Project PP 2022

General info

Project team

- Mihai-Valentin DUMITRU
- Vlad-Andrei BĂDOIU
- Teodora-Andreea ION
- Mihai UDUBAȘA
- Matei POPOVICI

- **Your Teaching Assistant**
  - You are highly **encouraged** to talk with your TA about the project and to ask questions during lab. You may also have sessions outside lab hours to check your code with them (either requested by you or by them).

Points

- You can get a maximum of **4.2 points** for the project, over the entire semester.
- You need a minimum of **2.5 points** from the project in order to qualify for the exam, along with a minimum of **3 points** accumulated during the semester (lab, project, lecture).
- Your final score will be computed at the end of the semester.
- There will be automated testing, but the score given by the checker is not necessarily the **final** score. There can be points removed by either:
  1. not complying to the task requirements (e.g. using library functions instead of implementing a required function), in which case the task is considered not completed and the points for that task are removed.
  2. not meeting a deadline (*see below*).

Task Sets & Deadlines:

- The project will be divided in **3 steps**, each with its own **task set**, submission assignment and number of points associated. The task sets will have progressively-increasing difficulty, and will reflect the lecture progress.

- Each task set has its own deadline; there is also a hard deadline for the project, **on the 19th of May**.
- You can submit solutions to each taskset until the **project deadline**.
- If you miss a task set deadline, you will lose **0.7 points** from that **taskset** (to a minimum of zero).
- To meet a task set deadline, you have to make a submission worth **at least 0.7 points** by that deadline; the grading team might manually deduce some points after the manual review, but unless this is done for extreme circumstances (e.g. hardcoding answers to automated tests), it **will not** count as missing the deadline.

Project description

We are working for the Haskell Health company that builds fitness trackers. For this project we want to build a basic data science analysis framework similar to pandas from Python.

The exercises are based on and tested with the tables that you can found in the file `Dataset.hs` :

- emails: contains the connection between the name of a person and their email
- eight_hours: contains the number of steps during 8 hours on a specific day
- physical_activity: contains the total steps, total distance, very active minutes, fairly active minutes, lightly active minutes of a person on a specific day
- sleep_min: contains the number of minutes slept in a week

When working on the project, we suggest parsing the table cells in a single function and passing appropriate data types to helper functions (e.g. for an eight_hours table, a helper function might work on a (String, [Float])).

During the whole project we will consider the following types: type CSV = String type Value = String type Row = [Value] type Table = [Row]

At the beginning of the `Tasks.hs` file you already have implemented the functions `split_by` , `read_csv` , `write_csv` that you can use to solve the tasks but you can also implement your own.

Important: Use the function `(printf “%.2f”)` when you have to transform a Float number to String.

Your module with the solution will have to be called `Tasks.hs` .

Checker

We offer you an automated checker and a test suite. We also encourage you to add your own tests.

To run the checker, simply run `runhaskell main.hs` in your shell (if you only wish to run certain tasksets, you can give it additional arguments: `runhaskell main.hs 1.4 2.3 2.4`).

The checker is written in Haskell and can be easily extended to add new testcases. Check `main.hs` for more.

Submission

The solution must be submitted to vmchecker (https://v2.vmchecker.grid.pub.ro/assignment/5/30/upload/). You need to upload a zip archive containing the file `Tasks.hs` as well as any other source files used. The archive should also contain a README file.

Taskset 1 (Introduction)

Deadline is on the **10th of April, 23:55**

Task 1 - 0.2p

We will first like to compute everyone's average number of steps per given day, based on the formula: $(10 + 11 + 12 + 13 + 14 + 15 + 16 + 17)/8$. You will have to write a function which takes an `eight_hours` table and returns a table with columns “Name”, “Average Number of Steps”.

Task 2 - 0.2p

Based on their number of steps, check:

- how many people have achieved their daily goal (have at least 1000 steps given the total number of steps on the 8 hours from the `eight_hours` table),
- what is the percentage of people who have achieved their goal and
- what is the average number of daily steps for all people.

For this you will have to implement 3 functions which take an `eight_hours` table and return an `Int` .

Task 3 - 0.2p

We want to find out which are the hardest hours to be active. We want to compute the average steps for each hour. The result will be a table with columns “H10”, “H11”, “H12”, “H13”, “H14”, “H15”, “H16”, “H17”, where the “Hx” column will represent the average number of steps for the `x` hour in the `eight_hours` table.

Task 4 - 0.2p

Similar to the previous task, we want to know how many minutes were spent being active based on their intensity. You will use the physical_activity table and compute a table with 3 rows “VeryActiveMinutes”, “FairlyActiveMinutes”, “LightlyActiveMinutes” and 3 columns (“range1”, “range2”, “range3”), each column with a range of number of minutes: first column will represent the range $[0-50)$, second $[50-100)$ and third $[100,500)$. A value in the table found at the intersection between a column and a row will be the number of people who have spent a number of minutes included in the range given by the column at the intensity given by the row.

Task 5 - 0.2p

At this point, we would like to see the overall leaderboard. You'll have to sort the people by their total number of steps (ascending). If two people have the same number of steps, they will be listed in alphabetical name order. The function for this will take a physical_activity table and return a table with columns “Name”, “Total steps”.

Hint: Use the `Ordering` *data type*

Task 6 - 0.2p

Then we will compute the difference between two parts of the day regarding the number of steps based on the table `eight_hours` . We will create a table with 4 columns: “Name”, “Average first 4h”, “Average last 4h”, “Difference”. This table will contain the people sorted ascending by the “Difference” column. If 2 people have the same difference, they will be listed in alphabetical name order.

Task 7 - 0.1p

Implement a function which applies a function to all values in a table.

```
vmap :: (Value -> Value) -> Table -> Table
```

An example use of this would be:

```
correct_table = vmap (\x -> if x == "" then "0" else x) table
```

Task 8 - 0.1p

Implement a function which applies a function to all entries (rows) in a table. The new column names are given. Additionally, you have to implement a function which takes a Row from `sleep_min` table and returns a Row with 2 values: email, total number of minutes slept that week.

```
rmap :: (Row -> Row) -> [String] -> Table -> Table
get_sleep_total :: Row -> Row
```

For the `get_sleep_total` function, print the number of minutes using `(printf “%.2f”)`.

Taskset 2

Deadline is on the **28th of April, 23:55**

Task 1 - 0.2p

Write a function which takes a column name and a Table and returns the Table sorted by that column (if multiple entries have the same values, then it is sorted by the first column). Sorting is ascending for columns with numeric values and lexicographical for strings. If the value is missing (`“”`), then it is considered before all values (e.g. `“” < “0”`).

```
tsort :: String -> Table -> Table
```

Task 2 - 0.1p

Implement a function which takes Tables t1 and t2 and adds all rows from t2 at the end of t1, if column names coincide. If columns names are not the same, t1 remains unchanged.

```
vunion :: Table -> Table -> Table
```

Task 3 - 0.2p

Implement a function which takes Tables t1 and t2 and extends each row of t1 with a row of t2 (simple horizontal union of 2 tables). If one of the tables has more lines than the other, the shorter table is padded with rows containing only empty strings.

```
hunion :: Table -> Table -> Table
```

Task 4 - 0.3p

Implement table join with respect to a key (a column name). If an entry from table 1 has the same value for the key as an entry from table 2, their contents must be merged. If there are other columns with the same name in both tables, then the value from t2 overrides the value from t1, unless it is null (`“”`).

```
tjoin :: String -> Table -> Table -> Table
```

Task 5 - 0.2p

Write the implementation for the cartesian product of 2 tables. The new column names are given. To generate the entries in the resulting table you have to apply the given operation on each entry in t1 with each entry in t2.

```
cartesian :: (Row -> Row -> Row) -> [String] -> Table -> Table -> Table
```

Task 6 - 0.2p

Extract from a table those columns specified by name in the first argument.

```
projection :: [String] -> Table -> Table
```

Task 7 - 0.2p

Filter rows based on a given condition applied to a specified column.

```
filterTable :: (Value -> Bool) -> String -> Table -> Table
```

Task Set 3 (Query Language, Graphs and Typo Correction)

Deadline is on the **12th of May, 23:55 Note**: the chekcer will not run until you have the implementation of the first tasks due to the missing definition of eval.

For the last part of our work at our data science framework at Haskell Health we will need two important functionalities: a query language and graph queries. This will make enable the user of our framework to quickly develop a product based on our framework. Moreover, to showcase the power of our framework to the marketing team, we will do typo corrections on the data using a distance function.

Query Language overview

The objective of this task set is to:

- allow programmers to **combine** table processing functions in a flexible way
- create a **separation** between what functions **do** and how they are implemented. This is helpful for a number of reasons:
    - **integration** of new table functionality later on
    - continuous code **upgrade** (new algorithms for different table processing functions can be implemented without altering the rest of the project)
    - debugging and testing

You will implement a **query language** which **represents** a variety of table transformations, some of which you have already implemented as table functions. A `Query` is a **sequence** (in some cases, a **combination**) of such transformations. You will also implement an evaluation function which performs a `Query` on a given table.

The query language to be implemented is shown below:

```
data Query =
    FromTable Table
    | AsList String Query
    | Sort String Query
    | ValueMap (Value -> Value) Query
    | RowMap (Row -> Row) [String] Query
    | VUnion Query Query
    | HUnion Query Query
    | TableJoin String Query Query
    | Cartesian (Row -> Row -> Row) [String] Query Query
    | Projection [String] Query
    | forall a. FEval a => Filter (FilterCondition a) Query
    | Graph EdgeOp Query

-- where EdgeOp is defined:
type EdgeOp = Row -> Row -> Maybe Value
```

Don't worry about Graph or Filter queries yet.

Query Evaluation

While most queries take a `Table` and return a transformed `Table`, there are some queries which evaluate to a list of `String`. Thus we define the type: `QResult` which describes any of these two possible query result types:

```
data QResult = Table Table | List [String]
```

In the skeleton we enroll `QResult` in class `Show`.

Task 3.1.: In order to ensure separation between queries and their evaluation we define class `Eval`, which offers function `eval`. Your job is to enroll `Query` in this class.

```
class Eval a where
    eval :: a -> QResult

instance Eval Query where
    ...
```

We explain below how each data constructor from `Query` should be evaluated:

1. `FromTable table`: returns the table it received as a parameter. Basically doesn't do anything. We do this such that we can pass tables as queries.
2. `AsList colname query`: returns values from column `colname` as a list.
3. `Sort colname query`: sorts table by column `colname`.
4. `ValueMap op query`: maps all values from table, using `op`.
5. `RowMap op colnames query`: maps all rows from table, using `op`.
6. `VUnion query1 query2`: vertical union of the 2 tables obtained through the evaluations of `query1` and `query2`.
7. `HUnion query1 query2`: horizontal union of the 2 tables.
8. `TableJoin colname query1 query2`: table join with respect to column `colname`.
9. `Cartesian op colnames query1 query2`: cartesian product.

10. `Projection colnames query` : extract specified columns from table.

You can look into the reference file to better understand how these functions work.

Filters & filter conditions

You may have noticed that filter query is commented-out. You can **uncomment** it at this stage. Filtering will receive a special treatment. Because filter conditions are usually complex, instead of performing successive filter queries it is better to build complex query conditions. For this reason, we define type `FilterCondition a` , illustrated below:

```
data FilterCondition a =
    Eq String a |
    Lt String a |
    Gt String a |
    In String [a] |
    FNot (FilterCondition a) |
    FieldEq String String
```

Remark: the type `FilterCondition a` is **polymorphic** because such conditions may be expressed over (in this homework) two types:

- `Float` and
- `String`

We briefly explain what each condition expresses:

1. `Eq colname ref` : checks if value from column `colname` is equal to `ref` .
2. `Lt colname ref` : checks if value from column `colname` is less than `ref` .
3. `Gt colname ref` : checks if value from column `colname` is greater than `ref` .
4. `In colname list` : checks if value from column `colname` is in list.
5. `FNot cond` : negates condition.
6. `FieldEq colname1 colname2` : checks if values from columns `colname1` and `colname2` are equal.

FilterCondition Evaluation

A `FilterCondition` must evaluate to an actual filtering function, which has type:

```
type FilterOp = Row -> Bool
```

Since such filtering functions work differently for `FilterCondition Float` and `FilterCondition String` , we need a class `FEval` which contains function `feval` . The latter is used to evaluate a `FilterCondition a` to a function of type `FilterOp` . In order to do so, `feval` needs to have information about column names (the table head), hence it's type is shown below.

```
class FEval a where
    feval :: [String] -> (FilterCondition a) -> FilterOp
```

Task 3.2.: Your task is to write the instances for `(FEval Float)` and `(FEval String)` .

```
instance FEval Float where
    ...
instance FEval String where
    ...
```

Task 3.3.: Now you can write the evaluation for the data constructor `Filter query` (see function **eval** from the previous section).

Graph queries

A **graph** is a special kind of table which has precisely the following column names: `[“From”, “To”, “Value”]`. Each row defines a **weighted edge** between node `From` and node `To`.

The query `Graph edgeop query` : creates such a table starting from the result of the evaluation of `query`. Suppose the query evaluates to a table **T**.

- The nodes are the **rows** in table **T**.
- The weight of an edge between 2 nodes is given by `edgeop`, which returns a `Maybe Value`. If `edgeop row1 row2` returns `Nothing`, then we don't have an edge between those 2 nodes. If it returns `Just val` then we have an edge between `row1` and `row2` of weight `val`.
- In the resulting table, each row describes an edge between node_i and node_j and will have the values:
    - “From” = first column from node_i
    - “To” = first column from node_j
    - “Value” = edgeop node_i node_j

The edge *node_i-node_j* is the same as *node_j-node_i*, so it should only appear once (graphs are unoriented). “From” value should be lexicographically before “To”.

Example: Suppose **T** is the table shown below:

Name	HoursSlept	Category
Mihai	9	321
Andrei	8	322
Stefan	10	321
Ana	9	322

If we would like to build a graph that connects all users in the same category (e.g. 321 is sports people), then:

```
edgeop [_,_,z] [_,_,c]
   | z == c = Just c
   | otherwise = Nothing
```

and the resulting graph will be:

From	To	Value
Mihai	Stefan	321

Ana	Andrei	322

If we would like to build a graph that connects users with hours of sleep equal or with a difference of **at least** a point, then:

```
edgeop [_,x,_] [_,y,_]
   | abs $ (read x :: Int) - (read y :: Int) <= 1 = Just "similar"
   | otherwise = Nothing
```

and the resulting graph is:

From	To	Value
Andrei	Mihai	similar
Mihai	Stefan	similar
Ana	Mihai	similar
Ana	Andrei	similar
Ana	Stefan	similar

Task 3.4.: Implement `eval` for Graph queries.

Note The result should be in alphabetical order between the two ends of the interval.

Similarities graph, using queries

We want to check the similarities between users' hours slept.

- For that, we want to obtain a graph where “From” and “To” are users' names and “Value” is the distance between the 2 users' hours slept.
- We define the distance between user1 and user2 as “the sum of intervals where they both slept a equal amount” (e.g. same value for “10” in `eight_hours`). Keep only the rows with `distance >= 5` .
- The edges in the resulting graph (the rows in the resulting table) should be sorted by the “Value” column. If email is missing, don't include that entry.

Task 3.5.: Your task is to write `similarities_query` as a **sequence of queries**, that once evaluated results in the graph described above.

The input table is `eight_hours` .

Note: `similarities_query` is a Query. The checker applies `eval` on it.

TL;DR Tasks

1. Enroll `Query` in class `Eval` (without `Filter` or `Graph`). **0.3p**
2. Enroll `FilterCondition` in class `FEval` and implement `eval` for `Filter` query. **0.2p**
3. Implement `eval` for `Graph` query. **0.2p**
4. Extract similarity graph. **0.3p**

For the final tasks of this project, we want to merge the information about users' stats from the 3 tables (8h of steps, minutes sleep week, physical activity in a day). The issue with joining these 3 tables is that table `Minutes_slept_week` is mapped using users' emails, whereas the other 2 tables are mapped using users' names.

Fortunately, we have another table, email_map, that maps users' names to their emails. The **bad news** is that this table contains **typos**.

Your job is to fix the typos in `email_map` table, generate the correct table and then use that to join the information about user's stats.

3.6 Typos (0.4p)

Column `Name` from table `email_map` contains typos. Example: *Zane Kayle* is misspelled as *Zan Kayle* or *Amelia Caden* is misspelled as *Amelia Camdden*. Fortunately, the name *Zane Kayle* contains only a few misspelled letters and it is guaranteed to appear in the correct form, in the name column of the `8h sleep` or `physical_activity_in_a_day` tables. Hence, we can use the latter names as **reference** to correct the typos.

The goal is to write function `correct_table`, which receives the name of the column containing typos, the table containing typos, a reference table (a table where the values from that column are correct, but not necessarily in the same order) and returns the first table where the typos have been fixed. All parameters are strings in CSV-format.

```
correct_table :: String -> Table -> Table -> Table
-- this will be tested using: correct_table "Nume" emails physical_activity
```

So in order to fix the typos, you will use a **reference table**, where the specified column has correct values.

Recommended steps

- extract the necessary column from the table with typos and the reference table (let's call these *T* and *Ref*);
- filter out only the values from T and Ref which don't have a perfect match in the other table - these are the problematic entries (this will help improve time performance);
- calculate the distance between each value from T and each value from Ref (distance = how similar the 2 strings are - *you decide how to formally define this distance*);
- for every value from T, its correct form is the value from Ref with the shortest distance to it;
- lastly, restore the original table, replacing the incorrect values from T with the correct values from Ref.

Note

- **Your implementation must be generic! It can't depend on the table structure or rows order. You can't assume that the rows in the the faulty table and in the reference table have the same order. Similarly, you can't choose a distance function that only works for these tables.**
- The steps above are only *recommended*. If you find another implementation that works and respects the first condition (generic implementation), that's great!
- Also recommended is that you use your previously implemented Query Language, but it's not a restriction. You might find some queries really helpful, such as Cartesian, Projection or Filter.