프로젝트 1-1

2014-15703 송화평

1. 프로젝트 설명

본 프로젝트는 JavaCC를 이용하여 SQL의 기본적인 기능인 create table, drop table, insert into, … 등을 지원하는 DBMS를 구현하는 것이며, 이번 단계에서는 이 명령어들을 파싱하는 SQL 파서를 구현하였다. 주어진 Grammar Definition 내용을 충실히 따르도록 구현하였으며, 구현 중간에 문자열 사용을 최대한 지양하고, 명령어를 최대한 토큰화하여 받아들이도록 설계하였다.

1. JavaCC

본 프로젝트를 수행하기 위해 사용된 JavaCC는 Java를 사용하는 compiler-compiler로, 본 프로젝트의 SQL 파서 생성기로서 사용되어, Java로 작성된 파서 코드를 만들어 낸다. 이를 위해 <http://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf>를 참고하여 JavaCC를 과제 수행에 필요한 정도로 공부할 수 있었다. 또한 뒤에서 설명할 choice conflict 문제를 해결하기 위해 <https://www.cs.purdue.edu/homes/hosking/javacc/doc/lookahead.html>를 참고하였다.

1. 토큰

EXIT, INT, CREATE\_TABLE 등 이미 구현되어 있는 여러 토큰을 참고하여 SQL 파서 구현에 필요한 파서들을 추가하였다. 위에서 언급했듯 실제 파서 구현에서 문자열이 아닌 토큰만을 사용하였기 때문에, SQL 구문에 사용되는 문자열을 미리 토큰으로 정의해 놓았으며, INT\_VALUE, CHAR\_STRING, 또는 DATE\_VALUE 등의 자료형 또한 토큰으로 정의하였다.

1. Java 코드

파서의 뼈대가 되는 부분은, SQL 구문을 사용자의 입력으로 받아 처리하기 위해 SimpleDBMSParser라는 Java 클래스이다. 이 클래스의 main() 함수에서 무한루프를 돌려, 그 안에서 System.in과 연결된 parser 객체의 메서드인 parser.command()를 통해 파서가 실행된다. 이 메서드 실행 도중 잘못된 문법을 입력하는 등의 예외가 일어나면 예외처리 구문을 통해 에러메시지를 출력하고 빠져나오지만, 무한루프 안이기 때문에 다시 parser.command()를 호출하여 파서를 재구동한다.

1. 파서 코드

SQL 파서는 정규표현식으로 표현된 문자열을 포착하여 이를 각각의 토큰으로 나타낸다. 이 파서의 가장 상위에는 command()가 있어, System.in으로 들어온 사용자의 SQL 구문 입력을 잡아낸다. 이 입력을 받아내기 위해 queryList()에서 세미콜론으로 각 query를 구분하여 토큰으로 잡은 후, 그 결과를 출력한다. 잡아낼 query는 query()에 선언되어 있으며, 프로젝트의 설명에 제시된 대로 create table, drop table, desc, insert into, delete from, select, show tables의 7가지 기본 SQL 명령어를 포착해낸다.

한 문자열이나 여러 문자열을 단순히 묶어서 받아내는 등의 단순한 조건은 토큰으로서 정의하였으나, 그보다 복잡한 문자열을 받아내야 하는 표현들은 정규표현식으로 나타낸 구문으로서 정의하였다. 각 구문의 정의는 Grammar Definition에 제시된 내용을 이름까지 그대로 반영하여 알아보기 쉽도록 하였다.

또한 토큰을 정의할 때 SELECT, OR, AND와 같이 단순히 하나의 문자열로 정의된 토큰들이 가장 먼저 정의되고, CHAR\_STRING이나 LEGAL\_IDENTIFIER 등 더 포괄적이고 많은 문자열을 매칭해주는 토큰을 뒤에 정의하였다. 이는 SELECT, OR, AND, REFERENCES 등 SQL 구문의 일부로서 사용될 키워드가 LEGAL\_IDENTIFIER 등의 토큰으로 잡히기 전에 먼저 위에 정의된 토큰으로 잡힘으로써 테이블이나 컬럼의 이름으로 사용됨을 사전에 방지해준다.

본 프로젝트의 SQL 파서는 정규식을 사용하는 만큼, 사용자 입력을 가장 처음부터 하나하나 토큰과 대조해가며 받아낸다. 이는 choice conflict 문제를 발생시키는데, 하나의 입력을 받아 낼 수 있는 선택지가 두 개 이상일 때 일어난다. 이러한 문제가 발생할 수 있는 선택 분기에 LOOKAHEAD()를 사용하여, 입력을 하나하나 대조하지 않고 여러 개의 입력을 모아 한번에 대조하도록 하였다. LOOKAHEAD(4) comparisonPredicate() | nullPredicate()으로 작성한 구문을 예로 들면, table name, PERIOD, column name, COMP\_OP의 최대 4가지 정보(table name과 PERIOD를 사용하지 않는다면 2개)를 받아 놓고 대조하지 않으면 파서는 이를 comparisonPredicate() 분기를 선택해야 할 지, nullPredicate() 분기를 선택해야 할 지 애매하여 무차별한 상황으로 간주한다. 이는 nullPredicate() 구문도 첫 3개의 토큰이 table name, PERIOD, column name이기 때문이다. 이 경우에는 앞서 있는 comparisonPredicate()을 항상 선택하여, nullPredicate()으로 받아와야 할 문자열도 ComparisonPredicate()으로 받게 된다. 이와 같은 choice conflict를 방지하기 위하여 4개의 토큰을 미리 받아놓고 대조하도록 LOOKAHEAD()를 사용하였다.

1. 가정

사용자는 입력을 delete from student where student.name is not null and student.age < 19;와 같이 한 줄로 줄 수도 있지만, delete from student까지 입력한 후 줄바꿈을 하고 where 구문을 입력할 수도 있고, ~ student.name is까지 입력한 후 줄바꿈을 하고 not null을 입력할 수도 있다고 가정하였다. 즉, 토큰 단위로 어느 부분에서든 줄바꿈을 하는 등의 입력 상황이 벌어질 수 있다고 가정하였다. 이는 한 줄로 입력하는 것 보다 더 가혹한 가정이지만, 위에서 설명한 토큰화를 통해 최대한 많은 구문을 토큰으로 만듦으로써 해결할 수 있었다.

1. 느낀 점

프로젝트의 가장 첫 번째 단계인 파서 구현을 하면서, DBMS를 구현한다기보다는 컴파일러를 구현한다는 느낌이 들었다. Compiler-compiler를 사용하여 파서를 구현하는 작업이었기에 당연하지만, 컴파일러 수업을 듣지 않고 여러 입력 상황을 고려하여 입력을 correct하게 받아내는 작업이 쉽지는 않았다. 하지만 우여곡절 끝에 SQL 파서를 구현함으로써 SQL의 기본 명령어의 문법과 그 작동 방식을 익힐 수 있어 앞으로의 데이터베이스 수업과 프로젝트 수행에 큰 밑거름이 될 것이다.