2024-2 Database (001) Project 1-1 Report

2023-12675 박지호

이번 프로젝트에서는 SQL query를 입력받아 parsing한 후, 해당 query의 이름을 출력하는 프로그램을 제작하였다. 이는 query를 입력받는 `receive_queries()`와, 입력받은 query를 처리하는 `process_query(query)` 두 함수를 통해 구성하였다. 이 중 `receive_queries()`는 여러 줄로 나뉘어 입력될 수 있는 query를 하나의 문자열로 연결한 다음, `;`을 기준으로 이 문자열을 나누어 query의 리스트를 반환한다.

`process_query(query)`는 query 하나를 처리하는 함수이다. 이때, 우선 `Lark.parse(text)`에 의해 query가 parsing되어 tree 형태로 정리된다. 이때, parsing 규칙은 grammar.lark 파일에 정의된 것을 따른다. 이후 `GetQueryName.transform(tree)`에 의해 tree로부터 해당 query의 이름이 얻어진다. 이 `process_query(query)` 함수는 이번 프로젝트에서는 이렇게 구한 query의 이름을 출력하는 기능만을 가지나, 이후 프로젝트가 확장됨에 따라 query의 내용을 실제로 수행하는 기능을 갖게 될 것이다. 또한, 이 함수는 query를 성공적으로 처리한 경우 `'SUCCESS'`, syntax error가 있어 처리에 실패한 경우 `'FAIL'`을 반환하며, query가 `exit;`인 경우 `'EXIT'`를 반환한다.

`GetQueryName` class는 Lark 패키지의 `Transformer` 클래스를 상속하는 클래스이다. `Transformer` 클래스에는 tree를 bottom-up으로 순회하며 node의 값을 변경하는 `transform(tree)` 메소드가 정의되어 있으며, 그 변경 규칙은 `GetQueryName`에 정의된 추가 메소드에 의해 결정된다. 이때, `transform(tree)` 호출 시 변경 대상 node를 만날 때마다 대응되는 메소드가 호출되기 때문에, 메소드 내에서 `print()`를 호출하는 것으로도 주어진 과제를 수행할 수 있었다. 하지만 `Transformer` 클래스는 tree의 내용을 변경하기 위한 클래스이기 때문에 이와 같이 메소드를 impure function으로 정의하여 side effect로 구현하는 방식은 `Transformer` 클래스의 의도된 사용 방식에 어긋난다는 느낌을 받았다. 따라서 `process_query(query)`에서 우선 query에 해당하는 node까지 이동한 다음, 이를 해당 query의 이름으로 변경하는 작업만 `GetQueryName` 클래스가 수행하는 방식을 채택하였다.

이 과정에서 parsing 결과 tree의 구조를 조금 더 면밀히 살펴보며 제공된 grammar.lark 파일이 여러 query를 한 번에 처리할 수 있게 구성되어 있다는 사실을 깨달을 수 있었다. 따라서 이를 활용하기 위해 `receive_queries()`에서 query를 나누는 과정을 생략하고, query sequence를 바로 parsing하는 방식을 고려하였다. 하지만 이 방식의 경우 query sequence 중 하나의 query라도 문제가 있다면 전체 sequence가 처리되지 않았는데, 프로젝트 명세에 따르면 이 경우 문제가 있는 query 이전까지는 정상적으로 처리되어야 한다. 이를 해결하기 위해 syntax error가 나는 경우에만 query를 분리하는 방식 등 여러 방법을 생각해 보았으나, 코드가 필요 이상으로 복잡해질 것 같아 결론적으로는 원래 구현을 채택하였다.

또한, `GetQueryName`의 메소드에서 query의 이름을 구하기 위해 string literal을 사용하는 방식 대신, tree의 구조로부터 query의 이름을 계산하는 방법 또한 생각해 보았다. 이때, parsing 결과 `SELECT`, `TABLE` 등 키워드는 `Token`이 되지만, 추가적인 정보를 가지는 parameter들은 `Tree`가 되기 때문에, query에 해당하는 node의 자식 중 `Token`이 아닌 것이 나올 때까지 `Token`의 `type`을 잇는 방식 또한 생각하였다. 하지만 이 경우 `DELETE`, `INSERT` query에 대해 각각 `DELETE FROM`, `INSERT INTO`가 출력되는 문제가 발생하였다. 또한, 다른 query에 대해서 이러한 접근이 통하는 것 역시 필연적인 결과는 아니라고 판단해 string literal을 반환하는 방식으로 회귀하였다.

떠올린 두 아이디어 모두 결과적으로 초기 구현을 채택하긴 했으나, 이들을 시도하는 과정에서 Lark의 tree를 다루어 본 경험은 프로젝트의 이후 단계를 진행할 때 도움이 될 것이라고 생각한다.

마지막으로, 최종 테스트 단계에서 이전 테스트 시에는 정상적으로 처리되던, `WHERE` clause에 `=`가 포함된 query가 parsing에 실패하는 문제가 발생하였다. 디버깅 결과, `EQUAL` token이 `COMP_OP`로

취급되지 않는 것이 확인되었다. 또한, `update_query`를 삭제하자 다시 정상적으로 인식되었기 때문에 이는 해당 symbol 정의 시에 `EQUAL` token을 사용한 것과 연관이 있는 것으로 추정된다. 결과적으로 `comparison_predicate`의 정의를 `comp_operand COMP_OP comp_operand | comp_operand EQUAL comp_operand`로 변경해 프로그램이 정상 작동하게 할 수는 있었으나, 이러한 문제가 발생한 이유는 밝혀내지 못했다.