The University of Melbourne
School of Computing and Information Systems

Declarative Programming
COMP30020

Semester 2, 2018

Project Specification

Project due Monday, 3 September 2018, 5:00PM
Worth 15%

The objective of this project is to practice and assess your understanding of functional programming and Haskell. You will write code to implement the guessing part of a logical guessing game.

The Game

There has been a daring daylight robbery committed by a pair of audacious criminals. Your program's job is to identify the criminals on the basis of the observations of three eyewitnesses. You've assembled a dozen of the usual suspects and now must work out which pair of them are the culprits. You will do this by putting them in lineups in pairs, asking your witnesses to indicate whether you have identified the culprits.

Each eyewitness, though, has a hazy memory, and each mostly only focused on one aspect of the culprits' appearance. The first witness mostly paid attention to the height of the perpetrators (whether *Short* or *Tall*); the second only took note of their hair colour (whether *Blond(e)*, *Redhead*, or *Dark haired*); and the third only noticed gender (*Male* or *Female*). The witnesses are not able to describe the perpetrators, they can only recognise them when they see them, and recognise whether they have the characteristics they observed at the time of the crime.

You can put pairs of people in a lineup as many times as necessary, and may include the same person in a lineup multiple times. For each lineup, your witnesses will tell you how many in the lineup are the people they saw committing the crime, as well as how many of them have the right height, how many have the right hair colour, and how many are of the right sex. Each lineup must have exactly two different people, and the order of people in the lineup is not significant.

Once the witnesses agree that both people in the lineup are guilty, your job is finished and you can charge the culprits. Naturally, the police chief is keen for you to do this as soon as possible.

For simplicity, we will identify each suspect by their characteristics, as a three letter identifier. The first letter indicates the height, either `S` or `T`. The second indicates hair colour, either `B`, `R` or `D`. The third specifies sex, either `M` or `F`. The feedback you receive from the witnesses comprise four numbers: the number of culprits you have identified, the number in the lineup with the right height, the number with the right hair colour, and the number with the right sex.

In counting the number with the correct characteristics, we can only count each person (and each culprit) once. For example, if the lineup has two males and only one of the culprits were male, that is counted as one correct sex. Likewise, if the lineup has one male and one female, but both culprits were male, that also counts as one correct sex. Also note that if one of the suspects in the lineup is identified as a culprit, that person is not counted in the number of correct heights, hair colours, and sexes.

Here are a few examples:

Culprit	**Lineup**	**Answer**
SBM, SBF	*SBM, TRF*	(1,0,0,1)
SBM, SBF	*TRF, SBM*	(1,0,0,1)
SBM, SBF	*TRF, TRM*	(0,0,0,2)
SBM, TDF	*TRF, SDM*	(0,2,1,2)

The Program

You will write Haskell code to determine which suspects to include in each lineup. Each lineup constitutes a "guess" as to which pair of suspects are the perpetrators, for which your program will receive feedback. This will require you to write a function to return your initial lineup, and another to use the feedback from the previous lineup to determine the next one. The former function will be called once per game, and then the latter function will be called repeatedly until it produces the correct lineup. You will find it useful to keep information between guesses; since Haskell is a purely functional language, you cannot use a global or static variable to store this. Therefore, your initial guess function must return this game state information, and your next guess function must take the game state as input and return a new game state as output. You may put any information you like in the game state, but you *must* define a type `GameState` to hold this information. If you do not need to maintain any game state, you may simply define `type GameState = ()`.

You may use any representation you like for suspects, but you must define a type `Person` to hold your representation. The quality of your design is one assessment criterion for this project, so take care to choose (and document!) a good design. You must also define a function `parsePerson` to convert a three-letter string of the form above into a `Person`. You must also define functions `height`, `hair`, and `sex` to return the height, hair colour, and sex of a `Person`. You may represent height, hair colour, and sex any way you like, as long as the types are in the `Eq` class.

You must also write a function `initialGuess` to return your first guess, and `nextGuess` to return your next guess given the results of your previous one.

Here is a summary of the types you must define:

Person Represents a suspect

GameState holds whatever information you need to help decide your next guess based on the previous one.

Here is a summary of the functions you must define:

parsePerson :: String → Maybe Person
takes a three-character string and returns `Just` p, where p is the person specified by that string. If an invalid string is provided, returns `Nothing`.

height :: Person → *whatever you like*
Returns the person's height, represented as any type in the *Eq* class.

hair :: Person → *whatever you like*
Returns the person's hair colour, represented as any type in the *Eq* class.

sex :: Person → *whatever you like*
Returns the person's sex, represented as any type in the *Eq* class.

feedback :: [Person] → [Person] → (Int,Int,Int,Int)
Takes first a list of the true culprits and second a list of the suspects in your lineup, and returns a quadruple of correct suspects, correct heights, correct hair colours, and correct sexes, in that order.

initialGuess :: ([Person],GameState)
Returns your initial lineup and initial game state.

nextGuess :: ([Person],GameState) → (Int,Int,Int,Int) → ([Person],GameState)
takes as input a pair of the previous guess and game state (as returned by `initialGuess` and `nextGuess`), and the feedback to this guess as a quadruple of correct suspects, correct height, correct hair colour, and correct sex, in that order, and returns a pair of the next guess and new game state.

You must call your source file `Proj1.hs` (or `Proj1.lhs` if you use literate Haskell), and it must begin with the module declaration:

```
module Proj1 (Person, parsePerson, height, hair, sex,
              GameState, initialGuess, nextGuess, feedback) where
```

(after any initial documentation comments). Please put all your code in this one file.

I will post a test driver program `Proj1Test.hs`, which will operate similarly to how I actually test your code. I will compile and link your code for testing using the command:

```
ghc -O2 --make Proj1Test
```

or similar. To run `Proj1Test`, give it the target as two separate command line arguments, for example `./Proj1Test TBM SRF` would run your code setting the actual culprits as a tall blond male and a short redhead female. It will then use your `Proj1` module to assemble lineups, showing you all your program's lineup selections and tell you how many guesses your program took to guess the culprits.

```
Your guess 1:  SBM SRM
My answer:  (0,1,2,1)
Your guess 2:  SBF TRM
My answer:  (0,2,2,2)
Your guess 3:  SRF TBM
My answer:  (2,0,0,0)
You got it in 3 guesses!
```

Assessment

This project is to be done individually (not in teams). Your project will be assessed on the following criteria:

10% Correctness of your `feedback` implementation;

10% Correctness of your `initialGuess` and `nextGuess` implementations;

50% Quality of your implementation;

30% Quality of your code and documentation

The correctness of your `feedback` function will be assessed based on the proportion of test cases it passes. The correctness of your `initialGuess` and `nextGuess` functions will be assessed based on whether it succeeds in guessing the culprits in the available time. The correctness of other functions will not be explicitly tested, but will be necessary to the correct functioning of all the tests.

The quality of your implementation will be assessed based on the number of guesses needed to find the given targets. All possible culprit pairs will be tested; full marks will be given for an average of 2.7 guesses or fewer per target, with marks falling on a logarithmic scale as the number of guesses rises.

Note that timeouts will be imposed on all tests. You will have at least 2 seconds to guess each target, regardless of how many guesses are needed. Executions taking longer than that may be unceremoniously terminated, leading to that test being assessed as failing. Your programs will be compiled with `GHC -O2` before testing, so 2 seconds per test is a very reasonable limit (it should be 100 times more than you need).

The quality of your code will be assessed based on appropriate use of Haskell, including the design of your data types, as well as on overall (language-independent) good coding and documentation practices. See the Project Coding Guidelines on the LMS for detailed suggestions for coding and documentation style. This is worth a substantial fraction of your project mark; an hour cleaning up and documenting your code before your final submission will be well rewarded.

Submission

You must submit your project from the student unix server `dimefox2.eng.unimelb.edu.au` or `nutmeg2.eng.unimelb.edu.au`. Note the `2` in each of these host names. Make sure the version of your program source files you wish to submit is on this server, then `cd` to the directory holding your source code and issue the command:

```
submit COMP30020 proj1 Proj1.hs
```

(substitute `Proj1.lhs` for `Proj1.hs` if you use literate Haskell).

Important: you must wait a minute or two (or more if the servers are busy) after submitting, and then issue the command

```
verify COMP30020 proj1 | less
```

This will show you the test results from your submission, as well as the file(s) you submitted. If the test results show any problems, correct them and submit again. You may submit as often as you like; only your final submission will be assessed.

If you wish to (re-)submit after the project deadline, you may do so by adding "`.late`" to the end of the project name (*i.e.,* `proj1.late`) in the `submit` and `verify` commands. But note that a penalty, described below, will apply to late submissions, so you should weigh the points you will lose for a late submission against the points you expect to gain by revising your program and submitting again.

It is your responsibility to verify your submission.

Windows users should see the LMS Resources list for instructions for downloading the (free) Putty and Winscp programs to allow you to use and copy files to the department servers from windows computers. Mac OS X and Linux users can use the `ssh`, `scp`, and `sftp` programs that come with your operating system.

Late Penalties

Late submissions will incur a penalty of 0.5% of the possible value of that submission per hour late, including evening and weekend hours. This means that a perfect project that is much more than 4 days late will receive less than half the marks for the project. If you have a medical or similar compelling reason for being late, you should contact the lecturer as early as possible to ask for an extension (preferably before the due date).

Hints

1. A very simple approach to this program is to simply guess every possible pair of persons until you guess right. There are only 66 distinct pairs, so on average it should only take about 33 lineups, making it perfectly feasible to do in 2 seconds. However, this will give a poor score for guess quality.

2. A better approach would be only to select lineups that are consistent with the answers you have received for previous lineups. You can do this by computing the list of all possible lineups, and removing elements that are inconsistent with any answers you have received to previous lineups. A possible target is inconsistent with an answer you have received for a previous lineup if the answer you would receive for that lineup and that (possible) pair of culprits is different from the answer you actually received for that lineup.

   You can use your `GameState` type to store your previous lineups and the corresponding answers. Or, more efficient and just as easy, store the list of remaining possible targets in your `GameState`, and pare it down each time you receive feedback for a lineup. Using this approach should give you about 80% of the marks for lineup quality.

3. To get full marks for guess quality, you will need to carefully choose each lineup so that it is most likely to leave a small remaining list of possible lineups. This you can do by computing, for each remaining possible lineup, the answer you will receive if it is the actual lineup, and then compute how many of the remaining possible lineups would yield the same output, and compute the *average* number of possible lineups that will remain after each lineup, giving the *expected* number of remaining possible lineups

for each guess, and choose the lineup with the smallest expected number of remaining possible lineups.

For example, suppose there are ten remaining candidate lineups, and one lineup gives the answer (2,0,0,0), three others give (1,0,1,1), and the remaining six give the answer (0,1,1,2). In this case, if you make that guess, there is a 1 in 10 chance of that being the right answer (so you are left with that as the only remaining candidate), 3 in 10 of being left with three candidates, and a 6 in 10 chance of being left with six candidates. This means on average you would expect this answer to leave you with

$$\frac{1}{10} \times 1 + \frac{3}{10} \times 3 + \frac{6}{10} \times 6 = 4.6$$

remaining candidates. You can perform a similar computation for every possible lineup, and select a lineup that gives the minimum expected number of remaining candidates.

Also note that if you do this incorrectly, the worst consequence is that your program takes more lineups than necessary to find the lineup. As long as you only ever guess a possible lineup, every guess other than the right one removes at least one possible lineup, so you will eventually guess the right lineup.

4. Do feel free to discuss the hints on the discussion forum, and to share overall strategy. If you find a different approach that works well, feel free to share it. But please do not share code.

5. Note that these are just hints; you are welcome to use any approach you like to solve this, as long as it is correct and runs within the allowed time.

Note Well:

This project is part of your final assessment, so cheating is not acceptable. Any form of material exchange with other students or anyone else other than subject staff regarding this project, whether written, electronic, or any other medium, is considered cheating, and so is the soliciting of help through public fora. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties. If you have questions regarding these rules, please ask the lecturer.