Proj 1-1 (SQL Parser)

핵심 모듈과 알고리즘에 대한 설명

1. **Token**
   Parser에게 넘겨지는 단위이다. 하나의 문자나 그런 문자를 Token으로하는 것들을 nest한 조합도 Token으로 정했다. nested되는 깊이를 구체적으로 정하진 않았지만 의미상 하나의 Token으로 생각되는 수준까지는 Token으로 명시했다.

2. **Skip**
   TokenManager에게 읽히고 parser에게 넘겨지지 않고 버려지는 문자들이다.

3. **BNF production(함수)**
   javaCC script에서 parser에게 어떤 token을 parsing해야하는지 명시해 주는 역할을 하며 Java method 정의와 비슷하게 생겼다.

```
//example
void Start() :
{}
{
    <NUMBER>
    (<PLUS> <NUMBER>)*
}
```

Token으로 구분되기에는 복잡한 문법은 BNF production에 담아 parsing되게 하였다. BNF production의 경우 option에서 **DEBUG_PARSER** 값을 **ture**로 하게 되면 BNF production이 call되고 return되는 것을 추적할 수 있어서 디버깅에 용이했다. 반면, Token은 parser에 consume될 때만 알 수 있어서 만약 Token의 구조가 복잡해지면 어디서 matching이 안되는지 확인이 어려웠을 것이다.

> 단순히 Token 하나를 parsing하는 함수도 있는데 이는 의미상 추후에라도 문법이 변화할 것을 염두하고 작성한 것이다.

1. **SimpleDBMSParser class**
   이번 프로젝트 파서 이름 "SimpleDBMSParser"에 대한 클래스이다. 구현된 Parser가 시작되는 지점으로 여기에 query에 따른 메세지를 출력하는 메소드와 각 쿼리에 해당하는 메세지 넘버를 정의했다.

2. **lexical state**
   여러 종류의 set으로 Token과 Skip 등을 정의할 수 있게 해주는 장치다. 다음과 같이 Token과 Skip 정의부 바로 위에 state를 적어주면 해당 state에서만 아래 Token을 consume하고 Skip을 무시하게 된다.

```
<STATE1, DEFAULT>
TOKEN:
{
<SOME_TOKEN : "hello_world">
}
```

> 이 장치는 *NON_QOUTE_CHARACTER* Token을 받을 때 공백이 무시되지 않게 하기 위해서 사용하였다.

구현한 내용에 대한 간략한 설명

구현한 parser는 기본적으로 주어진 SQL grammar에 따라서 만들어 졌다.
grammar에 명시된 input만을 parsing하고 해당하는 SQL 명령을 받았다는 사실을 출력한다. grammar에 맞지 않는 input이 들어왔을 때는 `"syntax error"`를 출력한다.
함수(BNF production)와 Token도 SQL grammar에 나온 문법 요소들과 동일하게 naming 했으므로 그중에 추가적인 설명이 필요한 내용만을 담았다.

- **predicate()**

```
void predicate() :
{}
{
LOOKAHEAD(4)
comparisonPredicate()
|
nullPredicate()
}
```

이 함수는 조건을 보고 boolean 값을 내는 부분을 parsing한다.

> ex)
> instructor.budget > 10000 // comparisonPredicate()
> student.name is not null // nullPredicate()

이번 프로젝트에서 기본적으로 parser는 따로 option을 주지 않았으므로 **global LOOKAHEAD** 값은 1이다. 즉, *choice point*에서 하나의 Token만을 보고 선택을 해야하는 것이다.
한편 `comparisonPredicate()`와 `nullPredicate()` 모두 최대로 겹쳤을 때 아래와 같은 공통적인 sequence로 시작할 수 있다.

```
< LEGAL_IDENTIFIER > < PERIOD > < LEGAL_IDENTIFIER > ...
```

따라서 `comparisonPredicate()`를 결정하는 데 필요한 **local LOOKAHEAD** 값을 4를 주었다. 아무리 늦어도 4번째 토큰에서는 분명히 `<COMP_OP>` 토큰이 나와서 `comparisonPredicate()`을 선택하면 된다는 것을 확정할 수 있기 때문이다.
이때, **local LOOKAHEAD** 값은 `comparisonPredicate()` 앞에만 설정해주면 된다. 왜냐하면 4번째 토큰까지 보고도 확정하지 못했다면 `nullPredicate()`가 확실하기 때문이다.

- **selectedColumn(), comOperand(), nullPredicate()**
  세 함수 모두 다음과 같은 sequence를 parsing하는데 괄호안 *choice point*에서 `< LEGAL_IDENTIFIER >` 토큰 하나만 보고 `< PERIOD >` 토큰을 받을 것인지 아니면 괄호 밖으로 나올 것인지 결정할 수가 없다.

```
(< LEGAL_IDENTIFIER > < PERIOD >)? < LEGAL_IDENTIFIER > ...
```

따라서 첫번째 `< LEGAL_IDENTIFIER >` 토큰 앞에 **local LOOKAHEAD** 값을 2를 주어서 `< PERIOD >` 토큰까지 확인하고 괄호를 나갈 것인지 결정하도록 했다.

```
(
  (
    LOOKAHEAD(2)
    tableName() // parsing < LEGAL_IDENTIFIER >
    < PERIOD >
  )?
  columnName() // parsing < LEGAL_IDENTIFIER >
)
```

- **CHAR_STRING**
  Token중에 `< CHAR_STRING >`은 조금 특별하다. query의 중간중간에 `" "`, `"\r"`, `"\n"`가 들어올 수 있으므로, Token과 Token사이는 공백과 개행문자를 무시하도록 설정해야하는데 `< CHAR_STRING >`의 경우 그 안에 포

함되어 있는 `< NON_QUOTE_CHARACTER >` Token이 `< SPACE >`를 포함시키므로 `< NON_QUOTE_CHARACTER >` sequence를 받을 때에는 `< SPACE >`를 무시하지 않는 방법이 필요하다.
나는 아래와 같이 *lexical state*가 다른 Token을 따로 정의함으로서 문제를 해결했다.

```
< STRING >
TOKEN :
{
< SPACE : " " >
| < NON_QUOTE_CHARACTER : < DIGIT > | < ALPHABET >   | < NON_QUO
| < NON_QUOTE_SPECIAL_CHARACTERS : "!" | "@" | "#"  | "~" | "`"
"_" | "-" | "[" | "]" | "{" | "}" | "\\" | "|" | ";" | ":" | "<"
"," | "." | "/" | "?" | "+" | "=" >
}
```

`STRING` state에서 실제로 consume되는 Token은 없다. 위 토큰들과 관련하여 parser에 consume되는 Token은 `< CHAR_STRING >` 밖에 없다. 하지만 `< CHAR_STRING >`은 `DEFAULT` state에 정의되어 있으므로 consume될 수 있다.

가정한 것들

1. 입력으로 **들어올 수 있는 문자**는 다음과 같다.
    - Alphabet(a~Z)
    - number(0~9)
    - special characters on the keyboard
    - new line, carrige return, space

컴파일과 실행 방법

compile

*eclipse*에 **javacc plug-in**을 설치하게 되면 매번 저장할 때마다 자동으로 'jj'파일을 컴파일 해준다. 만약 자동으로 컴파일 되지 않는다고 해도 'jj'파일을 우클릭한 뒤 **compile with javaCC**를 선택하면 수동으로 컴파일 할 수 있다.

실행방법

실행도 마찬가지로 *eclipse*에서 **run**버튼을 눌러서 프로젝트를 바로 실행할 수 있다.
혹은, 실행파일로 만들어서 실행하고 싶다면 해당 프로젝트를 우클릭 하고 **Export -> Runnable JAR File**을 선택한 뒤 **launch configuration**에 해당 프로젝트를 넣고 **finish**를 누르게 되면 Jar파일을 생성한다.
생성된 Jar 파일을 실행하기 위해서는 **terminal**에서 해당 jar 파일 위치로 간 뒤에 아래 shell 명령어를 실행시키면 된다.

```
java -jar <file name>.jar
```

프로젝트를 하면서 느낀 점

Java도 익숙치 않지만 JavaCC라는 plug-in은 처음 들어봐서 처음에 사용하기가 어려웠다. 조교님께서 주신 예제 코드를 보고서 처음에 `select` 구문을 parsing 하는 코드를 짤 때 많은 conflict를 마주쳐야 했다. 또한 lexical state를 다루는 부분에서 많은 오류를 거쳤다. 하지만 과제 참고 문서에 있는 레퍼런스를 참고해보면서 `JavaCC`, `TokenManager`, `Parser`, `lexical state`, `LOOKAHEAD`, `BNF production`와 같은 개념을 알게되었고, 문제를 하나하나 해결해 나갈 수 있었다.

> 조교님이 regex 문법을 알면 좋다고 하셨는데 생각보다 전문적인 regex 문법을 사용하지는 않았다.