Exercise sheet 5 2022-11-24

Due date: 2022-12-08 16:59

The goal of this exercise sheet, is to work with the standard library sum type `std::variant` and its helper function `std::visit`.

Exercise 1:

Complete the implementation of a simplified SQL SELECT clause verifier.

Parsing and verifying correct syntax is the bread and butter of compilers and interpreters. The rough sketch is the following: The program is handed a string of characters. The *lexer* (also *scanner* or *tokenizer*) converts this stream of characters, into a stream of, so called, *tokens*. A token represents valid symbols for the given grammar, for SQL these include keywords (*SELECT* or *FROM*), symbols (commas or semicolons) or identifiers (column or table names). Next, the parser takes the sequence of tokens created by the lexer, and converts that to an *abstract syntax tree*, which is just a tree representation of the tokens. Along side, it usually also performs a syntax check. This last part is what we want to concentrate in this exercise. We will not deal with the lexing part at all. We assume we are given a sequence of tokens. And all we want to compute is if that sequence is a valid sequence according to our grammar.

The grammar we will look at is a simplified version of the SQL. The complete grammar of SQL can be found at http://forcedotcom.github.io/phoenix/. As this is unnecessarily complex for our case, we will use a simplified version.

The grammar is the following:

1. The clause always starts with the `SELECT` statement
2. Then either:
   - all columns are select via `*`
   - a comma separated list of column names without a trailing comma
3. This is followed by the `FROM` statement
4. The `FROM` statement is followed by a table name
5. The table name is followed by at least one semicolon
6. A semicolon must be the last symbol

Any other sequence of tokens is invalid and should be rejected by our check.

For simple grammars like this, a nice approach is to implement an *finite state machine* (FSM). It will be feed a sequence of tokens, and if the machine is in an end state, the sequence of tokens is valid. For each state, the FSM needs to decide what is the next state according to current token.

Both tokens and such FSMs can be nicely modeled using a sum type like `std::variant`. Tokens are one of a closed set of types (e.g. keywords, identifiers, symbols). Each such type can be modeled using

a standalone type. For a FSM, a `std::variant` represents all the valid states of the state machine, and handling their transitions is then reduced to a form of pattern matching.

The exercise is two folds

- Add the missing types of tokens `token.h`.
- Add the missing states and the declaration of their `transition` functions in `validator.h`. Implement these functions, the `is_valid_sql_query` and the not implemented member functions of `SqlValidator` (`is_valid` and `handle`) in `validator.cpp`.

`is_valid_sql_query` returns true, iff the stream of tokens is a valid stream of tokens according to the above grammar. The `is_valid` member function of `SqlValidator`, return true, iff the FSM is in the `Valid` state. And the `handle` member function transitions the state machine from one state to the next, given the token.

The missing tokens are:

- From
- Comma
- Asterisks
- Semicolon

None of them carries any state information with them. The missing states are:

- `SelectStmt` — State if the FSM was feed a `token::Select`
- `AllColumns` — State reached from `SelectStmt`, iff an `Asterisks` token is encountered
- `NamedColumn` — State reached from `SelectStmt`, iff an `Identifier` token is encountered
- `MoreColumns` — State reached from `NamedColumn`, iff an `Comma` token is encountered
- `FromClause` — State reached from `NamedColumn` or `AllColumns`, iff an `From` token is encountered
- `TableName` — State reached from `FromClause`, iff an `Identifier` token is encountered

Additionally, you need to add the source files (`token.cpp` and `validator.cpp`) to the `hw05/CMakeLists.txt`.

Naming is important! The API of the `transition` function is given as an example for the *Start*, *Invalid* and *Valid* states. The other states need to follow the same style of signature.

You can of course add more parts of the real SQL grammar as you like :)

Correction

Remove the `#include "utils.h"` line from `validator.cpp`.