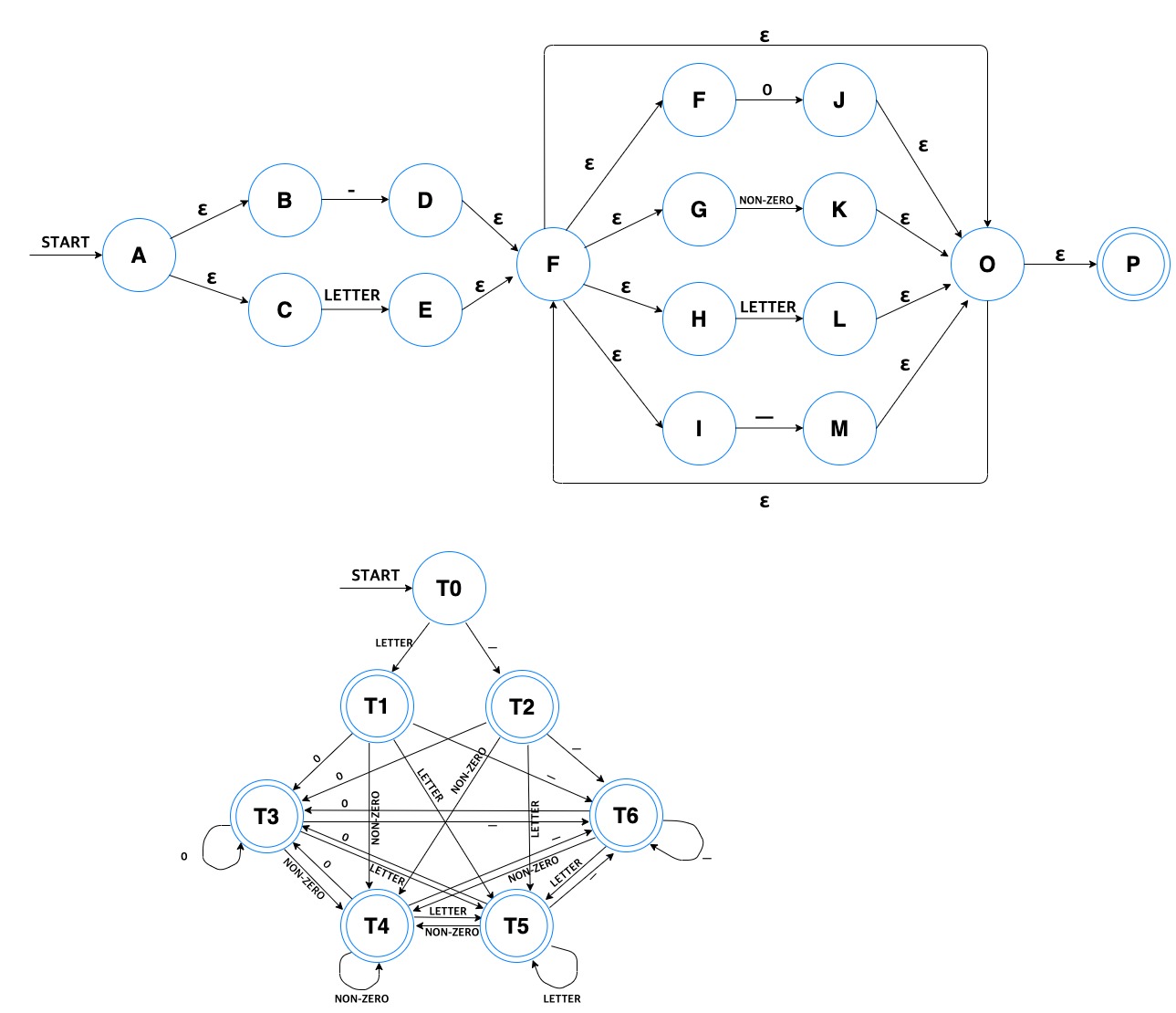
= = T5

= =T6

**- TRANSITION TABLE**

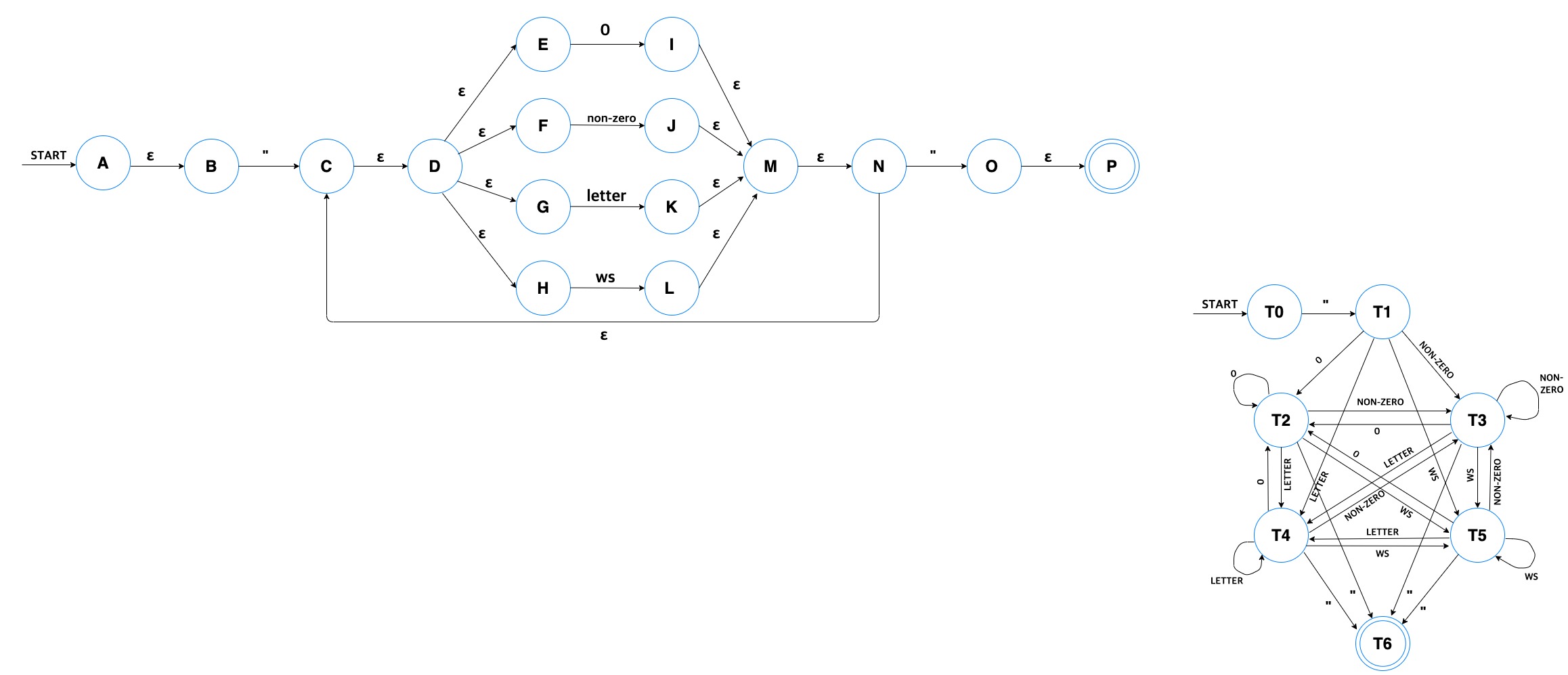
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 0 | NON-ZERO | LETTER | \_ |
| T0 |  |  | T2 | T1 |
| T1 | T3 | T4 | T5 | T6 |
| T2 | T3 | T4 | T5 | T6 |
| T3 | T3 | T4 | T5 | T6 |
| T4 | T3 | T4 | T5 | T6 |
| T5 | T3 | T4 | T5 | T6 |
| T6 | T3 | T4 | T5 | T6 |

**- DFA GRAPH**



**4. Literal String**

**- NFA GRAPH**



**- DRAW DFA**

= { A, B } = T0

=

=

=

=

= = { C, D, E, F, G, H } = T1

= = { I, M, N ,C, D, E, F, G, H } = T2

= = { J, M, N ,C, D, E, F, G, H } = T3

= = { K, M, N ,C, D, E, F, G, H } = T4

= = { L, M, N ,C, D, E, F, G, H } = T5

=

= = T2

= = T3

= = T4

= = T5

= = { O, P } = T6

= = T2

= = T3

= = T4

= = T5

= = { O, P } = T6

= = T2

= = T3

= = T4

= = T5

= = { O, P } = T6

= = T2

= = T3

= = T4

= = T5

= = { O, P } = T6

=

=

=

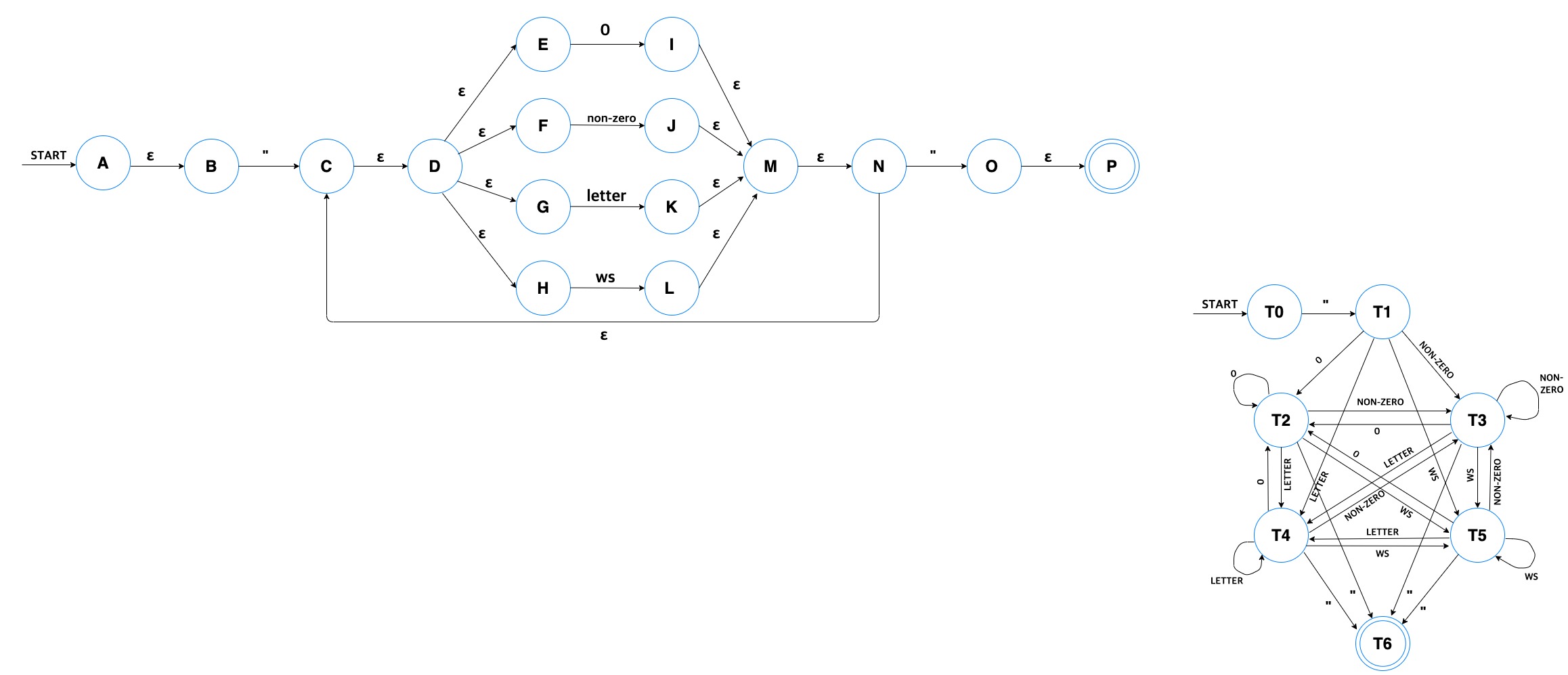
=

=

**- TRANSITION TABLE**

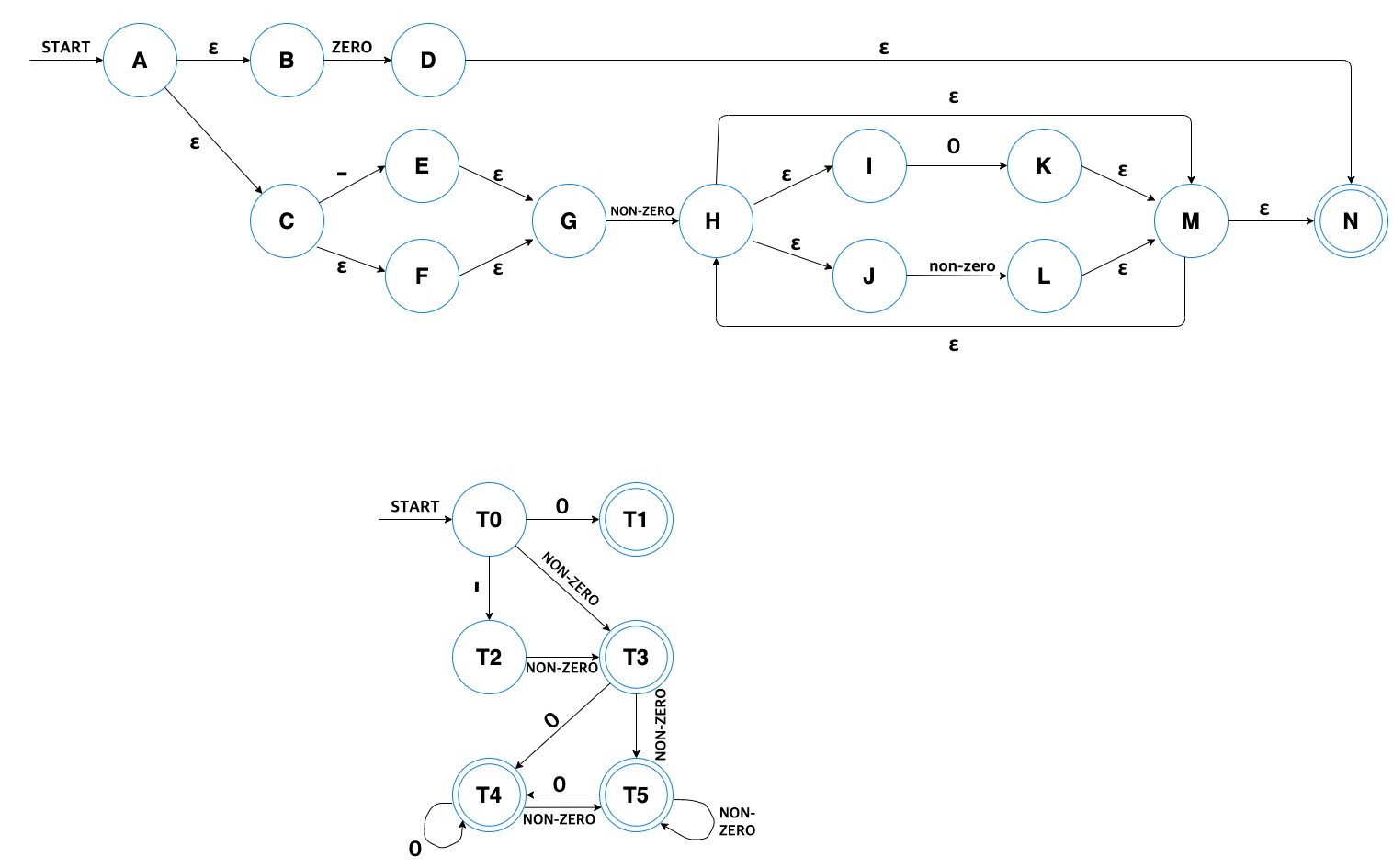
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | NON-ZERO | LETTER | WHITE\_SPACE | “  (DOUBLE\_  QUOTE) |
| T0 |  |  |  |  | T1 |
| T1 | T2 | T3 | T4 | T5 |  |
| T2 | T2 | T3 | T4 | T5 | T6 |
| T3 | T2 | T3 | T4 | T5 | T6 |
| T4 | T2 | T3 | T4 | T5 | T6 |
| T5 | T2 | T3 | T4 | T5 | T6 |
| T6 |  |  |  |  |  |

**- DFA GRAPH**



**5. Signed Integer**

**- NFA GRAPH**



**- DFA DRAW**

= { A, B, C, F, G } = T0

= = { D, N }= T1

= = { E, G } = T2

= = { H, I, J, M, N } = T3

=

=

=

=

=

= = T3

= = { K, M, N, H, I, J } = T4

=

= = { L, M, N, H, I, J } = T5

= = T4

=

= = T5

= = T4

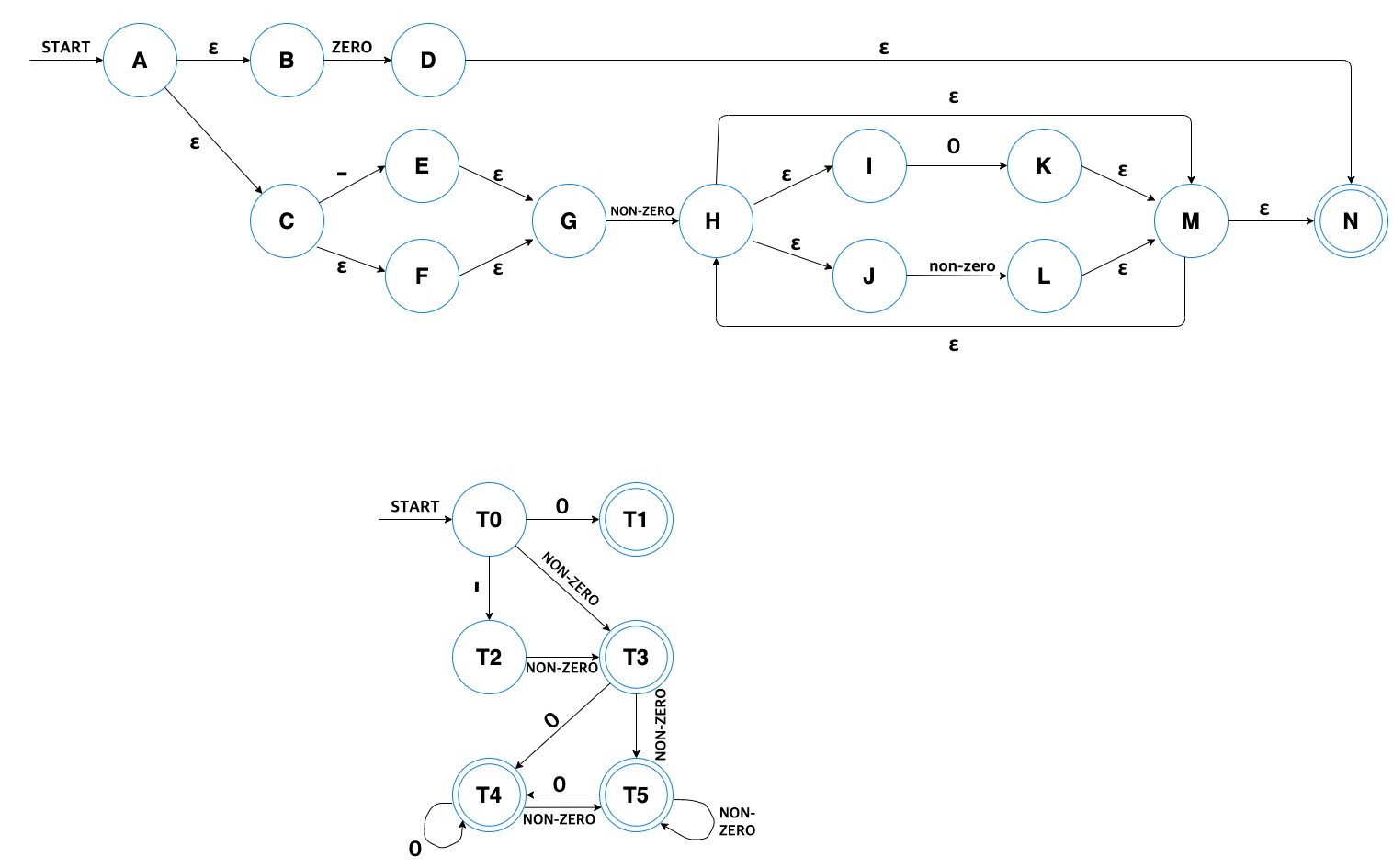
=

= = T5

**- TRANSITION TABLE**

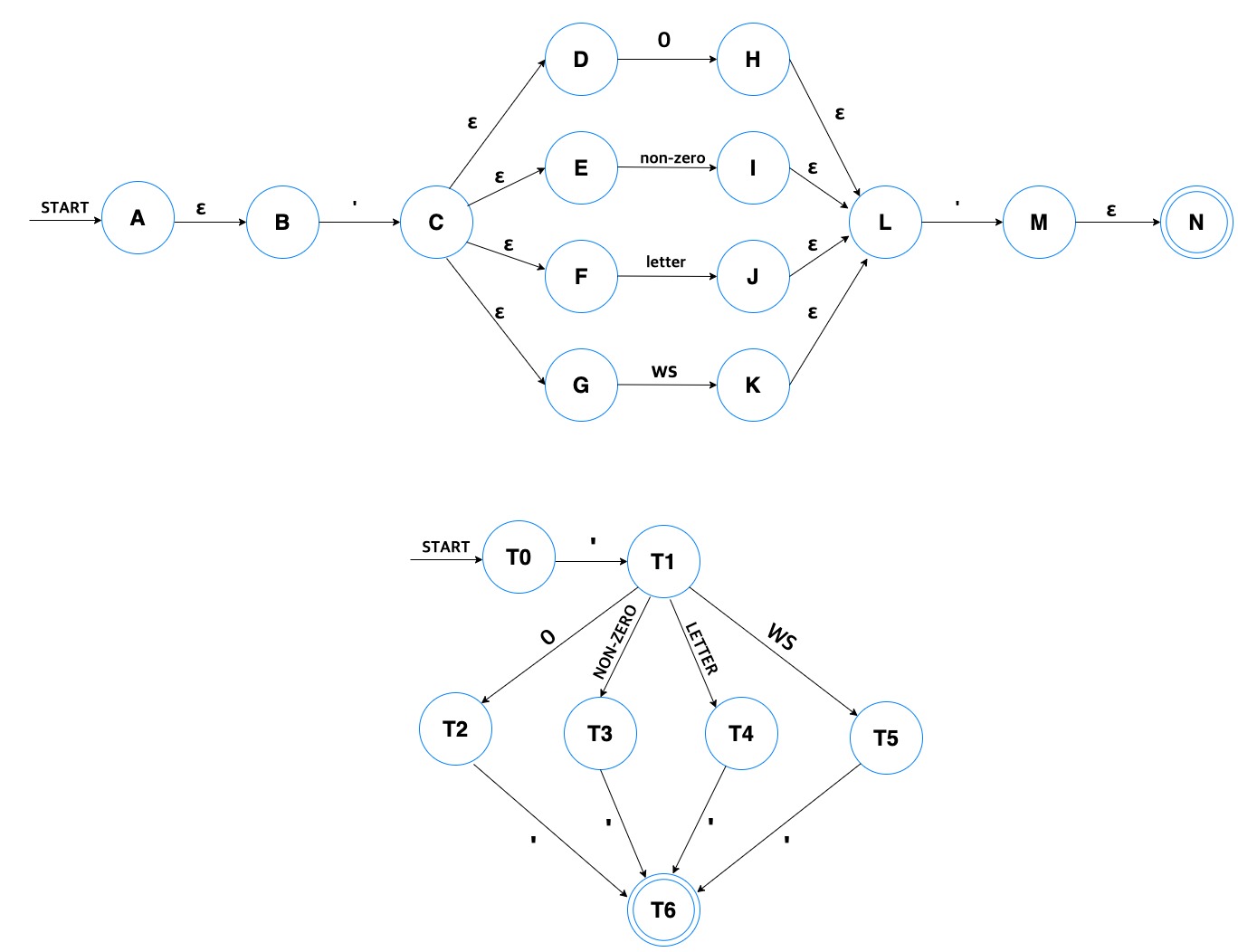
|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 | MINUS | NON-ZERO |
| T0 | T1 |  |  |
| T1 |  |  |  |
| T2 |  |  | T3 |
| T3 | T4 |  | T5 |
| T4 | T4 |  | T5 |
| T5 | T4 |  | T5 |

**- DFA GRAPH**



**6. Single Character**

**- NFA GRAPH**



**- DRAW DFA**

= { A, B } = T0

=

=

=

=

= = { C, D, E, F, G, H } = T1

= = { H, L } = T2

= = { I, L } = T3

= = { J, L } = T4

= = { K, L } = T5

=

=

=

=

=

= = { M, N } = T6

=

=

=

=

= = T6

=

=

=

=

= = T6

=

=

=

=

= = T6

=

=

=

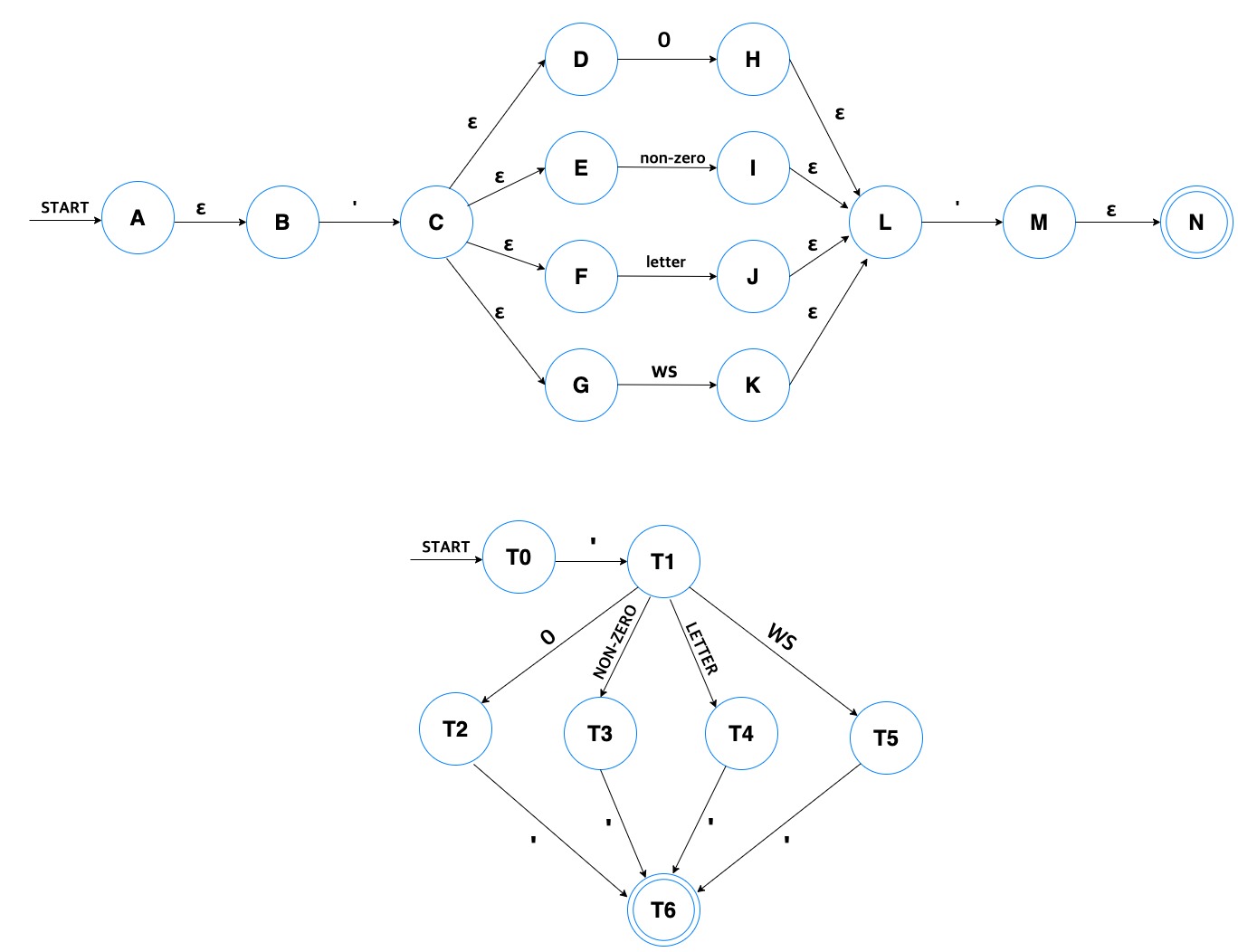
=

=

**- TRANSITION TABLE**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | NON-ZERO | LETTER | WHITE\_SPACE | ‘  (QUOTE) |
| T0 |  |  |  |  | T1 |
| T1 | T2 | T3 | T4 | T5 |  |
| T2 |  |  |  |  | T6 |
| T3 |  |  |  |  | T6 |
| T4 |  |  |  |  | T6 |
| T5 |  |  |  |  | T6 |
| T6 |  |  |  |  |  |

**- DFA GRAPH**



**IV. IMPLEMENTATION**

**0. Developing Environmemt**

|  |  |  |  |
| --- | --- | --- | --- |
| Name | OS | Language | IDE |
| 이나혁 | Mac OS 11 Big Sur  (Intel Processor) | Python3.6 | PyCharm  & Jupyter Notebook |
| 이하윤 | Mac OS 11 Big Sur  (M1 Silicon) | Python3.8 | PyCharm  & VS Code |

**1. Define Tokens & Symbols**

# define token & symbol  
ALPHABET = ['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']  
ZERO = ['0']  
MINUS = ['-']  
NON\_ZERO = ['1', '2', '3', '4', '5', '6', '7', '8', '9']  
UNDER\_BAR = ['\_']  
ARITHMETIC = ['+', '-', '\*', '/']  
COMPARISON = ['<', '>', '!', '=']  
ASSIGN = ['=']  
QUOTE = ["'"]  
DOUBLE\_QUOTE = ['"']  
SYMBOL = ['~', '@', '#', '$', '%', '^', '&', '?', '!', '|' , '\\', '.', ':']  
COMMA = [',']  
PARENS = ['(', '‘, '[', ']', '{', '}']  
SEMI = [';']  
WHITE\_SPACE = [' ', '\t', '\n']

LETTER = ALPHABET + SYMBOL  
OTHERS = SYMBOL + COMMA + PARENS + WHITE\_SPACE + SEMI + ASSIGN  
VTYPE = ['int', 'char', 'boolean', 'String', 'void', 'byte', 'double', 'float',  
 'long']  
KEYWORD = ['abstract', 'break', 'case', 'catch', 'class', 'continue', 'default',  
 'do', 'else', 'extends', 'false', 'finally', 'for', 'if', 'implements',  
 'import', 'instanceof', 'interface', 'native', 'new', 'null', 'package',  
 'private', 'protected', 'public', 'return', 'short', 'static', 'super',  
 'switch', 'synchronized', 'this', 'throw', 'throws', 'true', 'try', 'while', 'args', 'main']

**2. Implement DFA**

앞서 그린 DFA를 구현한다. state를 정의하고 현재 state를 지정한다. 입력받은 token을 한 글자씩 split해 순차적으로 DFA에 입력한다. 순서대로 입력된 input을 끝까지 읽었을 때, 현재 state가 final state에 있으면 True를 반환(accept)하고 그렇지 않을 경우 False(reject)을 반환한다.

def isArithmetic(token):

state = ['T0', 'T1', 'T2', 'T3', 'T4']   
 locate = state[0]  
  
 for value in token:  
 if locate == state[0]:  
 if value in ARITHMETIC[0]:  
 locate = state[1]  
 elif value in ARITHMETIC[1]:  
 locate = state[2]  
 elif value in ARITHMETIC[2]:  
 locate = state[3]  
 elif value in ARITHMETIC[3]:  
 locate = state[4]  
 else:  
 return False  
 else:  
 return False  
  
 if locate == state[1] or locate == state[2] or locate == state[3] or locate == state[4]:  
 return True  
 else:  
 return False

위와 같은 알고리즘으로 arithmetic, comparison, identifier, literal string, signed integer, single character, others(그 외)를 판단하는 함수 isArithmetic(token), isComparison(token), isID(token), isLiteralString(token), isSignedINT(token), isSingleCharacteRtoken), isOthers(token)을 정의한다.

**3. Check Double Quote(“) / Check Single Quote(‘)**

Double Quote(“)로 둘러싸인 것과 Single Quote(‘)로 둘러싸인 것은 각각 String과 Char로 인식하여 하나의 Lexeme을 구성해야한다. 따라서 사전 전처리를 통해 이를 묶어줄 수 있도록 한다. 각각의 묶인 구성이 Literal String을 만족시키는지, 그리고 Character를 만족시키는지 확인하는 DFA 검사의 과정은 위에 정의한 함수들로 이루어지므로 이 과정에서 추가할 필요는 없다.

이에 따라 checkDoubleQuote(list), checkSingleQuote(list)의 두 함수를 구현하여 Input Stream을 list형으로 받아 전처리된 list를 반환하는 함수를 구성하였다. 두 함수는 거의 동일한 형태를 가지기 때문에 아래에 이 중 하나의 함수만 첨부하였다.

### Double Quote(")를 체크하여 List 내 Char를 결합해주는 함수  
def checkDoubleQuote(string\_list: list):  
 result = []  
 temp = []  
 checkOn = False  
 for i in string\_list:  
 if i == '"':  
 if checkOn == True:  
 temp.append('"')  
 result.append(''.join(temp))  
 temp = []  
 checkOn = False  
 elif checkOn == False:  
 checkOn = True  
 temp.append('"')  
 else:  
 if checkOn == False:  
 result.append(i)  
 elif checkOn == True:  
 temp.append(i)  
 # List 반환  
 return result

**4. Input Stream Preprocessing**

입력 Stream을 받아 이를 Char 단위로 쪼개어 List로 반환한다. 이 때, 위(3.)에서 정의한 두 함수를 이용하여 Quote에 대한 전처리도 함께 진행한다. 이 함수가 적용된 예시는 아래와 같다.

**INPUT (String)** **>** ‘cau 123 “good”’

**OUTPUT** **(List) >** [‘c’, ‘a’, u’, ‘ ‘, ‘1’, ‘2’, ‘3’, ‘ ‘, ‘”good”’]

### 입력 Stream을 받아 이를 Char 단위로 쪼개어 List 반환 ("", '' 처리 포함)  
def preProcessing(input\_str: str):  
 # 전역 변수 사용  
 global initial\_input  
 # 결과 출력을 위한 Stream 복사  
 initial\_input = input\_str  
 # String to List  
 test = list(input\_str)  
 # "", '' 처리 수행  
 test = checkDoubleQuote(checkSingleQuote(test))  
 # List 반환  
 return test

**5. Tokenize**

구현의 가장 핵심적인 부분이다. 위 전처리 과정을 통해 String을 List로 변환하였고, 이 List가 곧 이 함수의 Input이 된다. 이 함수는 재귀(Recursion) 방식으로 구현된다. List를 입력받은 이 함수는 첫 원소부터 순차적으로 탐색해가며 Lexeme이 완성되면 그 값과 Token을 저장한다. 그리고 기존 List에서 해당 Lexeme에 대한 Char 원소들을 제거한 후 제거된 새 List를 Input으로 다시 Tokenize 함수를 실행한다. 이러한 과정이 재귀로 진행되며, Input Stream에 대한 List가 모든 Lexeme이 저장되어 최종적으로 비게 되었을 때 return하게 된다.

아래는 구현에 대한 세부적인 설명이다.

1. **Tokenize 함수의 Return (재귀 종료)**

위에서 설명했듯, 이 함수는 재귀 방식으로 동작하며 Lexeme을 추출하고 난 나머지 Char List를 새로운 Input으로 한다고 하였다. 그렇다면 언제 이 함수는 Return하느냐에 대한 부분이 이 부분이다. 분석해야할 모든 String, 즉 Code(String)에 존재하는 각각의 모든 Char가 분석되어 List가 비게 되면 이 함수는 return하게 된다.

# 더 이상 분석할 String이 존재하지 않으면 tokenize 함수 반환  
if len(input\_string) == 0:

return

1. **검사 대상 리스트 구현**

순차적으로 Char를 읽으며 DFA 검사를 수행하고 Lexeme이 완성되었을 때 Token를 저장할 수 있어야 한다. 예를 들어 ‘if123’ 라는 ID에 대한 검사가 수행이 된다면 최소한 6번(‘i’, ‘if’, ‘if1’, ‘if12’, ‘if123’, 그리고 Reject)에 대한 검사가 반드시 진행되어야 한다. 이 단계는 이러한 검사 대상 리스트를 새롭게 제작하는 단계이다. 이 부분에 대한 예시는 아래와 같다.

**BEGIN >** [‘a’, ‘b’, ‘c’, ‘1’]

**END >** [[‘a’], [‘a’, ‘b’], [‘a’, ‘b’, ‘c’], [‘a’, ‘b’, ‘c’, ‘1’]]

# 반복문을 통해 검사할 Char 리스트를 재구성  
# ex. 'abc1' -> [['a'], ['a','b'], ['a','b','c'], ['a','b','c','1']]  
while True:  
 test\_iter = []  
 for i in range(len(input\_string)):  
 test\_iter.append(input\_string[0:i + 1])  
 break

이후, 이 List에 대한 반복 검사가 수행된다.

# 재귀 방식의 Char List 반복 검사  
for i in test\_iter:  
 # 검사 횟수 Count  
 count = count + 1

1. **Lexeme 완성 시**

먼저 DFA 검사를 하였을 때 각 검사에 대해 모두 Reject인 경우 바로 직전의 검사까지의 Char들로 Lexeme을 구성할 수 있다. Lexeme 구성을 마치면 새로운 Input Stream을 구성하게 되며 Loop를 탈출하게 된다.

# 모든 DFA 검사에서 Reject되는 경우 반복문 break 후 다음 Char List 검사  
if ((isArithmetic(temp\_str) == False)  
 and (isComparison(temp\_str) == False)  
 and (isID(temp\_str) == False)  
 and (isLiteralString(temp\_str) == False)  
 and (isSignedINT(temp\_str) == False)  
 and (isSingleCharacteRtemp\_str) == False)  
 and (isOthers(temp\_str) == False)):   
  
 # Reject 직전의 Char까지 하여 Lexeme 구성  
 token = ''.join(i[0:-1])  
  
 # 다음 Input Stream 구성을 위해 Count - 1 수행  
 count = count - 1  
  
 break

1. **DFA 검사**

Arithmetic, Comparison, ID, LiteralString, SignedINT, SingleCharacter, Others에 대한 DFA 검사를 수행한다. 그리고 각 검사 종류 별로 Lexeme이 완성되기 전까지 Accept된 횟수를 저장한다. 이는 (-) Symbol이 INT로써 사용될지 Operator로써 사용될지 구분하는 용, 혹은 다른 애매한 상황을 구분하는 용으로 사용될 수 있기 때문이다.

Arithmetic과 Comparison에 대한 검사는 간단하게 구현된다.

# Arithmetic DFA 검사  
if (isArithmetic(temp\_str) == True):  
 count\_Arith = count\_Arith + 1  
 token\_type = 'ARITHMETIC'

# Comparision DFA 검사  
if (isComparison(temp\_str) == True):  
 count\_Compa = count\_Compa + 1  
 token\_type = 'COMPARISON'

Identifier의 경우 ID로 일단 분류 되었지만 Java 언어에서 사용되는 예약어(Keyword) 혹은 데이터 타입을 명시한 VType일 수 있다. 따라서 추가 조치를 수행해주어야 한다.

# Identifier DFA 검사  
if (isID(temp\_str) == True):  
 count\_ID = count\_ID + 1  
  
 # 사전에 지정된 예약어(Keyword)인 경우  
 if temp\_str in KEYWORD:  
 token\_type = 'KEYWORD'  
  
 # 데이터 타입(Value Type)인 경우  
 elif temp\_str in VTYPE:  
 token\_type = 'VTYPE'  
  
 # 그 외의 경우 ID  
 else:  
 token\_type = 'ID'

Literal String, Signed Integer, Single Character에 대한 검사는 간단하게 구현된다.

# Literal String DFA 검사  
if (isLiteralString(temp\_str) == True):  
 count\_Liter = count\_Liter + 1  
 token\_type = 'LITERAL'  
  
# Signed Integer DFA 검사  
if (isSignedINT(temp\_str) == True):  
 count\_SiINT = count\_SiINT + 1  
 token\_type = 'INT'  
  
# Single Character DFA 검사  
if (isSingleCharacteRtemp\_str) == True):  
 count\_SiChr = count\_SiChr + 1  
 token\_type = 'CHAR'

DFA 검사를 통해 Others로 분류된 Lexeme은 해당 value에 따라 무슨 Token인지 추가로 분류해줄 필요가 있다. 예를 들면 기호(Symbol), 콤마(Comma), 괄호(Parens, ex. []{}()), White Space, 세미콜론(Semi), 할당 연산자(=)이 있다.

# 그 외의 경우(Others)에 대한 DFA 검사  
if (isOthers(temp\_str) == True):  
 count\_Others = count\_Other + 1  
 token\_type = 'OTHERS'  
  
 # Others는 기호(Symbol), 콤마(Comma), 괄호(Parens, ex.[]{}()), White Space, 세미콜론(Semi), 할당 연산자(Assign,=)으로 구성  
 OTHERS = SYMBOL + COMMA + PARENS + WHITE\_SPACE + SEMI + ASSIGN  
  
 # 사전에 지정된 기호(Symbol)인 경우  
 if temp\_str in SYMBOL:  
 token\_type = 'SYMBOR'  
  
 # 사전에 지정된 콤마(Comma)인 경우  
 elif temp\_str in COMMA:  
 token\_type = 'COMMA'  
  
 # 사전에 지정된 괄호 종류(Parens)인 경우  
 elif temp\_str in PARENS:  
 token\_type = 'PARENS'  
  
 # White Space(' ', '\n', '\t')의 경우  
 elif temp\_str in WHITE\_SPACE:  
 token\_type = 'WHITE\_SPACE'  
  
 # 세미콜론(;)의 경우  
 elif temp\_str in SEMI:  
 token\_type = 'SEMI'  
  
 # 할당 연산자(=)의 경우  
 elif temp\_str in ASSIGN:  
 token\_type = 'ASSIGN'

1. **마지막 Lexeme 검사인 경우**

마지막 Lexeme 검사인 경우 Reject이 발생되지 않을 수 있다. 이 경우 정상적으로 Accept된 DFA를 확인하여 Token의 타입을 확인한 후 저장하게 된다.

# Char List의 마지막 검사의 경우  
if (test\_iter.index(i) == len(test\_iter) - 1) and (True in [isArithmetic(temp\_str), isComparison(temp\_str), isID(temp\_str), isLiteralString(temp\_str), isSignedINT(temp\_str), isSingleCharacteRtemp\_str), isOthers(temp\_str)]):  
  
 try:  
 # (-) Symbol 처리, 연산자가 될 것인지, Integer 처리할 것인지 결정  
 if (count\_Arith >= 1) and (count\_SiINT >= 1) and (token\_key[-1] in ['INT', 'ID']):  
 # -- Arithmatic DFA Accept 횟수와 Signed Interger DFA Accept 횟수가 모두 1 이상이고,  
 # -- 바로 직전 Token이 Int 혹은 ID일 경우 연산자 처리  
 token = temp\_str[0]  
 token\_type = 'ARITHMETIC'  
 else:  
 token = temp\_str  
  
 # (-) Symbol이 Input의 맨 앞에 나온 경우 Out of List range 예외 처리  
 except:  
 token = temp\_str

1. **모든 검사에 대해 Lexeme이 결정되지 않을 경우**

모든 Char List, 그리고 각각의 DFA 검사에 대해 Lexeme이 결정되지 않을 수 있다. 이 경우 No Token 처리한다.

# 모든 DFA 검사에서 Reject되어 Lexeme 분류가 이루어지지 않은 경우  
if token == "":  
 token = "NO TOKEN"

1. **Lexeme 구성 시 새로운 Input List를 생성, 재귀 호출**

Lexeme이 구성되면 Input List에서 해당 Lexeme을 구성하는데 사용되었던 Char들을 제거한 후 새로운 Input List를 생성한다. 그리고 이를 다시 tokenize() 함수에 넣는 재귀 호출을 진행하게 된다. 예를 들면 아래와 같다.

**BEFORE >** [‘1’, ‘2’, ‘3’, ‘a’, ‘b’, ‘c’, ‘;’]

**Store** <INT, 123>

**AFTER >** [‘a’, ‘b’, ‘c’, ‘;’] **=> tokenize(Recursion)!**

# 재귀 함수의 Input이 될 새로운 Input String List를 결정  
try:  
 if (count\_Arith >= 1) and (count\_SiINT >= 1) and (token\_key[-1] in ['INT', 'ID']):  
 token = temp\_str[0]  
 token\_type = 'ARITHMETIC'  
  
 new\_test = input\_string[1:]  
  
 else:  
 new\_test = input\_string[count:]  
  
# (-) Symbol이 Input의 맨 앞에 나온 경우 Out of List range 예외 처리  
except:  
 new\_test = input\_string[count:]  
  
# print("FINDME", input\_string)  
  
# 결정된 Lexeme을 전역 변수로 선언된 List에 삽입  
token\_value.append(token)  
  
# 결정된 Token Type을 전역 변수로 선언된 List에 삽입  
token\_key.append(token\_type)  
  
# 재귀 방식으로 나머지 Input String에 대한 Tokenize 계속 실행  
tokenize(new\_test)

**6. Save Result**

추출된 Token 정보를 저장하여 Output File을 생성한다. 이 때 저장되는 파일명은 <input\_file\_name>\_output.txt’이다.

### 결과를 새로운 파일에 출력 및 저장하는 함수  
def save\_result():  
 global token\_value  
 global token\_key  
 # Output File에 대한 이름은 '<input\_file\_name>\_output.txt'로 저장됨  
 save = open(file\_path + '\_output.txt', 'w')  
 for i, j in zip(token\_value, token\_key):  
 # White Space를 제외하고 저장  
 if j != 'WHITE\_SPACE':  
 save.write("<" + j + ", " + i + ">\n")  
 # print("<"+j+",", i+">")  
 print("Successfully token list saved")  
 save.close

**7. 전체 Process 실행**

Input file은 Command에서 Input File명을 받아와 읽게 된다. 함수로 구성된 각 단계를 거친 후 전체 Lexical Analyzer는 종료된다.

# Command 내 인자 개수 확인  
if len(sys.argv) != 2:  
 print("Insufficient arguments")  
 sys.exit()  
  
# input File 열기  
file\_path = sys.argv[1]  
f = open(file\_path, 'r')

### 전체 Lexical Analyzer Flow에 대한 Process 실행  
def process(input\_str: str):  
 global token\_value  
 global token\_key  
 token\_value = []  
 token\_key = []  
 initial\_input = ''  
 # PreProcessing  
 test = preProcessing(input\_str)  
 # Tokenize  
 tokenize(test)  
 # Save Result  
 save\_result()  
  
  
# Command에서 받은 Code에 대한 Lexical Analize Process 호출  
process(f.read())

**8. (-) Symbol에 대한 Issue**

- 기호는 INT Token의 한 Char로 간주될 수도 있고, 혹은 Operator로 간주될 수 있다. 상황에 따라 다르기 때문에 별도의 과정을 거쳤다. 아래는 전체 코드의 일부분이다. 조건문의 조건을 확인하면 count\_Arith, count\_SiINT가 모두 1 이상이고, 해당 Lexeme 추출 직전에 추출된 Lexeme의 종류가 Integer 혹은 Identifier인지 확인하고 있다.

여기서 count\_Arith는 DFA에 대한 검사를 진행했을 때 Lexeme에 대한 Token을 결정짓기 전까지의 누적 Arith\_DFA 검사에 대한 Accept 횟수이다. 마찬가지로 count\_SiINT Ehgks Token 결정 전까지의 누적 Singed Int DFA 검사에 대한 Accept 횟수이다. – 심볼이 연산자로 분류되기 위해선 누적 횟수가 두 count에 대해 모두 1 이상이어야 한다는 실험 결과가 있었다.

try:  
 # (-) Symbol 처리, 연산자가 될 것인지, Integer 처리할 것인지 결정  
 if (count\_Arith >= 1) and (count\_SiINT >= 1) and (token\_key[-1] in ['INT', 'ID']):  
 # -- Arithmatic DFA Accept 횟수와 Signed Interger DFA Accept 횟수가 모두 1 이상이고,  
 # -- 바로 직전 Token이 Int 혹은 ID일 경우 연산자 처리  
 token = temp\_str[0]  
 token\_type = 'ARITHMETIC'  
 else:  
 token = temp\_str  
  
# (-) Symbol이 Input의 맨 앞에 나온 경우 Out of List range 예외 처리  
except:  
 token = temp\_str

**V. Experiment**

**- Case 1 :**

|  |  |
| --- | --- |
| input | output |
| -0 0abc0 123if 123if0 ' ' a-1 | <ARITHMETIC, -> <INT, 0> <INT, 0> <ID, abc0> <INT, 123> <KEYWORD, if> <INT, 123> <ID, if0> <CHAR, ' '> <ID, a> <ARITHMETIC, -> <INT, 1> |

**- Case 2:**

|  |  |
| --- | --- |
| input | output |
| 001 0010 0010a0010 0010-10 0010--10 | <INT, 0> <INT, 0> <INT, 1> <INT, 0> <INT, 0> <INT, 10> <INT, 0> <INT, 0> <INT, 10> <ID, a0010> <INT, 0> <INT, 0> <INT, 10> <ARITHMETIC, -> <INT, 10> <INT, 0> <INT, 0> <INT, 10> <ARITHMETIC, -> <INT, -10> |

**- Case 3:**

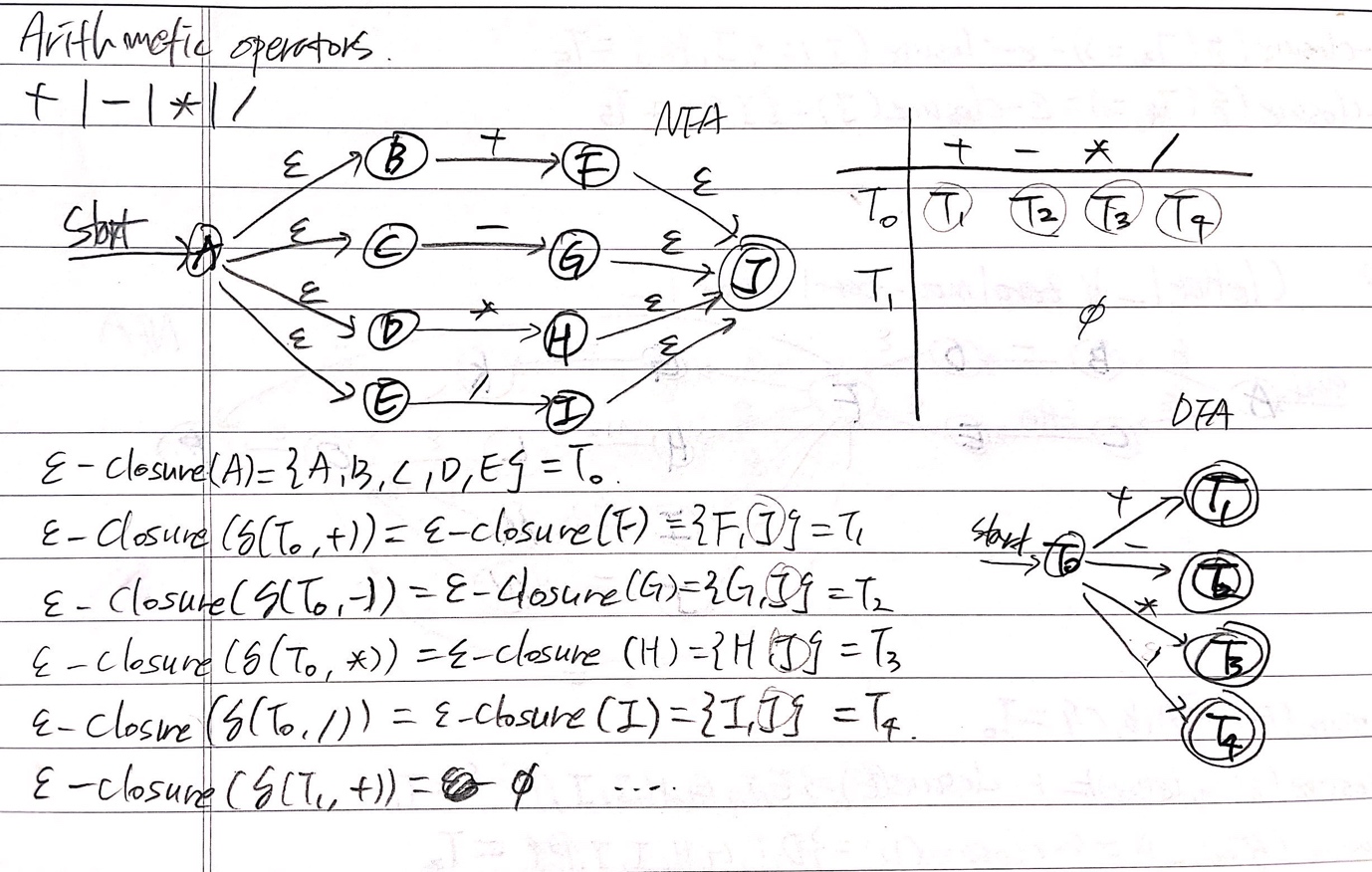
|  |  |
| --- | --- |
| input | output |
| int main(){char if123='1';int 0a=a+-1;return -0;} | <VTYPE, int> <KEYWORD, main> <PARENS, (> <PARENS, )> <PARENS, {> <VTYPE, char> <ID, if123> <ASSIGN, => <CHAR, '1'> <SEMI, ;> <VTYPE, int> <INT, 0> <ID, a> <ASSIGN, => <ID, a> <ARITHMETIC, +> <INT, -1> <SEMI, ;> <KEYWORD, return> <ARITHMETIC, -> <INT, 0> <SEMI, ;> <PARENS, }> |

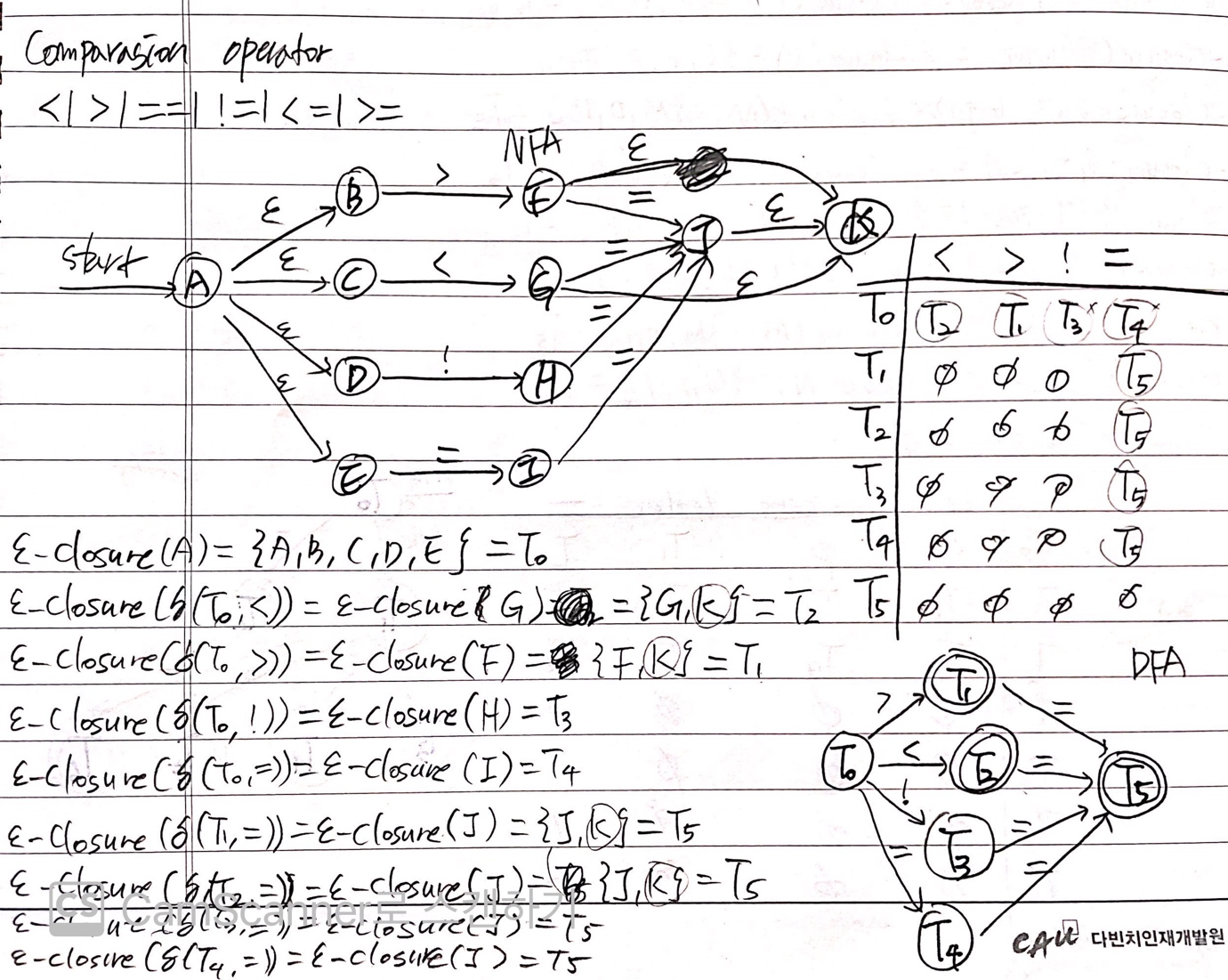
**- Case 4:**

|  |  |
| --- | --- |
| input | output |
| package test; public abstract class Test{  int a;  public Test(){  a=-*10*;  } } | <KEYWORD, package> <ID, test> <SEMI, ;> <KEYWORD, public> <KEYWORD, abstract> <KEYWORD, class> <ID, Test> <PARENS, {> <VTYPE, int> <ID, a> <SEMI, ;> <KEYWORD, public> <ID, Test> <PARENS, (> <PARENS, )> <PARENS, {> <ID, a> <ASSIGN, => <INT, -10> <SEMI, ;> <PARENS, }> <PARENS, }> |

**VI. Appendix**

**- NFA and DFA with transition table write down by hand**



텍스트, 서류이(가) 표시된 사진

자동 생성된 설명텍스트, 서류이(가) 표시된 사진

자동 생성된 설명텍스트, 서류이(가) 표시된 사진

자동 생성된 설명텍스트, 서류이(가) 표시된 사진

자동 생성된 설명