2015 fall 4190.310-001 Database

Project 1-1: SQL Parser

2013-11395 김희훈

1. Introduction

이번 프로젝트에서는 JavaCC를 이용하여 CREATE TABLE, DROP TABLE, DESC, SHOW TABLES, INSERT, DELETE, SELECT 구문을 파싱하는 간단한 SQL Parser를 구현하였다.

2. Implementation

문법이 EBNF에 가깝게 주어져있으므로 JavaCC의 문법 파일인 .jj로 쉽게 번역할 수 있다. 토큰들은 TOKEN 타입의 regular_expr_production으로, 나머지 non-terminals는 bnf_production로 구현하였다. 기계적으로 번역하면 되기 때문에 여기서는 특이사항만 다루도록 한다.

A. Left recursion

이 파서는 Top-down 방식이기 때문에 left recursion이 있는 문법은 구현할 수가 없다. 그런데 다음 두 non-terminal이 left recursion을 포함하고 있었다.

<BOOLEAN VALUE EXPRESSION> ::= <BOOLEAN TERM> | <BOOLEAN VALUE EXPRESSION> or <BOOLEAN TERM>

<BOOLEAN TERM> ::= <BOOLEAN FACTOR> | <BOOLEAN TERM> and <BOOLEAN FACTOR>

그래서 다음 형태로 바꾸어 left recursion을 제거해 주었다.

<BOOLEAN VALUE EXPRESSION> ::= <BOOLEAN TERM> [{ or <BOOLEAN TERM> }...]

<BOOLEAN TERM> ::= <BOOLEAN FACTOR> [{ and <BOOLEAN FACTOR> }...]

B. Lookahead

<PREDICATE> 같은 경우 <COMPARISON PREDICATE>와 <NULL PREDICATE> 모두 prefix로 <TABLE NAME> <PERIOD> <COLUMN NAME>이 가능해서 4 이상의 lookahead가 필요하다. 큰 lookahead는 파서의 성능에 좋지 않으므로 다음과 같이 전개 후 정리를 시도하였다.

<PREDICATE> ::= <COMPARISON PREDICATE> | <NULL PREDICATE>

::= (<COMPARABLE VALUE> <COMP OP> <COMP OPERAND>)

| ([<TABLE NAME> <PERIOD>] <COLUMN NAME> <COMP OP> <COMP OPERAND>)

| ([<TABLE NAME> <PERIOD>] <COLUMN NAME> <NULL OPERATION>)

여기서 새로운 non-terminal <TABLE AND COLUMN NAME>을 다음과 같이 정의하자.

<TABLE AND COLUMN NAME> ::= [<TABLE NAME> <PERIOD>] <COLUMN NAME>

그러면 <PREDICATE>는 최종적으로 다음과 같이 정리된다.

<PREDICATE> ::= (<COMPARABLE VALUE> <COMP OP> <COMP OPERAND>)

| (<TABLE AND COLUMN NAME> (<COMP OP> <COMP OPERAND> | <NULL OPERATION>))

이렇게 하면 <PREDICATE>에는 lookahead가 생기지 않는다. <TABLE AND COLUMN NAME>에는 2의 lookahead가 발생하는데, 이는 <TABLE NAME>과 <COLUMN NAME>이 문법상으로는 구분이 안되기 때문이며 더 이상 줄일 수 없다.

C. Token priority

가능한 토큰 후보가 여러 개일 때, **1. 길이가 긴 것**, **2. 문법 정의에서 앞에 오는 것** 순으로 우선순위가 결정된다. 그래서 다음 토큰들의 정의 순서에 주의해야 한다.

- keyword들을 먼저 정의하지 않으면 <LEGAL_IDENTIFIER>로 파싱될 수 있다.
- 괄호, 반점, 온점 등 한 글자 특수문자들을 먼저 정의하지 않으면 <NON_QUOTE_SPECIAL_CHARACTERS>로 파싱될 수 있다.
- <INT_VALUE>를 <DIGIT>보다 먼저 정의하지 않으면, char(0)과 같은 구문에서 0이 <DIGIT>으로 파싱된다.
- <LEGAL_IDENTIFIER>를 <ALPHABET>보다 먼저 정의하지 않으면, select a와 같은 구문에서 a가 <ALPHABET>으로 파싱된다.

D. 기타

- 공백은 SKIP으로 이미 정의되어 있어 <SPACE : " ">과 같은 토큰 생성은 불가능하다. 필요한 곳에 " " literal 그대로 넣어 주어야 한다.
- <NON_QUOTE_SPECIAL_CHARACTERS>의 경우 아스키 코드 순서를 이용하여 ["!", "#"-"&", "("-"/", ":"-"@", "["-"`", "{"-"~"]로 정의했다.

3. 컴파일 및 실행 방법

소스는 첨부파일 및 Github에서 구할 수 있다. SimpleDBMSGrammar.jj를 JavaCC로 컴파일하여 *.java를 생성한 후, javac로 컴파일하고 jar로 실행가능한 jar 파일을 만들 수 있다. 이를 java –jar 옵션으로 실행하면 된다.

4. 느낀 점

먼저, 구현 시간보다 이클립스 셋팅 시간이 더 오래 걸려서 다시는 이클립스를 이용하지 않겠다는 다짐을 하였다. Unix 환경인데도 line delimiter를 ₩r₩n로 붙이고, git 플러그인인 egit에는 파일 삭제만 있으면 커밋이 안되는 버그가 있으며, JavaCC 플러그인은 *.jar 파일 경로를 올바르게 찾지 못하였다.

컴파일러 수업에서 top-down 파서를 lex 같은 툴 없이 구현했었는데, JavaCC를 사용하니 구현도 간단하고 warning으로 대부분의 버그를 잡을 수 있었다.

테스트 입력을 수동으로 만들었는데, 추후 코드 커버리지 툴을 이용하여 더 강한 테스트를 구성하고자 한다.