CS337 Project 4
Database Querying in Haskell

TA in charge: Lara Schmidt (lschmidt@cs.utexas.edu)
Due: Friday, April 19, 11:59 pm

1 Overview

Lists are a common data structure in computer science. Indeed, it is unusual for a program not to use some list of data - the common array is one particular implementation of a list. Sorting is an important operation on lists. Lists are usually maintained as ordered lists for efficiency reasons. For this project, you will implement a number of functions over lists, including sorting operations.

Implementing list operations recursively is often more straightforward. The goal of this project is for you to become familiar with recursion in the context of list manipulation and sorting in Haskell.

For this project, you will implement functions to return information about a database, which consists of a list of tuples.

2 Description

The database stores employee information and it is required to develop a number of functions. Each entry in the database has four fields: *employee*, *spouse*, *salary*, and *manager*. The *employee* field is a string which is the name of the employee, *spouse* is the name of his/her spouse—henceforth, “his/her” will be abbreviated to “its” and “he/she” will be “it”—, *salary* is the employee’s annual salary and *manager* is the name of the employee’s manager. Assume that the database contains all the records of a hierarchical (tree-structure) organization in which every employee’s spouse is also an employee. Each manager is also an employee except *root*, who is the manager of all highest level managers. There is no record for *root* in the database. An employee whose manager is *root* will have the string “Root” in its manager field.

A manager of an employee is also called its *direct* manager; a *super* manager is either a direct manager or a super manager of a direct manager; thus, *root* is every employee’s super manager. It is a transitive relation.

Write functions for each of the following tasks below. Use exactly the same names for the functions as listed below (a starting project stub is provided with type signatures. You should use this to assure proper automatic grading of your assignment). In the following type expressions, DB is the type of the database, a list of 4-tuples as described above. You will find it useful to define a number of auxiliary functions, which you can use in the other functions. One such function could be *salary*, which given a name as an argument returns the corresponding salary.

1. Call an employee *overpaid* if it has a manager and its salary exceeds its manager's. It is *grossly overpaid* if it has at least one super manager and its salary exceeds the salaries of **all** its super managers. Implement functions listing all overpaid and grossly overpaid employees in a database. Assume that *root's* salary is 100,000.

```
overpaid :: DB -> [String]
grossly_overpaid :: DB -> [String]
```

2. List all employees who directly manage their spouses; do the same for super management.

```
spouse_manager :: DB -> [String]
spouse_manager_super :: DB -> [String]
```

3. List all managers who are super managers of both an employee and its spouse. Do **not** include root in this list.

```
super_manager :: DB -> [String]
```

4. Are there employees *e* and *f* such that *e's* spouse is *f's* manager and *f's* spouse is *e's* manager? The output is the list of **all** such *(e,f)* pairs (If x and y satisfy the above conditions, both (x,y) and (y,x) should be present in the resulting output).

```
nepotism :: DB -> [(String, String)]
```

5. Find the family that makes the most money. The output is a list of pair of strings with the names of both spouses. The reason a list is returned is that more than one family may share the maximum combined income. However, unlike problem 4, this function should not return the same family twice (either (x,y) or (y,x) appears, but not both).

```
rich :: DB -> [(String, String)]
```

6. Define the *rank* of a manager as the number of employees it manages. Define the *worth* of a manager as its salary divided by its rank: (salary/rank). Create three lists in which you list all managers (excluding root) in decreasing order of their salaries, ranks, and worth. Note that employees that aren't managers are not included in any of these results (particularly worth, since a divide by zero error would result). You will need to convert salary and rank to type Float in order to use the (/) operator. The `fromInteger` function will convert an Integer to any type (the correct type will be inferred). Strangely

enough, you may also need the `toInteger` function to first convert the type Int to type Integer (an annoying aspect of Haskell’s strong typing).

```
sorted_salaries :: DB -> [String]
sorted_rank :: DB -> [String]
sorted_worth :: DB -> [String]
```

7. The database is in *normal form* if the manager of *x* appears as an employee before *x* in the list. Write a function to convert a database to its normal form. Note that *root* should **not** appear in the normalized database (though conceptually, it’s position would be at the front).

```
normalize :: DB -> DB
```

3 Implementing and Testing

Remember that *root* is not part of the database. Employees whose manager is *root* will have the string “Root” in the manager part of the tuple.

Your program should implement the database as a list. A sample definition of the types involved is shown below. Do not attempt to use any non-list-based implementation of the DB (these type synonyms are provided in the project4.hs stub file).

```
type Employee = String
type Spouse = String
type Salary = Integer
type Manager = String
type Record = (Employee, Spouse, Salary, Manager)
type DB = [Record]
```

Your **README** should include the output generated on the following databases (cut/paste of command and result from program). These databases are available within the project4.hs stub file. Take special care to ensure that your code works on the empty database.

```
database0 = [(“Lana Turner”, “Buster Keaton”, 80000,
“Virginia Dare”), ("Ted Hughes", "Edna Millay", 70000,
"Virginia Dare"), ("Virginia Dare", "Laurence Sterne",
100000, "Edna Millay"), (“Buster Keaton”, “Lana Turner”,
80000, “Ingrid Joyce”), ("James Joyce", "Ingrid Joyce",
60000, "Root"), (“Vanessa Redgrave”, “Michael Readgrave”,
110000, “James Joyce”), (“Michael Redgrave”, “Vanessa
Redgrave”, 40000, “Vanessa Redgrave”), ("Edna Millay", "Ted
Hughes", 70000, "Root"), ("Laurence Sterne", "Virginia
Dare", 60000, "James Joyce"), (“Ingrid Joyce”, “James
Joyce”, 60000, “Virginia Dare”)]
database1 = []
database2 = [(“Carol”, “Eric”, 200000, “Bob”), (“Fran”,
“Dan”, 200000, “Eric”), (“Bob”, “Alex”, 100000, “Alex”),
(“Dan”, “Fran”, 150000, “Carol”), (“Alex”, “Bob”, 100000,
“Root”), (“Eric”, “Carol”, 300000, “Dan”)]
database3 = [(“Carol”, “Eric”, 200000, “Root”), (“Fran”,
“Dan”, 200000, “Root”), (“Bob”, “Alex”, 100000, “Alex”),
(“Dan”, “Fran”, 150000, “Root”), (“Alex”, “Bob”, 100000,
“Root”), (“Eric”, “Carol”, 300000, “Root”)]
```

4 Turning in Your Project

All the functions should be defined in a single Haskell file named **project4.hs**.

```
turnin ––submit lschmidt project4 project4.hs readme.txt
```

Where project4.hs is your Haskell source and readme.txt is your readme file. Please follow the project protocol. Make sure your README states both names AND email addresses of the people in the team, as well as output from the required test databases. Also, should you be unable to complete any part of the assignment, you should say so in the readme file so that I am aware of the lack of functionality when grading.

5 Questions and Project Clarifications

Clarifications regarding this project will be posted to Piazza. You are responsible for these updates.