# Database Project 1-1 Report

2013-11431 정현진

## 1. 프로젝트 구현

PRJ1-1 은 스펙에 명시된 문법을 JavaCC 문법에 맞춰 옮겨 적는 작업이 대부분이었기 때문에 크게 난해한 부분은 없었다. 구현하면서 신경 써야 했던 부분은 다음과 같다.

### 1-1. Token 의 순서

JavaCC 는 중복된 정의가 있는 token 을 선택할 때, 다음과 같이 두 가지 규칙을 따른다.

1. Input stream 에 가장 길게 매칭이 되는 토큰을 선택한다.
2. 같은 길이로 매칭이 될 경우, 먼저 정의된 토큰을 선택한다.

토큰을 정의하면서 규칙 2 번을 자주 고려해야 했는데, 다음과 같은 사항들에 규칙 2 번이 적용되었다.

1. Keyword 로 사용되는 토큰들과 〈LEGAL IDENTIFIER〉의 순서: 〈LEGAL IDENTIFIER〉는 Keyword 토큰들을 쓰면 안 되므로, Keyword 토큰들은 〈LEGAL IDENTIFIER〉보다 먼저 정의되어 있어야 한다.
2. 〈LEGAL IDENTIFIER〉와 〈ALPHABET〉의 순서: 〈LEGAL IDENTIFIER〉가 〈ALPHABET〉보다 먼저 정의되어 있어야 길이 1 의 문자가 입력될 때 해당 문자가 〈ALPHABET〉으로 정의되지 않는다.
3. 〈INT_VALUE〉와 〈DIGIT〉의 순서: 2 번과 마찬가지로 〈INT_VALUE〉가 〈DIGIT〉보다 먼저 정의되어 있어야 한다.
4. 한 글자 특수문자들과 〈NON_QUOTE_SPECIAL_CHARACTERS〉의 순서: 한 글자 특수문자들이 〈NON_QUOTE_SPECIAL_CHARACTERS〉로 선택되지 않기 위해서 한 글자 토큰들을 먼저 정의해 두어야 한다.

### 1-2. LOOKAHEAD

문법 명세 상 LOOKAHEAD 가 필요한 부분이 있었다. 우선

〈COLUMN IN TABLE〉 ::= [〈TABLE NAME〉 〈PERIOD〉] 〈COLUMN NAME〉

위와 같은 표현이 여러 번 중복되어서 사용되었기 때문에 재사용을 위해 〈COLUMN IN TABLE〉로 정의해주었다. 위의 표현에서 〈TABLE NAME〉과 〈COLUMN NAME〉은 정의상 모두 〈LEGAL_IDENTIFIER〉 토큰이므로 2 번의 LOOKAHEAD 가 사용되었다.

또한, 〈PREDICATE〉을 파싱할 때 최대 4 번의 LOOKAHEAD 가 필요한데, 이는 Parser 의 성능을 떨어트린다고 생각해 LOOKAHEAD 의 수를 줄이기 위해서 〈PREDICATE〉과 연관된 표현들의 문법을 재정의하였다.

위에 정의한 〈COLUMN IN TABLE〉을 사용해〈PREDICATE〉을 풀어 쓰면 다음과 같다.

〈PREDICATE〉 ::= 〈COLUMN IN TABLE〉 (〈COMP OP〉 〈COMP OPERAND〉 | 〈NULL OPERATION〉)

| 〈COMPARABLE VALUE〉 〈COMP OP〉 〈COMP OPERAND〉

여기서 〈PREDICATE〉의 OR 의 좌, 우의 표현들을 묶어서 재정의하여, 최종적으로 〈PREDICATE〉을 다음과 같이 표현하였다.

〈PREDICATE〉 ::= 〈COLUMN IN TABLE PREDICATE〉 | 〈COMPARABLE VALUE PREDICATE〉

〈COLUMN IN TABLE PREDICATE〉 ::= 〈COLUMN IN TABLE〉

((〈COMP OP〉 〈COMP OPERAND〉) | 〈NULL OPERATION〉)

〈COMPARABLE VALUE PREDICATE〉 ::= 〈COMPARABLE VALUE〉 〈COMP OP〉 〈COMP OPERAND〉

문법을 바꿈으로써 〈PREDICATE〉에서의 LOOKAHEAD 의 수를 4 회에서, 〈COLUMN IN TABLE〉에서만 발생하는 2 회로 줄일 수 있었다.

### 1-3. 기타

문법 명세에는 〈SPACE〉 토큰이 정의되어 있지만, 실제로 〈SPACE〉 토큰을 정의하면 〈SKIP〉과 충돌이 있다는 에러가 발생하므로, 〈SPACE〉는 따로 토큰을 정의하지 않고 " "로 사용하였다.

## 2. 가정한 것들

우선 KEYWORD 토큰에서 2 단어로 이루어진 토큰(ex: 〈CREATE TABLE〉)을 한 글자로 이루어진 2 개의 토큰으로 분리했다(ex: 〈CREATE〉 〈TABLE〉). 사용자가 첫 단어를 입력한 뒤 Enter 를 입력할 경우가 있다고 가정했다.

또한 틀린 토큰을 입력하고 enter 를 입력하면 바로 에러 메시지가 생성된다. Enter 를 입력한 순간 이전 줄의 입력 값은 바꿀 수가 없으므로 잘못된 값이 입력되었다면 유저가 앞으로 무엇을 입력하든 에러가 발생하게 된다. 따라서 빠르게 에러 처리되었다는 결과를 받는 것이 사용자 경험에 더 좋을 것 같다고 가정했다.

## 3. 구현하지 못한 내용

Parser 클래스에서 출력까지 담당하는 것은 Single Responsibility Principle 을 위반하므로 따로 출력을 담당하는 클래스를 구현하는 방식을 생각했지만, 향후 프로젝트의 스펙이 어떻게 주어질 지 모르기 때문에 우선 뼈대 코드와 동일한 방식을 사용해 출력하였다.

또한 prompt 의 출력에 대해, 결과 출력에는 prompt 가 출력되지 않는 것이 더 옳다고 생각했다. 하지만 Query Sequence 에서 여러 줄의 출력에 prompt 를 모두 출력시키지 않는 것은 현재 프로젝트 구조상 어렵다고 생각했다. 현재 한 줄에 하나의 query 가 입력된 경우 prompt 가 출력되지 않고, query sequence 의 경우 첫 번째 줄은 prompt 가 출력되지 않지만 두 번째 결과부터 prompt 가 출력되는 상태이다.

## 4. 느낀 점

컴파일러 수업을 들을 때 프로젝트를 하면서 파싱을 한 적이 있는데 JavaCC 를 이용하여 파싱을 해보니 구현이 간편하고 직관적이어서 좋았던 것 같다.