Lexical Analyzer

7조 (이창민, 서정현)

목차

1. **토큰 정의 및 정규표현식**
2. **DFA Transition Table**
3. **구현 방식**

# **토큰 정의 및 정규표현식**

전체 입력을 표현하기 위해서 MERGED라는 하나의 식을 정의하였고, 입력받을 수 있는 토큰을 모두 포함하였다.

<MERGED> = <NUMBER> | <CHAR> | <BOOL> | <STRING> | <TYPE> | <WSPACE> | <ID> | <IF> | <ELSE> | <WHILE> | <CLASS> | <RETURN> | <OPER> | <ASSIGN> | <COMPARE> | <SEMICOLON> | <LBRACE> | <RBRACE> | <LPAREN> | <RPAREN> | <LBRACKET> | <RBRACKET> | <COMMA>

NUMBER는 정수를 표현하는 식이다. 식을 표현하기 위해 두 가지 식을 더 정의했는데, 1에서 9까지의 10진수를 표현하는 NONZERO와 NONZERO에서 0을 추가한 DIGIT이라는 식이다.

<NONZERO> = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<DIGIT> = 0 | <NONZERO>

<NUMBER> = 0 | (- | ε)<NONZERO><DIGIT>\*

의 형태로 정의하였다.

CHAR는 단일 chararcter를 인식하기 위한 식이다. 형태는

<CHAR> = ‘<LETTER>’

로 정의하였다. 시작과 끝에 작은따옴표를 사용하였고, 가운데에서 인식하는 LETTER는

<LETTER> = <DIGIT> | <ALPHABET> | ‘ ’

로 정의하였다. 이 식에서 ALPHABET는 영어 알파벳을 인식하는 식이고, ‘ ’는 스페이스바를 인식하는 표현인데 표현상 부득이하게 작은따옴표를 사용하여 ‘ ’로 표기했다. 실제로는 스페이스바 한 글자만을 인식한다.

BOOL은 참/거짓의 값을 나타내는 true와 false를 나타내는 식이다. 형태는

<BOOL> = true | false

와 같다.

STRING은 문자열을 나타내는 표현식이다. 형태는

<STRING> = “<LETTER>\*”이다.

CHAR를 표현할 때 사용한 LETTER를 재활용하여 kleene closure로 표기하였는데, STRING이 CHAR의 집합이기 때문이라고 판단했기 때문이다. 그리고 시작과 끝에 큰따옴표를 붙여 문자열이라는 것을 알 수 있도록 하였다.

TYPE은 변수의 자료형을 나타내는 표현식이다.

<TYPE> = int | char | boolean | String

으로 표현한다. Java에서 사용하는 자료형의 문법을 따라서 표기하였다.

WSPACE는 공백 문자들의 집합이다. 형태는

<WSPACE> = \n | \t | ‘ ‘

이다. 개행문자, 탭, 그리고 스페이스바를 인식한다.

ID는 변수와 함수의 이름을 나타내는 표현식이다. 형태는

<ID> = (<ALPHABET> | \_ )(<ALPHABET> | <DIGIT> | \_ )\*

로 정의하였다. 처음 문자는 숫자가 될 수 없기 때문에 앞에 ALPHABET과 \_만 인식하도록 하고, 그 다음부터는 몇 글자가 나올 지 알 수 없기 때문에 kleene closure로 표기하였다.

IF, ELSE, WHILE, CLASS, RETURN은 각각 예약어인 if, else, while, class, return만을 인식하는 식이다.

<IF> = if

<ELSE> = else

<WHILE> = while

<CLASS> = class

<RETURN> = return

OPER는 사칙연산자를 인식하는 표현식이다. 형태는

<OPER> = + | - | \* | /

이다. 과제에 주어진 네 가지의 연산만을 인식한다.

ASSIGN은 대입 연산자이다.

<ASSIGN> = =

으로 표현한다.

COMPARE는 비교 연산자를 인식하는 말이다.

<COMPARE> = ((= | !)=) | (>(= | ε)) | (<(= | ε))

의 형태로, >, <, !=, == 네 가지를 인식한다.

SEMICOLON과 COMMA는 각각 ;과 ,이다.

<SEMICOLON> = ;

<COMMA> = ,

LBRACE, RBRACE, LPAREN, RPAREN, LBRACKET, RBRACKET는 각각 [, ], (, ), {, }이다.

<LBRACE> = [

<RBRACE> = ]

<LPAREN> = (

<RPAREN> = )

<LBRACKET> = {

<RBRACKET> = }

# **DFA Transition Table**

DFA는 그림으로 표현하기 너무 복잡하였고 가독성도 좋지 않을 것이란 판단을 하여 Transition Table의 형태로 정의하였고, 결과물을 DFA\_Transition\_Table.xlsx 파일에 정리해놓았다. 코드 상에서는 파일 입출력에 더 편리한 형태로 바꾼 DFA\_Transition\_Table.txt 파일을 인식하여 동작하도록 구현하였다. xlsx 파일에서는 전이 함수가 정의되지 않는 곳, 즉 𝛷은 편집의 용이성을 위하여 비워놓았지만, txt 파일에서는 구현의 편의성을 위해 모두 -1로 정의하였다. DFA의 총 state 수는 85개이고, 입력 가능한 문자는 숫자 10개, 알파벳 대소문자 52개를 포함하여 총 83개를 사용하기 때문에 table은 85 \* 83 행렬로 표현하였다. 행에서 state를 표현하고, 열에서 입력 가능한 문자를 표현한다.

# **구현 방식**

## 파일 분할

1. [dfa\_transition\_table.txt]

DFA Transition Table을 저장하고 있는 파일이다.

1. [finish\_state\_return.txt]

Finish State에서 반환되는 Token의 Type Table을 저장하고 있는 파일이다

1. [token.h / token.cpp]

Token 클래스를 정의하는 헤더 파일과 구현 파일이다.

1. [character\_mapping.h / character\_mapping.cpp]

입력받은 Character를 Transition Table의 열 번호로 변환하는 함수의 정의와 구현이다.

1. [table\_init.h / table\_init.cpp]

위의 dfa\_transition\_table.txt와 fianl\_state\_return.txt파일을 읽는 함수의 정의와 구현이다.

1. [lexical\_analyzer.cpp]

main 함수를 포함하고 있는 파일로, 실질적인 Lexical analysis를 수행하는 함수들의 모음이다.

## Lexical Analysis 동작 원리

1. If) argc != 2: Error

분석하고자 하는 파일의 이름인 argv[1]에 해당하는 Input file의 이름을 Load한다. 만약 인자의 수가 맞지 않는다면 에러이므로 그대로 종료하고, 2인 경우에만 다음 과정으로 넘어간다.

1. If) File Read Fail: Error

프로그램에서는 총 세 개의 파일을 읽게 된다. argv[1] = Input file, dfa\_transition\_table.txt, 그리고 final\_state\_return.txt을 읽어들이는데, 하나라도 없을 경우 더 이상 진행할 수 없으므로 상황에 맞는 에러 메시지를 출력하고 종료된다. 파일을 읽는 데 이상이 없었다면, 문자 하나하나를 입력받아 Token을 인식하는 루프 단계로 넘어간다.

1. while loop

Input file에서 Character Read

If) Transition exist : Character push, State update

Else)

If) State가 Terminal: Token Recognize

Else) Error

while문을 사용하여 파일의 끝까지 한 문자씩 읽어들인다. 만약 읽은 문자가 lexical analyzer에서 취급하지 않는 문자일 경우, 입력 불가능한 문자라는 에러 메시지를 내고 프로그램은 종료된다. 아니라면 그 문자를 저장하고 문자에 맞는 state로 전이한다. transition rule이 없는 경우에 도착하면 현재 state가 terminal state인지를 final\_state\_return.txt에서 읽어들인 정보와 비교한다. 만약 terminal이 아니라면, 끝나면 안될 부분에서 끝난 것이므로 줄 번호와 에러 메시지, 그리고 지금까지 입력받은 내용을 출력하고 프로그램은 종료된다. 아니라면 지금까지 입력받은 문자열이 하나의 Token으로 분류될 수 있음을 나타내므로, state의 정보로 Token의 종류를 분류하여 새로운 Token으로 저장하게 된다.

1. Output file write

입력 파일의 끝까지 에러 없이 읽어들였다면, 저장한 토큰들을 출력 파일에 쓰게 된다. 출력이 무사히 끝났다면 lexical analysis가 끝났다는 메시지를 출력하고 프로그램은 종료된다.

## Token Recognition Issue

현재 State가 Final State일때 finish\_state\_return 배열에서 Token Type을 얻고, 지금까지 Push된 문자열을 Token Value로 얻는다.

이 때, ‘-’ 가 뜻하는 것이 마이너스 연산자일 수도 있고 정수 앞에 붙은 음수 부호일 수도 있다. 구현한 DFA Transition Table에 기반하여 Token을 인식하면 ‘-’ 뒤에 숫자가 오는 경우 반드시 음수로 분류하기 때문에 문제가 생길 수 있다. 따라서 이 경우 우선 NUMBER로 인식하여 진행한 후, 바로 직전에 분류한 Token Type이 NUMBER나 ID인 경우, 가장 앞의 문자를 사칙 연산자 중 빼기 기호인 <OPER, ->와 나머지 뒤의 숫자를 NUMBER로 분할하여 Token 배열에 저장한다.

WSPACE는 DFA Transition Table에 할당되어 있고, Token 인식 과정 중에서 인식하지만, 출력은 하지 않기 때문에 WSPACE 타입의 토큰을 받은 경우 Token 배열에 저장하지 않고, 무시하고 다음 과정을 진행한다.