Programming in Haskell – Project Assignment

UNIZG FER, 2015/2016

Handed out: December 12 2015. Due: January 26 2016

1 Introduction

Your task is to implement modules of a course management system inspired by Ferko, but much simpler. It will contain most of the functionalities needed for the organization of a course such as Programming in Haskell, when combined with other modules.

You will implement these modules in groups of 1-3 people, and will have until the last week of classes to do so. You will present your solution to the teaching staff. You will use the `git` versioning tool and must make your solution available in a *public* repository, using some service such as Github or Bitbucket, both during the development and at the presentation. You must format your project as a cabal project.

You may use any module you wish to aid you in the development process.

2 Modules

The project consists of various modules that interact with each other in various ways. Your task is to implement $n + 1$ of these modules, where n is the number of people in your group. For example, if your group consists of two people, you have to implement three of these modules. You *may*, of course, implement more than is your due and extra points might be assigned for doing so. Note that these extra points will be used *only* for ranking students at the end of the course and cannot be exchanged for homework assignment points. If you have to use a data structure or function from a module you are not implementing, feel free to use dummy values.

2.1 Assignments and Submissions

Implement an `Assignment` data type that maps to a directory structure on disk, as well as a number of functions over it. An assigment is defined by three parameters – a *year*, a *type* and a *number*, as well as a `Configuration` data structure, all of which must be persisted on disk in some manner. For example, the third homework assignment in AY 2015/16 could map to directory `${root}/2015/homework/3`. To elaborate, *mapping to the filesystem* means that the module uses files and directories on the hard disk as the storage medium for its data. It should create a directory structure to store its data, in a manner (more or less) traversable by a human user.

Use the following types, or something similar enough, in the implementation:

```
-- | Academic year shorthand (e.g. 2015 for 2015/16)
type Year = Integer
```

```
-- | An assignment type
data Type = Homework | Exam | Project deriving Show

-- | A an assignment configuration data structure
-- | Uses Data.Time.UTCTime
-- | If files is an empty list, ANY number of files are OK
data Configuration = { published    :: UTCTime -- When to publish
                     , deadline     :: UTCTime -- Submission deadline
                     , lateDeadline :: UTCTime -- Late submission deadline
                     , files        :: [String] -- File names to expect
                     , minScore     :: Double -- Minimum achievable
                     , maxScore     :: Double -- Maximum achievable
                     , required     :: Double -- Score req to pass
                     -- Anything else you might think would be useful
                     } deriving Show

-- | An assignment descriptor
data Assignment = { year   :: Year
                  , type   :: Type
                  , number :: Int
                  -- Anything else you might think would be useful
                  } deriving Show
```

Make sure that the configuration remains stored in some way (e.g. by creating a configuration file in the directory that the descriptor is stored to).

Additionally, you have to create a `Submission` data structure that represents a submitted solution. It should map to (i.e. be stored in) a subdirectory. For example, a homework assignment 5 is defined for AY 2015/16, and requires the files `Homework.hs` and `Exercises.hs` to be submitted. A user with the identifier `Branko` submitted his `Homework.hs` file, but not the exercises. It is then reviewed by user Janko. You might want to create the following directory structure to deal with this case:

```
2015
  homework
    5
      .config
      Assignment.pdf -- See below
      Branko
        Homework.hs
        review-Janko.txt -- See Reviews section
```

Implement the `Submission` structure however you wish, as long as it can be used as given in the functions to below. Furthermore, you have to implement the following functions over such a filesystem (again, it is permitted to perform changes, as long as it remains coherent):

```
-- | Lists the user identifiers for submissions made for an assignment
listSubmissions :: Assignment -> IO [UserIdentifier]
```

```
    -- | Views a single submission in detail
    getSubmission :: Assignment -> UserIdentifer -> IO Submission

    -- | Creates a new assignment from Assignment, configuration and PDF file
    -- | The PDF file should be copied, moved or symlinked so that it is
    -- | accessible from the assignment directory.
    createAssignment :: Assignment -> Configuration -> FilePath -> IO ()


    -- | Gets the configuration object for an assignment
    getConfiguration :: Assignment -> IO Configuration

    -- | Given a solution file body, adds a solution directory/file to the
    -- | directory structure of an assignment. It will indicate an error
    -- | (using Maybe, exceptions, or some other mechanism) if the file is
    -- | not in a defined permitted list. It will override already made
    -- | submissions.
    -- | Assignment -> File Body -> File Name -> Error indication (?)
    upload :: Assignment -> Text -> String -> IO (Maybe Submission)

    -- | Lists the files contained in a submission
    listFiles :: Submission -> IO [FilePath]

    -- | Computes a file path for a submission
    getSubmissionPath :: Submission -> FilePath
```

Make sure to take care of possible errors. If they cannot be handled, throw an exception or use some data type such as Either for error notification.

2.2 User

Create a module that handles users. This includes creating users, authenticating them, and handling their roles. A user is identified by (minimally) these fields:

```
-- | A user identifier (not DB id) like a username or JMBAG
type UserIdentifier = String

-- | The user's role in the course
data Role = Student   Integer -- Academic Year shorthand (2015 for 2015/16)
          | TA        Integer Integer -- AY shorthand range (inclusive)
          | Professor
            deriving Eq, Ord, Show

-- | A user (the definition can be bigger)
data User = { identifier   :: UserIdentifier
            , email        :: String
            , pwdHash      :: String
            , role         :: Role
            } deriving Eq, Show
```

You may freely add fields to the user. Change the listed fields only if necessary. You have to write a `User` implementation over a relational database. You might want to use packages such as `Persistent` and `esqueleto` for this. Provide at least the following methods:

```
-- | Takes a user identifier, e-mail, password and role.
-- | Performs password hashing and stores the user into the
-- | database, returning a filled User. If creating it fails (e.g.
-- | the user identifier is already taken), throws an appropriate
-- | exception.
createUser :: UserIdentifier -> String -> String -> Role -> IO User

-- | Updates a given user. Identifies it by the UserIdentifier (or
-- | maybe database id field, if you added it) in the User and overwrites
-- | the DB entry with the values in the User structure. Throws an
-- | appropriate error if it cannot do so; e.g. the user does not exist.
updateUser :: User -> IO ()

-- | Deletes a user referenced by identifier. If no such user or the
-- | operation fails, an appropriate exception is thrown.
deleteUser :: UserIdentifer -> IO ()

-- | Lists all the users
listUsers :: IO [User]

-- | Lists all users in a given role
listUsersInRole :: Role -> IO [User]

-- | Fetches a single user by identifier
getUser :: UserIdentifier -> IO User

-- | Checks whether the user has a role of AT LEAST X in a given academic
-- | year.
isRoleInYear :: User -> Role -> Integer -> Bool
```

2.3 E-Mail Notifications

Implement an e-mail notification system using the `smtp-mail` library. Additionally, implement a templating language that can process messages and fill in fields from a `Data.Map`. It should also allow for `if-then-else` block some some similar construct, as well as the basic boolean operators. You are free to include additional functionality such as loops or simple helper functions such as `isEmpty`. You can perform the parsing procedure in any way you wish, or use something such as Template Haskell, if applicable. Here is an example of a possible templating language (your implementation may differ):

```
Hello {firstName}

@if(isTA or isProf)
Something
```

```
@else
SomethingElse
@endif

Regards,
{authorName}
```

Provide documentation (e.g. markdown or HTML) that briefly describes how to use your templating language.

The configuration for the e-mail notification utility needs to be kept in a configuration file. Use some human-readable syntax such as `json` or `yaml`. The configuration needs to contain at least a host, port, sender e-mail address, username and password, some of which may be optional.

Provide at least the following functions:

```
-- | An alias for the template contents
type Template = Text

-- | Some configuration object mirroring a file.
-- | Define your own structure and use Maybe X for
-- | optional fields.
data Configuration = { ... } deriving Show

-- | Parses an expression and returns either a result or an error
-- | message, if the parsing fails. If a variable is undefined, you
-- | can either return an error or replace it with an empty string.
-- | You can use a more elaborate type than String for the Map values,
-- | e.g. something that differentiates between Strings and Bools.
compileTemplate :: Template -> Map String String -> Either Error Text

-- | Reads the e-mail configuration object from a file, using some
-- | default path to config file.
readConfig :: IO Configuration

-- | Sends an e-mail with given text to list of e-mail addresses
-- | using given configuration. Throws appropriate error upon failure.
sendMail :: Configuration -> Text -> [String] -> IO ()
```

2.4 Reviews

Implement a review system over the `Assignment` and `Submission` objects. It should contain functionalities for (pseudo-)randomly pairing reviewer-reviewee pairs, role constraints (a TA review is not the same as a student review), as well as be capable of storing and loading said reviews from a database or file system. It should use functionalities from the first module, such as the `Configuration` object defined there. Make sure it provides error checking (e.g. a student cannot receive a grade greater than the maximum, or smaller than the minimum). Implement the following functionalities, or something recognisably similar:

```
-- | A user's role or authorization level as a reviewer
```

```
data Role = Student | Staff deriving Eq, Ord, Show

-- | A review assignment representation
data ReviewAssignment = { reviewer   :: UserIdentifier
                        , reviewee   :: UserIdentifier
                        , role       :: Role
                        , assignment :: Assignment
                        } deriving Eq, Show

-- | A finished review
data Review = { reviewAssignment :: ReviewAssignment
              , score            :: Double
              , text             :: Text
              } deriving Show

-- | Takes an Assignment, a list of reviewer identifiers and a
-- | list of reviewee identifiers and assigns N reviewees for each
-- | reviewer. It makes sure that a user never reviews themselves.
-- | The reviewer is assigned the reviews with the provided role.
assignNReviews :: Assignment -> [UserIdentifier]
                             -> [UserIdentifier]
                             -> Int
                             -> Role
                             -> IO [ReviewAssignment]


-- | Takes an assignment, a list of reviewers and reviewees and a
-- | role. Assigns revieews to reviewers pseudorandomly until the
-- | list of revieews is exhausted. For N reviewers and M
-- | reviewees, ensures no reviewer gets less than
-- | floor (M / N) or more than ceil (M / N) reviews.
-- | Should NOT always assign more reviews to users listed
-- | at the beginning of the reviewer list.
assignReviews :: Assignment -> [UserIdentifier]
                            -> [UserIdentifier]
                            -> Role
                            -> IO [ReviewAssignment]

-- | Stores a list of review assignments into a database or
-- | file system.
storeAssigments :: [ReviewAssignments] -> IO ()

-- | Retrieves all ReviewAssignments for an Assignment from
-- | a database or file system.
assignedReviews :: Assignment -> IO [ReviewAssignment]

-- | Retrieves all ReviewAssignments for an Assignment and
-- | a UserIdentifier, i.e. all the reviews for that assigment
```

```
-- | the user has to perform.
assignmentsBy :: Assignment -> UserIdentifier
                            -> IO [ReviewAssignment]

-- | Retrieves all ReviewAssignments for an Assignment and
-- | a UserIdentifier, i.e. all the reviews for that assignment
-- | where the user is a reviewee.
assignmentsFor :: Assignment -> UserIdentifier
                             -> IO [ReviewAssignment]

-- | Completes a review assignment and stores the result in a
-- | file system or database.
saveReview :: Review -> IO ()

-- | Loads all the completed review results for an assignment
reviews :: Assignment -> IO [Review]

-- | Loads all the completed review results for an assignment
-- | that were performed by a user.
reviewsBy :: Assignment -> UserIdentifer -> IO [Review]

-- | Loads all the completed review results for an assignment
-- | where the user's code was being reviewed.
reviewsFor :: Assignment -> UserIdentifier -> IO [Review]
```

Again, handle errors appropriately. You may change the model slightly (e.g. merge the `Review` and `ReviewAssignment` objects), as long as it is coherent and well documented.

2.5 Scores and Statistics

Extend the Assignment/Submission and Review modules by providing a simple interface for generating and viewing scores and statistics concerning them. It should allow us to get the following:

- The statistics for an academic year,
- The statistics for all items of a certain type (e.g. HA) in an academic year,
- The statistics for a certain item of a certain type (e.g. HA3) in an academic year,
- The data for a certain user for the three combinations above.

The statistics should comprise of at least the following:

- The maximum and minimum *achievable* scores,
- The minimum passing score,
- The minimum and maximum *achieved* scores,
- The score mean and median,

- A set of data points that can be used for plotting a graph (e.g. a histogram with 10 buckets).

The data for a single user-assignment pair should consist of at least the following:

- The user's achieved score,
- Whether the user has a passing score.

Additionally, it should be possible to define one of the first three combinations (AY, AY + type, or AY + type + number) and receive a list of users with their assigned data, sorted by score in descending order.

Note: Only reviews done in the `Staff` role are considered relevant for the statistics!

Here is a sketch of a possible solution:

```
-- | A "bucket" containing a count of values
-- | in a certain range for plotting a histogram.
data Bucket = { rangeMin :: Double
              , rangeMax :: Double
              , count    :: Int
              } deriving Show

-- | A statistics container
data Statistics = { minPossible :: Double
                  , maxPossible :: Double
                  , mean        :: Double
                  , median      :: Double
                  , minAchieved :: Double
                  , maxAchieved :: Double
                  , histogram   :: [Bucket]
                  } deriving Show

-- | A user's score and accompanying data
data Score = { points :: Double
             , passed :: Bool
             } deriving Eq, Ord, Show

data UserScore = { identifier :: UserIdentifier
                 , score      :: Score
                 } deriving Eq, Ord, Show

-- | Computes the statistics for an entire academic year
stats :: Integer -> IO Statistics

-- | Computes the statistics for a certain assignment
-- | type in an academic year.
typeStats :: Integer -> Type -> IO Statistics

-- | Computes the statistics for a certain assignment
-- | (combination of AY, type and number)
```

```
assignmentStats :: Assignment -> IO Statistics

-- | Fetches the user score for an entire academic year
score :: Integer -> UserIdentifier -> IO Score

-- | Fetches the user score for an assignment type in AY
typeScore :: Integer -> Type -> UserIdentifier -> IO Score

-- | Fetches the user score for a certain assignment
assignmentScore :: Assignment -> UserIdentifier -> IO Score

-- | Fetches the ranked users for an entire AY, sorted in
-- | descending order by score.
ranked :: Integer -> IO [UserScore]

-- | Fetches the ranked users for an assignment type in AY,
-- | sorted in descending order by score.
typeRanked :: Integer -> Type -> IO [UserScore]

-- | Fetches the ranked users for a specific assignment, sorted
-- | in descending order by score.
assignmentRanked :: Assignment -> IO [UserScore]
```

Handle errors appropriately. You do not have to follow the sketched model, as long as your solution has at least the listed features, is coherent and well-documented.

2.6 Other

If you have an idea for another component that might fit into the project, feel free to suggest it to us. If we deem it appropriate, you will be permitted to implement it in place of one of the modules listed above. Exceptionally, if your idea is of a significantly higher difficulty than the presented modules, it might be exchanged for multiple modules. For example, if a team of two suggests a fairly difficult module, they might only have to implement their suggested module and one other component instead of three modules.

3 Project Structure

Your code should be structured and split into modules according to functionality. This means that there should be *at least one* module for each sub-task you have implemented. Modularisation of code is especially important in larger tasks such as this one.

4 Submission

The project is due for Tuesday, January 26th, in the last week of classes, at a time and place yet to be determined, in the evening hours. You will demonstrate your project to the teaching assistants on your own laptop. If at all possible, all members of the team should be present. Your submission will be graded depending on the percentage of implemented functionality, robustness and code quality. You need at least 15 points (out of 30) to pass the course.