Cmpe 260 Haskell Project

Due Date: May 20, 2016 23:55 PM

The acts of the mind, wherein it exerts its power over simple ideas, are chiefly these three:

1. *Combining several simple ideas into one compound one, and thus all complex ideas are made.*
2. *The second is bringing two ideas, whether simple or complex, together, and setting them by one another so as to take a view of them at once, without uniting them into one, by which it gets all its ideas of relations.*
3. *The third is separating them from all other ideas that accompany them in their real existence: this is called abstraction, and thus all its general ideas are made.*

– John Locke, *An Essay Concerning Human Understanding* (1690)

1 Introduction

In this project, you're going to implement a *very simple* in-memory data store. Actual in-memory databases have a much complicated internal structure to provide efficiency, atomicity, consistency, durability; some of which you're going to see in Cmpe 321 *Introduction to Databases* course. In contrast to complicated architecture and design concerns of database management systems, this project is kept rather simple as an exercise on functional programming.

2 Tables

Your in-memory data store has only one fundamental data structure: tables. Although Haskell provides nice ways to construct objects and classes to create such data structures with encapsulation, it's out of scope of this project. Our fundamental data structure for tables is, as you all expect, *lists*. A table is actually a list in the form:

```haskell
tablename = [fields values...]
--Example:
students = [("name", "id", "gpa"), ("ali", "1", "3.2"), ("ayse", "2", "3.7")]

users = [("name", "surname", "id"), ("ali", "ay", "3"), ("ayse", "güneş", "2")]
```

Use tuples for the fields and values. The first tuple of the list contains the fields. The others contain list of values. Ensure that each entry in each tuple is of type String.

3 The functions

We indeed have a very simple table structure, but we need some functions to maintain them. Our *functions* are functions in mathematical sense, they ***do not*** create any side effect, change a value, print something to screen or depend on a global value. This is an important feature of functional programming and it indeed makes unit testing (in a real environment) much easier.

3.1 `table list`

The `table` function takes table name and a list and returns a Boolean value indicating whether it's argument is a table. This function shouldn't fail to run. Some examples:

```
> table [("name", "id", "gpa"), ("ali", "1", "3.2"), ("ayse", "2", "3.7")]
=> true
> table [("name", "id", "gpa")]
=> true
> table [("name", "id", "gpa"), ("ali", "1", "3.2"), ("ayse", "2", "3.7", "eggs")]
 This will give an error.
> table ["spam", ("ali", "1", "3.2"), ("ayse", "2", "3.7")]
 This will give an error.
> table [("name", "id", "gpa"), "spam", "eggs", ("ayse", "2", "3.7")]
This will give an error.
```

Here,

- First two cases are valid tables.
- The other cases give error.

3.2 `let tablename = createtable fields`

This function creates an empty table whose name is `table-name` and fields are given in `fields` tuple. Here is an example:

```
> let students = createtable ("name", "id", "gpa")
> students
=>[("name", "id", "gpa")]
```

Unless otherwise specified, assume that we have following list in examples:

```
students = [("name", "id", "gpa"), ("ali", "1", "3.2"), ("ayse", "2", "3.7")]
```

3.3 `get table row field`

Returns value of `field` in given `row`. Uses `table` to resolve field names. Assuming `field` is a field of `table`. An example:

```
> get students ("ali", "1", "3.2") "gpa"
=> "3.2"

> get students ("ali", "1", "3.2") "id"
=> "1"

> get students ("ali", "1", "3.2") "name"
=> "ali"
```

3.4 `alter table row fieldsvalues`

Alters values given in `fields-values` in given `row`. Uses `table` to resolve field names. Doesn't change original `row`, instead returns a new one. `fieldsvalues` is a list of field-value pairs. An example:

```
> alter students ("ali", "1", "3.2") [("name", "ahmet"), ("gpa", "3.3")]
=> ("ahmet", "1", "3.3")

> alter students ("ali", "1", "3.2") [("gpa", "3.3"), ("name", "ahmet")]
=>("ahmet", "1", "3.3")

> alter students ("ali", "1", "3.2") [("gpa", "3.3")]
=>("ali", "1", "3.3")

> alter students ("ali", "1", "3.2") [("name", "veli")]
=>("veli", "1", "3.2")

> alter students ("ali", "1", "3.2") [("id", "2")]
=>("ali", "2", "3.2")

> alter students ("ali", "1", "3.2") [("gpa", "3.3")]
=>("ali", "1", "3.3")

> alter students ("ali", "1", "3.2") [("name", "ahmet"), ("gpa", "3.3"), ("id", "3")]
=>("ahmet", "3", "3.3")

> alter students ("ali", "1", "3.2") [("id", "3"), ("name", "ahmet"), ("gpa", "3.3")]
=>("ahmet", "3", "3.3")
```

In the first example we changed values of `name` field to `'ahmet` and `gpa` field to `3.3`. Note that original students table is not changed.

The list of 2-tuples can have 1 2-tuples, 2 2-tuples or 3 2-tuples as shown in examples. Also the ordering can vary as shown in the examples.

3.5 `addrow table row`

Adds a row to a table. Adding a row to end of a table is inefficient (it takes O(N) time where N is number of rows), so we're adding the row *at the beginning*! after the field names. Here is an example:

```
--students list is not changed/updated
> addrow students ("asli", "3", "2.8")
=> [("name", "id", "gpa"), ("asli", "3", "2.8"), ("ali", "1", "3.2"), ("ayse", "2", "3.7")]
```

Note that original table *is not changed*, instead our function returned a new table. As stated above, our functions do not change.

3.6 `addrows table rows ...`

Adds many rows to a table. Kind of a bulk addition operator. An example:

```
--students list is not changed/updated
> addrows students [("asli", "3", "2.8"), ("ahmet", "4", "0.8")]
=> [("name", "id", "gpa"), ("asli", "3", "2.8"), ("ahmet", "4", "0.8"),
                        ("ali", "1", "3.2"), ("ayse", "2", "3.7")]
```

The order which you add the rows in this case may change, use the one that's suitable for you. This is a variadic function, that is it's arity is indefinite. Hint: you can use `foldr` or `foldl` for defining this function.

3.7 `deleterows table fieldsvalues`

This function deletes rows which has given field equal to a certain value. Here is an example:

```
--students list is not changed/updated
--Here, we deleted rows whose name field is equal to ali.
> deleterows students [("name", "ali")]
=> [("name", "id", "gpa"), ("ayse", "2", "3.7")]

--Here, we do not delete rows.
> deleterows students [("name", "ali"), ("gpa","3.7")]
=> [("name", "id", "gpa"), ("ali", "1", "3.2"), ("ayse", "2", "3.7")]

--Here, we deleted rows whose name field is equal to ali AND gpa field is equal to 3.2.
> deleterows students [("name", "ali"), ("gpa","3.2")]
=> [("name", "id", "gpa") ("ayse", "2", "3.7")]
```

For now assume,

```
students = [("name", "id", "gpa"), ("ali", "1", "3.2"), ("ali","3","3.2"), ("ayse", "2", "3.7")]
```

```
--students list is not changed/updated
--Here, we deleted ROWS whose name field is equal to ali
> deleterows students [("name", "ali")]
=> [("name", "id", "gpa"), ("ayse", "2", "3.7")]

--Here, we deleted ROWS whose name field is equal to ali AND gpa field is equal to 3.2.
> deleterows students [("name", "ali"), ("gpa","3.2")]
```

```
=> [("name", "id", "gpa"), ("ayse", "2", "3.7")]

--Here, we deleted ROWS whose name field is equal to ali AND id field is equal to 1.
> deleterows students [("name", "ali"), ("id","1")]
=> [("name", "id", "gpa"), ("ali", "3", "3.2"), ("ayse", "2", "3.7")]
```

3.8 `updaterows table fields-values`

This function changes the values of second field of rows whose field equals to the first specified field. It returns other rows as-is. An example:

```
--students list is not changed/updated
--Here, we updated values of gpa field of ALL rows whose name field equals to 'ali to 3.3
> updaterows students [("name", "ali"), ("gpa", "3.3")]
=> [("name", "id", "gpa"), ("ali", "1", "3.3"), ("ayse", "2", "3.7"))]
```

For now assume,

```
students = [("name", "id", "gpa"), ("ali", "1", "3.2"), ("ali","3","2.5"), ("ayse", "2", "3.7")]
```

```
--students list is not changed/updated
> updaterows students [("name", "ali"), ("gpa", "3.3")]
=> [("name", "id", "gpa"), ("ali", "1", "3.3"), ("ali","3","3.3"), ("ayse", "2", "3.7")]
```

3.9 `selectrows table fields field-value`

This function returns only fields given in `fields` for the given field value from table.The `fields` tuple can have 2 elements only. (No such cases will be tested:

selectrows students ("gpa") ("name", "ali"),

selectrows students ("gpa", "id", "name") ("name", "ali")).

The last parameter is a tuple with ONLY 2 strings where the first one is field name and second one field value. No such tuple ("name","ali","id","2") is to be taken for the last parameter.

An example:

```
--We created a table from GPA and ID number of students whose name is ali.
> selectrows students ("gpa", "id") ("name", "ali")
=> [("gpa", "id"), ("3.2", "1")]

--Note that tuples can be in any permutation.
> selectrows students ("id", "gpa") ("name", "ali")
=> [("id", "gpa"), ("1", "3.2")]
```

For now assume,

```
students = [("name", "id", "gpa"), ("ali", "1", "3.2"), ("ali","3","2.5"), ("ayse", "2", "3.7")]
```

```
--We created a table from GPA and ID number of students whose name is ali.
> selectrows students ("gpa", "id") ("name", "ali")
=> [("gpa", "id"), ("3.2", "1"), ("2.5", "3")]
```

3.10 `allrows table row`

This function returns all rows and some fields of a table. You can assume that only one field will be queried. (No such case: allrows students "gpa" "id").

An example:

```
> allrows students "gpa"
=> ["gpa", "3.2", "3.7"]
```

Here, we selected GPA of all students.

3.11 `sortby table field`

Sorts the table by a given field in ascending order. The sorting is to be done lexicographically. You can use built-in `sort` function to define this function. An example:

```
> sortby students "id"
=> [("name", "id", "gpa"), ("ali", "1", "3.2"), ("ayse", "2", "3.7")]
```

In case a field has more than one entries with the same value what will be done? For instance, a list has 4 entries with id 1. Then you will sort these with a precedence order. The default sorting order is the first field, then second field and then third field entry of the list. That corresponds to "name", "id" and "gpa" in an order in students list.

When you sort by "id", if more than one of the users have same "id" then sort them according to their "names". If again there are entries with same "name" then sort them according to their "gpa".

name	*id*	*gpa*
ali	1	2.0
ali	1	3.2
veli	1	1.0
veli	1	2.0

When you sort by "name", if more than one of the users have same "name" then sort them according to their "id". If again there are entries with same "id" then sort them according to their "gpa".

When you sort by "gpa", if more than one of the users have same "gpa" then sort them according to their "names". If again there are entries with same "name" then sort them according to their "id".

3.12 `distinct table`

Removes any duplicate rows in a table. Hint: you can sort the table first to find the duplicates easily and more efficiently (in $O(N \log N)$ rather than $O(N^2)$ time). An example:

Assume we have the following list.

```
students = [("name", "id", "gpa"), ("ali", "1", "3.3"), ("ayse", "2", "3.4"),
                                ("ali", "2", "3.3"), ("ali", "1", "3.3"))]
```

```
> distinct students
=> [("name", "id", "gpa"), ("ali", "1", "3.3"), ("ayse", "2", "3.4"), ("ali", "2", "3.3")]
```

4 Efficiency

Since you're going to write a kind of code that should be rather efficient in production environment, the code must be efficient, also there is a lot room for improvement (such as implementing tables with indexes implicitly), but they are not required for this project. You should still be sure that your methods run with a considerably large amount of data (say, about $N = 10^6$ rows for 10 tables) in a short amount of time (say, $t < 1s$), with simple comparison functions.

After all, nearly all of your functions, save `sort-by` and `distinct`, can have linear time complexity without using any complicated data structures such as trees, and some of them (such as `add-row`) should run in constant time in terms of number of rows.

5 Documentation and Clarity

Documenting the code is essential for developing in the long term and as you're not proficient in functional programming very much, you tend to solve the problems in rather obscure and/or unnatural ways. Thus, you should document every function in your project and you will be graded for documentation of your code in case your code becomes hard to understand.

You should add documentation to each function you defined (both the ones required by the project description ad the ones you've created as helpers) in the form below:

```
-- sqr x
--
-- Calculates square of x.
--
-- Examples:
-- > sqr 3
-- => 9
-- > sqr 5
-- => 25
sqr :: (Num a) => a -> a
sqr x = x*x
```

Besides documentation, it is also important to write code that is readable, so avoid small, clever, hard-to-comprehend hacks (unless they fit the very nature of functional programming, in which case you should explain the hack in documentation) and unnecessary complexity in your code. You are also graded for code clarity. When you cannot find a clearer way to handle something, explain how your code works in documentation, as it is important for us to grade your project and to be sure that you understood the concepts behind functional programming.

6 Submission

You are going to submit a file explicitly named as `project-YOUR_STUDENT_ID.hs` including the filename extension, all lowercase. The first line of the code must be in the form:

```
-- compiling: yes
-- complete: yes
-- name surname
```

The first line denotes whether your code is compiled correctly[1], the second line denotes whether you completed all of the project, which must be "no" (without quotes) if you're doing a partial submission. `name surname` must match with your name and surname and must be in *lowercase*. The lines must be lowercase.

Check that you're submitting the code as described above. Explicitly check that,

1. The filename is correct, including the extension.
2. You submitted the right file, you can check that by downloading your submission from Moodle.
3. You didn't submit a zip file or something else.
4. The filename is all in *lowercase*.

7 Prohibited Constructs

The following language constructs are ***explicitly prohibited***. You *will not get any points* if you use them:

1. Any language construct that uses for loop.
2. Any construct that causes any form of mutation[2] (impurity).

8 Some Tips on The Project

- You can check out PS slides.
- You can (and should) use higher-order functions like `map`, `filter`, `foldl` and `foldr`. You are also encouraged to use anonymous functions with help of `lambda`. The functions in project are designed to make use of them.

[1]here to be compiled correctly means that you get no errors when you press the Run button to run your code

[2]Mutation means creating any kind of change, either via changing value of a variable, a memory slot, or an element of a container; or via doing input/output.