Compiler

Spring, 2022

School of Software, CAU

**Project #1**

**- Lexical Analyzer –**

컴퓨터공학과 20175371 강명석

**1. define Tokens**

토큰은 <Token Name, Token Value>의 쌍으로써 특정한 어휘를 대표하는 symbol을 의미한다. 주어진 어휘 사양에 해당하는 내용을 토대로 토큰을 정의하면 다음과 같다..

[표 1] Token

|  |  |
| --- | --- |
| Type | int, char, boolean, string |
| Integer | 0, 1, 22, 123, 56, -1, -22, -123, -56, … |
| Char | ‘a’, ‘1’, ‘ ‘, ‘&’,… |
| Bool | true, false |
| Literal | “Hello World”, “My Student ID is 202222”, … |
| Id | I, j, k, abc, ab\_123, func1, func\_, \_\_func\_bar\_\_ , … |
| Keyword | If, else, while, class, return |
| Arithmetic Operator | +, -, /, \* |
| Assign | = |
| Compare | <. >, ==, !=, <=, >= |
| SemiColon | ; |
| Lparen | ( |
| RParen | ) |
| LBrace | { |
| RBrace | } |
| LBraket | [ |
| RBraket | ] |
| Comma | , |
| WhiteSpace | \t, \n, blanks |

**2. Regular Expression**

정의한 토큰정보를 토대로 Regular Expression을 작성한다.

-----------------------------------------------------------------------------------

MERGED = Type | Integer | Char | Literal | Bool | Id | Keyword | Operator | SpecialSymbol | WhiteSpace

NZeroDigit = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Letter = a | b | c ... | y | z | A | B ... | Y | Z

Alphabet = Letter | Digit | ! | ' | " | # | $ | % | & | ~ | ( | ) | \* | + | , | - | . | / | : | ; | < | = | > | ? | @ | [ | } | ] | ^ | \_ | ` | { | | | \t | \n | \r

Arithmetic = + | - | \* | /

Assign = =

Compare = < | > | == | != | <= | >=

Type = int | char | boolean | string

Integer = 0 | -NZeroDigit(Digit)\* | NZeroDigit(Digit)\*

Char = 'Alphabet'

Literal = "(Alphabet)\*"

Bool = true | false

Id = (*\_|Letter)(Letter|Digit|\_*)\*

KeyWord = if | else | while | class | return

Operator = Arithmetic | Assign | Compare

SpecialSymbol = ; | ( | ) | { | } | [ | ] | ,

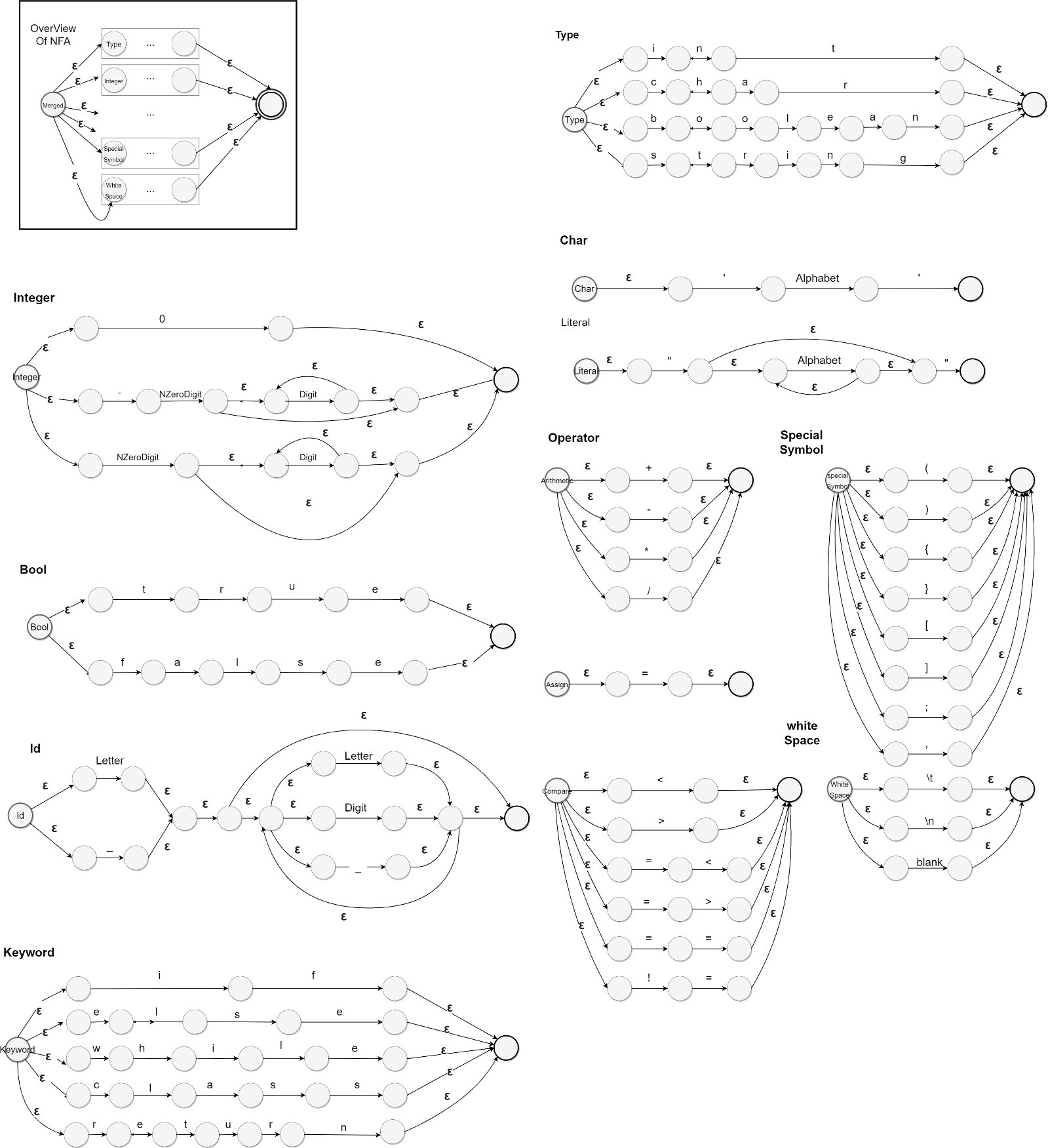
WhiteSpace = [blank] | \t | \n

-----------------------------------------------------------------------------------------------------------------

이때 본 프로젝트에서 입력 문자의 정의역이 특정되어 있지 않아서 화면에 출력가능한 문자들의 집합(문자, 숫자, 특수문자)을 입력 문자의 정의역으로 설정하였으며, 해당 정의역을 의미하는 단어로 ‘Alphabet’을 사용했다.

**3. NFA**

Regular Expression을 토대로 NFA를 작성한다. 그림 1은 NFA를 토큰단위로 나누어서 그린 그림이고, 그림 2는 NFA를 하나의 페이지에 모두 그린 그림이다. NFA,DFA를 Draw.io 사이트를 이용했다.

[](https://user-images.githubusercontent.com/33647663/162559648-feedacb0-a2de-4776-9fca-70799ec5a5ec.png)

[그림 1 – NFA Graph]

텍스트, 자연, 비이(가) 표시된 사진

자동 생성된 설명

[그림 2 – NFA Graph]

**4. DFA & Transition Table**

NFA를 DFA로 변환하는 작업에서, Keyword, Type, Bool 3가지 토큰에 대한 내용은 제외했다. 이 세가지 토큰을 제외하는 이유는 다음과 같다.

1. 위의 토큰 타입들은 모두 문자열로 이루어져 있다. 해당 타입들을 Transition Table에 포함하게 된다면, a-z 까지의 각각의 symbol이 들어올 경우에 대해서 모두 state를 따로 만들어 줘야 한다. 만약 이 세가지 타입을 제외하고 만든다면 나머지 토큰들은 a-z,A-Z를 모두 하나의 symbol 묶음인 Letter로 표현하여 Transition Table의 크기를 훨씬 줄일 수 있다.

2. 3가지 타입의 토큰들은 주어진 Specification에서 모두 Id의 정규표현식에 포함이 된다. 즉, DFA로 Id를 판단하고 난 뒤, Id의 lexeme에서 미리 정의된 Keyword, Type, Bool의 문자열들만 따로 비교하는 것이 구현관점에서 훨씬 간결하며 유지보수의 관점에서도 이해하기 쉽다.

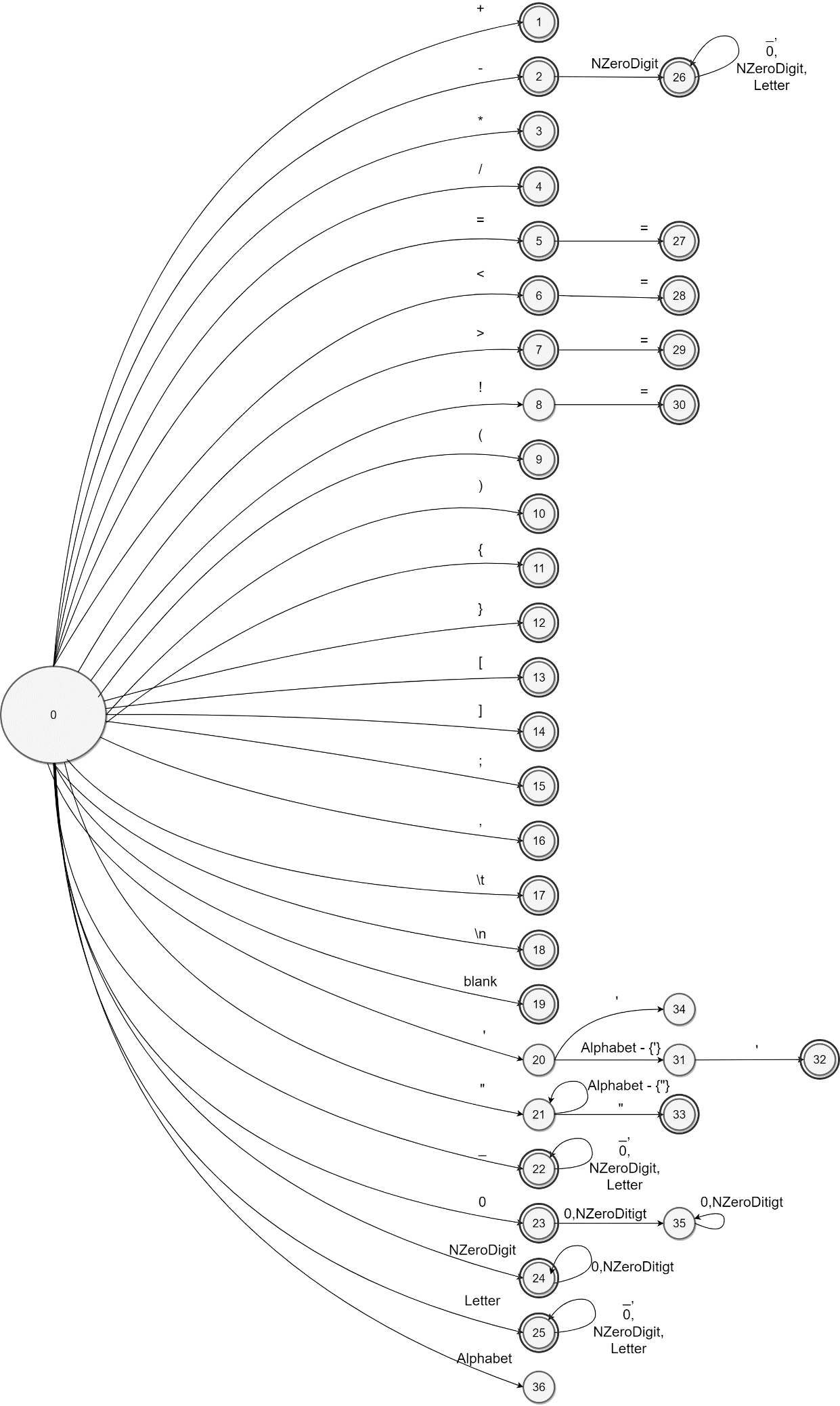
위의 이유를 고려하여 Transition Table에서는 Keyword, Type, Bool의 케이스는 고려하지 않았고, 만약 해당 하는 character stream이 들어온다면 모두 Id 토큰으로 나올 것이다. 이는 구현 단계에서 처리해 줄 것이다. [그림 3]이 NFA를 이용하여 만든 Transition Table로 Token의 경우 해당 State에서 종료된다면 해당 State에 해당하는 토큰이 무엇인지를 나타낸 정보이다. 중간에 !ERR로 표시한 state는 Terminal State가 아닌 상태로 해당 state에서 DFA가 종료된다면 이는 오류임을 나타낸다. 예를들어 20번 State의 경우 현재 single quote를 하나 읽은 상태로 만약 이 상태에서 종료 된다면 이는 오류 임을 나타낸다.

그리고 해당 테이블에서 alphabet의 의미는 표에서 같은 행에서 state 를 Transition을 일으키는 문자를 제외한 모든 문자로써 가장 마지막 우선순위를 가진다. 20번 state에서는 single quote가 들어오면 34번 state로 상태를 변환하는 것이고, 이때의 alphabet은 single quote를 제외한 모든 문자를 의미한다. 0번 state에서는 +,-, … Nzero, Letter을 제외한 모든 문자로 그 의미를 작성하였다.

테이블이(가) 표시된 사진

자동 생성된 설명

[그림 3 – Transition Table]



[그림 4 – DFA Graph]

**5. Implementation**

Lexical Analyzer는 파이썬을 이용하여 구현하였다. 주요 파일은 lex.py, table.py로 구성되며 test\_code 폴더에 예제 코드를 저장하였다.

**(1) table.py**

DFA를 만드는데 사용한 Transition table 내용을 Python 자료구조로 변환한 내용으로써 Transition\_table과 State\_table이 있다.

**Transition\_table** : 현재 상태에서 들어오는 symbol에 따라 상태변환이 어떻게 되는지를 나타낸 Table이다. Transition\_table[0][‘+’]와 같이 사용하며 그 결과로 1을 반환하며 이는 0의 상태에서 ‘+’를 읽으면 상태 1로 이동을 한다는 의미이다. 아래는 요약된 Transition\_table 구조이다.

|  |
| --- |
| **Transition\_table = {**  **0 : {**  **'+' : 1 ,**  **'-' : 2,**  **'\*' : 3,**  **.... // 생략**  **'Letter' : 25,**  **'Alphabet' : 36**  **},**  **1 : {},**  **2 : {**  **'NZDigit' : 26**  **},**  **3 : {},**  **4 : {},**  **5 : {**  **'=' : 27,**  **},**  **6 : {**  **'=' : 28,**  **},**  **7 : {**  **'=' : 29,**  **},**  **... // 생략**  **35 : {**  **'0' : 35,**  **'NZDigit' : 35**  **},**  **36 : {}**  **}** |

**State\_table** : 현재 상태에서 정의되지 않은 Input Symbol이 들어오면, 현재 상태에 따른 토큰 정보와 lexeme을 반환해야 한다. State\_table에는 현재 상태에 따른 Token 정보를 담고 있는 테이블이다. State\_table[1]은 1의 상태에서 토큰 정보를 반환하여 아래에서 ‘ArithmeticOperator’를 반환하게 된다. 아래는 State\_table 구조이다.

|  |
| --- |
| **State\_table = [**  **None,**  **'ArithmeticOperator', # +**  **'ArithmeticOperator', # -**  **'ArithmeticOperator', # \***  **'ArithmeticOperator', # /**  **'Assign', # =**  **'Compare', # <**  **'Compare', # >**  **'ERR', # single !**  **'LPAREN', # (**  **'RPAREN', # )**  **'LBRACE', # {**  **'RBRACE', # }**  **'LBRAKET', # [**  **'RBRAKET', # ]**  **'SCOLON', # ;**  **'COMMA', # ,**  **'WHITESPACE', # \t**  **'WHITESPACE', # \n**  **'WHITESPACE', # 0x20**  **'ERR', # finished with single quote**  **'ERR', # finished with double quote**  **'Id', # \_(digit|letter|\_)\***  **'Integer', # 0**  **'Integer', # [1-9][0-9]\***  **'Id', # letter(digit|letter|\_)\***  **'Integer', # -[1-9][0-9]\***  **'Compare', # ==**  **'Compare', # <=**  **'Compare', # >=**  **'Compare', # !=**  **'ERR', # Only Single symbol in ''**  **'Char', # 'Alphabet'**  **'String', # "(Alphabet)\*"**  **'ERR', # '' is not allowed.need exactly one symbol in ''**  **'ERR', # 00,001,0123 is not allowed**  **'ERR' # Unavailabe Input Character**  **]** |

**(2) lex.py**

메인 lexical analyzer 파일로써 주요 함수는 다음과 같다.

|  |  |
| --- | --- |
| main | Argument 파싱, 파일 Read, lexical()함수를 호출, 결과를 저장하는 기능 |
| Lexical(text) | text파일에서 index를 하나씩 증가시키며 input symbol을 처리. 주요 로직은 EOF에 도달할 때 까지, 현재 상태와 input symbol을 이용하여 State를 변경시키면서 Transition Table에서 현재 상태에서 이동할 수 없는 input symbol이 들어왔을 때 token 정보와 lexeme 정보를 저장하는 기능.  만약 현재 state가 non terminal state인데 해당하지 않는 input symbol이 들어온다면 report\_error 함수를 호출하여 에러 출력 |
| Report\_error(state,lexme,line) | 현재 상태(state), 현재 까지 저장된 lexeme, 라인수를 넘겨 받아서 해당하는 상태에 따른 오류 정보를 출력하는 함수 |
| State\_change(state,ch) | Transition\_table을 이용하여 현재 상태(state)와 input symbol(ch)을 입력받아 결과를 처리해주는 함수. |

**(2-1) def lexical(text)**

디테일한 코드는 제외하고 틀이 되는 내용은 다음과 같다. 현재 state 정보와, input symbol을 토대로 계속 transition을 수행하며, 현재 state에 해당하지 않는 input symbol이 들어왔을 때, Token 이름과 Lexeme 정보를 저장하며 수행한다.

|  |
| --- |
| **def lexical(text):**  **while True:**  **// EOF를 읽을 때 까지 무한 반복**  **ch = text[i] //input symbol 정보 저장**  **next\_state = state\_change(state,ch) //현재 state와 input symbol 정보를 토대로 next\_state 정보**  **if next\_state == 0 or next\_state == -1: // 0 : input\_symbol로 transition 안될때**  **// -1 : EOF를 읽었을 때**  **token = State\_table[state] // 현재 상태에 맞는 Token 정보를 얻음**  **# save result in token\_result array**  **token\_result.append({'token' : token ,'lexeme' : lexeme}) // token, lexeme 정보를 저장**  **state = 0 // state,lexeme을 초기화 하고 다시 읽기 시작**  **lexeme = ''**  **# EOF**  **if next\_state == -1: // EOF일때만 반복 종료**  **break**  **else:**  **state = next\_state // next\_state로 transition이 되었으면 lexeme 읽고**  **lexeme += ch // I 값을 증가시켜서 다음 input symbol 읽음**  **i += 1**  **return token\_result** |

**(2-2) report\_error(state,lexeme,line\_num)**

DFA Transition Table에 따른 오류코드 내용이다. [그림 3]과 [그림 4]에서 final state가 아닐 때 종료되면 오류를 출력하는 것으로 총 7개의 오류를 정의하였다.

|  |
| --- |
| **def report\_error(state,lexeme,line\_num) -> None:**  **if(state == 8):**  **raise Exception("Line[{}] Wrong value after ! \nLexeme : {} ".format(line\_num,lexeme))**  **if(state == 20):**  **raise Exception("Line[{}] Single quote is not Closed. \nLexeme : {} ".format(line\_num,lexeme))**  **if(state == 21):**  **raise Exception("Line[{}] Double quote is not Closed. \nLexeme : {} ".format(line\_num,lexeme))**  **if(state == 31):**  **raise Exception("Line[{}] Only 1 symbol is permitted in single quote. \nLexeme : {}".format(line\_num,lexeme))**  **if(state == 34):**  **raise Exception("Line[{}] Blank is not permitted in Single quote. \nLexeme : {} ".format(line\_num,lexeme))**  **if(state == 35):**  **raise Exception("Line[{}] Integer Start with 0 is not permitted. \nLexeme : {}".format(line\_num,lexeme))**  **if(state == 36):**  **raise Exception("Line[{}] Invalid Input. \nLexeme : {} ".format(line\_num,lexeme))** |

[표 1] Error Code

|  |  |
| --- | --- |
| State 8 | Lexical Specification에서 !=은 Compare Operator로 정의할 수 있지만 ! 문자가 단독으로 나타났을 때는 정의된 바가 없다. 이 부분은 오류로 처리 하였다. |
| State 20 | Single Quote(‘) 를 읽었지만 그 상태에서 EOF를 읽어서 프로그램이 종료된 경우이다. 싱글 쿼터가 닫히지 않았다는 오류 메시지를 출력한다. |
| State 21 | Double Quote(“) 를 읽었지만 그 상태에서 EOF를 읽어서 프로그램이 종료된 경우이다. 더블 쿼터가 닫히지 않았다는 오류 메시지를 출력한다. |
| State 31 | Single Character 정의에서 single Symbol만 가능하다고 명시하였다. 따라서 Single quote 안에 2개이상의 문자가 들어온다면 오류 메시지를 출력한다. |
| State 34 | Single Character 정의에서 single Symbol만 가능하다고 명시하였다. 따라서 Single quote 아무런 문자가 들어오지 않는 blank의 경우 오류로 처리하였다. 이때,Literal String의 경우 Double quote로 앞뒤가 감싸져만 있으면 된다고 제약사항이 되어 있기 때문에 이런 오류(31,34)가 없는 것을 볼 수 있다. |
| State 35 | Signed Integer에서 0으로 시작하는 숫자는 명시적으로 안된다고 제약사항이 되어 있기 때문에 이를 처리하기 위한 하나의 state를 더 만들었다. 만약 이 state를 만들지 않은 상태에서 0123을 입력한다면 0 토큰하나, 123 토큰하나가 따로따로 인식이 되기 때문에 필요한 작업이다. 해당 상태는 0을 입력받고 digit을 입력받으면 들어오는 상태로 [그림 4]의 dfa에 state 23부분에서 확인 가능하다. |
| State 36 | 맨 처음 State에서 +,-,\*, … Letter,Digit 등을 제외한 특수 문자들이 입력되었을 때 들어오는 State이다. 이는 Invalid Input으로 처리하였다. |

**6. Usage**

**(1) Requirements**

해당 코드를 작성하면서 파이썬 관련 모듈을 사용하지 않았기 때문에 설치해야 하는 모듈은 없다. 테스트는 파이썬 3.5.3 버전에서 테스트 하였으며 다른 버전을 사용해도 상관은 없다.

**(2) Usage**

1. 테스트할 코드를 **test\_code 폴더**에 만든다. 기본으로 오류 테스트용 파일 error1.txt - error7.txt 파일이 들어 있고 정상 테스트 코드 test.txt 파일이 들어있다.

2. python lex.py [filename] 와 같은 방식으로 실행한다.

**(3) Test**

다음 [그림 5]는 우분투에서 실행한 결과이다. 오류가 없는 파일의 경우 <input\_file\_name>\_output.txt 파일로 결과를 저장하며 파일에는 [그림 6]과 같이 토큰의 정보가 저장된다. 만약 input 파일에 문제가 있을 경우 그림[7]과 같이 오류에 해당하는 내용과, 몇번 째 라인에서 오류가 발생했는지에 대한 정보 및 lexeme 정보가 출력된다. 총 7개에 대한 오류를 정의 했기 때문에 각각 다른 오류가 나오고 있다.

정상 테스트 코드인 test.txt 파일에는 Lexical Specification에 명시된 모든 토큰들에 대한 정보를 다 담고 있으며, 특히 ‘-‘ 심볼의 경우 operation의 역할을 하는지, 음수의 역할을 하는지를 테스트 할 수 있도록 관련 테스트 코드도 포함되어 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

[그림 5 – 정상 파일 실행 화면]

**텍스트이(가) 표시된 사진

자동 생성된 설명**

[그림 6 – 토큰 출력 결과]

**텍스트이(가) 표시된 사진

자동 생성된 설명**

[그림 7 – 에러 파일 실행 결과]