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 15th 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
```