USCS 2010 – Applied Functional Programming

Database

Andres Löh

Difficulty: medium

This assignment deals with (a subset of) SQL, the *structural query language* for dealing with data in relational databases. As often, Wikipedia has a good summary on the history and the look of the language.

In current database systems, SQL is used not only to describe queries, but also to describe the format of tables and their contents. In this assignment, we implement a very simple database system entirely in Haskell, and develop a small SQL parser together with an interpreter so that we can describe data for that database and run queries. It is *not* an assignment that teaches you how to interface to a real database system.

For this task, you are supposed to parser combinators in order to write the parser. You can use any parser combinator library you like.

Structure

This assignment consists of several parts: You have to devise the abstract syntax of a fragment of the SQL grammar, to write a parser using the parser combinators. Furthermore, you have to implement the database system, and you have to write functions that interpret the SQL abstract syntax in terms of actions on the database system.

It is important to realize that there is not just one order in which you can approach this task. The representation of the database itself in terms of Haskell datatypes is given. Hence, you can start implementing database operations in Haskell and later write the parser and the semantics functions. Or, you can start with the parser, then write the semantic functions and implement the database in the end.

You can even decide to mix the approaches, and first implement some language constructs completely, and then move on to the next. Do whatever you feel works best, and if you get stuck somewhere, try if you can make progress in another area.

SQL grammar

The fragment of SQL we consider is given by the following grammar:

program	::=	*tablecommand** *query*
tablecommand	::=	*create* | *insert*
create	::=	`CREATE TABLE` *name* `(` *names*? `)`
insert	::=	`INSERT INTO` *name* *insertion*
insertion	::=	`VALUES` `(` *entries*? `)`
	|	*query*
query	::=	`SELECT` *names* `FROM` *names* *wherepart*?
wherepart	::=	`WHERE` *expression*
expression	::=	*expression* `AND` *expression*
	|	*expression* `OR` *expression*
	|	`NOT` *expression*
	|	`(` *expression* `)`
	|	*operand* *operator* *operand*
names	::=	*name* `,` *names* | *name*
entries	::=	*entry* `,` *entries* | *entry*
entry	::=	*string* | *number*
operand	::=	*name* | *entry*
operator	::=	`<` | `>` | `=`

Terminals are written in typewriter font, nonterminals in italics. Between any two symbols (terminals or nonterminal) in the above rules, arbitrary amounts of whitespace are allowed.

In addition, there are the following nonterminals for the lexical syntax – *no* whitespace is allowed anywhere in these rules:

name	::=	*letter* *alphanum**
string	::=	`'` *char** `'`
number	::=	*digit* *digit**
alphanum	::=	*letter* | *digit*
letter	::=	any letter
digit	::=	any digit
char	::=	any character except the single quote `'`

1. Define Haskell datatypes to describe the abstract syntax of SQL. First apply the general strategy to come up with a first version, but then reflect and consider if you cannot optimize a bit by introducing lists and occurrences of `Maybe`. Call the datatype for the starting nonterminal *program* `Program`.

Hint: if you want to split the syntax into smaller chunks, then omit *wherepart* and *expression* for the beginning.

2. The fact that whitespace can occur almost everywhere within an SQL statement makes the definition of parsers slightly trickier than usual. Nevertheless, we can define our parser in a single step, without the need for a separate lexical analyzer. The idea is to make most of the parsers consume additional spaces at the end of the input.

Define a parser

```
spaces :: Parser String
```

that greedily parses as much whitespace as possible (use the `isSpace` function from module `Data.Char`). You can use the declaration

```
import Data.Char
```

in the beginning of your program in order to import a module.

Then, use this parser to define parsers

```
keyword :: String -> Parser String
parens  :: Parser a -> Parser a
commas  :: Parser a -> Parser [a]
```

These are variants of `token`, `parenthesised` and `listOf` from `ParseLib`, but allow whitespace after each token.

Example:

```
parens (keyword "SELECT") "(  SELECT   ) "
```

should successfully parse the string `"SELECT"`, but

```
parens (keyword "SELECT") "(SEL ECT)"
```

and

```
parens (keyowrd "SELECT") "  (SELECT)"
```

should fail (i.e., drop the spaces in while error-correcting, because no spaces are allowed in the middle of a keyword, or in the beginning).

3 (medium)**.** The grammar, as given, is ambiguous (and left-recursive) for expressions. Remove the ambiguity and left recursion by assuming that `NOT`, `AND`, and `OR` have the usual priorities. Allow parentheses to be used to explicitly group expressions. Note that you do not have to reflect this in the abstract syntax (i.e., the Haskell datatypes), but you have to use the transformed grammar in order to define the parser.

4 (medium)**.** Define a parser

```
parseProgram :: Parser Program
```

that can parse an SQL program. Define all the other parsers for the other nonterminals. Make use of the combinators defined above, and define more if required. The guideline should always be that every parser consumes additional spaces in the end, but not at the beginning. Only the parser for the complete program `parseProgram` should also allow spaces at the beginning.

You can test your parser on some simple SQL programs, for instance

```
INSERT INTO foo VALUES (2,3,4)
SELECT name,location FROM addresses
```

Note that you can, for debugging purposes, also test individual parsers you define. In fact, try to verify every parser you define on a few examples.

Database implementation

We are going to implement our database directly in Haskell in a very straightforward way, with a focus on simplicity rather than efficiency. We model the entire database as a finite map. In order to get finite maps into your program, you should add the following import statements to the module header of your solution:

```
import qualified Data.Map as M
import Data.Map (Map (..))
```

Now, the database maps table names to tables:

```
type DB    = Map TName Table
type TName = String
```

A table contains the names of its columns, plus a list of rows that are the entries in the table.

```
data Table   = Table Columns [Row]
  deriving Show
type Columns = [Name]
type Name    = String
type Row     = [Entry]
```

As entries, we allow either strings or integers. We use the `deriving` construct to derive functions not only to show entries, but also to compare them.

```
data Entry = String String | Int Int
  deriving (Show, Eq, Ord)
```

(Note that we are introducing *constructors* `String` and `Int`, each parameterized with one argument of the indicated type.)

A simple table that associates course abbreviations and years with the name of the lecturers can be represented as follows:

```
exampleTable :: Table
exampleTable =
  Table
     ["course",          "year",   "lecturer"]
    [[String "INFOAFP",  Int 2007, String "Andres Loeh"],
     [String "INFOAFP",  Int 2006, String "Bastiaan Heeren"],
     [String "INFOFPLC", Int 2008, String "Andres Loeh"],
     [String "INFOSWE",  Int 2008, String "Jurriaan Hage"]]
```

Operations on the database can be modelled using a state monad:

```
type DBM = State DB
```

For this to work, you have to

```
import Control.Monad.State
```

This module contains the definition we discussed in the course:

```
newtype State s a = State (s -> (a, s))
```

and defines functions

```
runState :: State s a -> s -> (a, s)
put      :: s -> State s ()
get      :: State s s
```

as discussed. In addition, it also defines a function

```
modify :: (s -> s) -> State s ()
modify f =
  do
    s <- get
    put (f s)
```

that modifies the state according to the function given.

5. Define a function

```
printTable :: Table -> String
```

that turns a table into a readable string. Print the column headers, a line of horizontal dashes, and then all the rows. The column entries should be aligned, and sufficient room should be reserved for each column to hold the widest entry that occurs in that column.

Example:

```
putStrLn (printTable exampleTable)
```

should return something like the following output:

```
course     year lecturer
----------------------------------
'INFOAFP'  2007 'Andres Loeh'
'INFOAFP'  2006 'Bastiaan Heeren'
'INFOFPLC' 2008 'Andres Loeh'
'INFOSWE'  2008 'Jurriaan Hage'
```

6. Define a function

```
createTable :: TName -> Columns -> DBM ()
```

that, given the name of a table and column names, adds a new empty table with these column names.

7. Define a function

```
insertInto :: TName -> Row -> DBM ()
```

that inserts a new row into a table. The function should check that the number of entries in the row matches the number of columns in the table and do nothing if the numbers don't match.

8 (medium)**.** Define functions

```
select  :: Table -> RowProperty -> Table
project :: Table -> Columns -> Table
pair    :: Table -> Table -> Table
```

where

```
type RowProperty = Row -> Bool
```

The function `select` should keep only the rows from a table that have the given property.

The function `project` should keep only the columns from the table that are given, in that order. If columns are given that don't exist in the table, then those should be ignored.

The function `pair` should compute that cartesian product of two tables. The resulting table has all the columns of the first table, followed by all the columns from the second table. The rows are all combinations of rows from the first and rows from the second table. In particular, if the first table has c_1 columns and r_1 rows, and the second table has c_2 columns and r_2 rows, then the resulting table has $c_1 + c_2$ columns and $r_1 \cdot r_2$ rows.

9 (difficult)**.** Define a function

```
iExpression :: Expression -> [Name] -> RowProperty
```

that – assuming that `Expression` is the datatype used to represent nonterminal *expression* – interprets an expression as a row property. The additional argument of type `[Name]` indicates the column names of the table the property is operating on.

To understand how this works, let's look at the grammar for expressions. An expression is essentially a conjunction or disjunction of conditions, where each condition is some form of comparison between operands. Operands can either be constants or (column) names.

Assuming you have defined

```
data Operand = Entry Entry | Name Name
  deriving Show
```

you can define a helper function

```
iOperand :: Operand -> [Name] -> (Row -> Entry)
iOperand (Entry e) ns r = e
iOperand (Name n)  ns r = r !! (fromJust (findIndex (==n) ns))
```

that interprets an operand correctly as either a constant value or looking up the appropriately named column from the given row. The function `iOperand` uses `fromJust` from the `Data.Maybe` module and hence will fail if an "unknown" column name is mentioned, but we will leave this for a bonus exercise to fix.

Using `iOperand`, you can now define `iExpression`.

10 (medium). Define a function

```
iQuery :: Query -> DBM Table
```

that – assuming `Query` represents the nonterminal *query* – interprets a query as an operation on the database that returns a new table. The strategy is as follows: lookup all the tables mentioned in the `FROM` part and compute their product (using `pair`). Then use the columns of the product table to turn the `WHERE` part into a row property using `iExpression`. Use `select` to apply the row property to all the rows of the product table, and finally `project` to keep the columns specified in the `SELECT` part of the query.

11. Write a function

```
iTableCommand :: TableCommand -> DBM ()
```

that – assuming that `TableCommand` is the datatype for the nonterminal *tablecommand* – interprets an insert or create command as a database operation. Since an insert command can contain a query, you will have to use `iQuery` to define `iTableCommand`.

12. Write a function

```
iProgram :: Program -> DBM Table
```

that interprets an entire SQL program and returns the table delivered by the final query.

13. Write a

```
main :: IO ()
```

that uses the function

```
getArgs :: IO [String]
```

from the module `System.Environment` in order to check the command line arguments of the program. The argument should be interpreted as a filename, the file read using

```
readFile :: String -> IO String
```

and the contents of the file should be interpreted as a program using `parseProgram` and `iProgram`. After that, the program should print the resulting table using `printTable`.

As a final example, here is a possible input file:

```
CREATE TABLE courses (course, year, lecturer)
INSERT INTO courses VALUES ('INFOAFP',  2007, 'Andres Loeh')
INSERT INTO courses VALUES ('INFOAFP',  2006, 'Bastiaan Heeren')
INSERT INTO courses VALUES ('INFOFPLC', 2008, 'Andres Loeh')
INSERT INTO courses VALUES ('INFOSWE',  2008, 'Jurriaan Hage')
CREATE TABLE topics (subject, topic)
INSERT INTO topics VALUES ('INFOAFP', 'monad transformers')
INSERT INTO topics VALUES ('INFOFPLC', 'Haskell')
INSERT INTO topics VALUES ('INFOFPLC', 'parser combinators')
INSERT INTO topics VALUES ('INFOSWE', 'version management')
INSERT INTO topics VALUES ('INFOSWE', 'deployment')
SELECT lecturer, topic FROM courses, topics WHERE course = subject
```

and the output:

```
lecturer          topic
---------------------------------------
'Jurriaan Hage'   'deployment'
'Jurriaan Hage'   'version management'
'Andres Loeh'     'parser combinators'
'Andres Loeh'     'Haskell'
'Bastiaan Heeren' 'monad transformers'
'Andres Loeh'     'monad transformers'
```

Extra exercise for LC-only students

14 (medium). Define the algebra and fold function for your abstract syntax. Reexpress all the interpretation functions as a fold.

Bonus exercises

15 (bonus). Write a little interactive loop that reads SQL table commands or queries from the command line and interprets them. For table commands, just update the state. For queries, print the result. Update your `main` function to use the interactive loop if no filename is specified as a command line argument.

16 (bonus, medium). Add the possibility to save the current database to a file and read it from a file. This requires you to devise a good storage format for the database and be able to read that format again. An option is to use the standard `Read` and `Show` functions Haskell provides.

17 (bonus, variable). Make the program more robust by checking against all sorts of incorrect inputs (such as mentioning column names that don't exist) and giving meaningful errors in such a case.

18 (bonus, variable)**.** Add more features of SQL to the query language. An easy extension is to allow renaming of columns with `AS` in a `SELECT` statement, or to allow selecting all columns using `ALL`. Look for SQL descriptions to get more ideas for additional commands.