Assignment: Haskell and Java 8 Features

Instructions

This assignment is made of three parts, the third one (proposing additional exercises in Haskell) is optional.

Part 1 - Functional programming in Haskell

This assignment requires you to implement a type constructor providing the functionalities of Multisets. Your implementation must be based on the following *concrete* Haskell definition of the `ListBag` type constructor:

```
data ListBag a = LB [(a, Int)]
  deriving (Show, Eq)
```

Therefore a `ListBag` contains a list of pairs whose first component is the actual element of the multiset, and the second component is its *multiplicity*, that is the number of occurrences of such element in the multiset. A `ListBag` is **well-formed** if it does not contain two pairs `(v, k)` and `(v', k')` with `v = v'`.

Exercise 1: Constructors and operations

The goal of this exercise is to write an implementation of multisets represented concretely as elements of the type constructor `ListBag`. As described below, the proposed implementation must be well documented and must pass the provided tests.

- Using the type constructor `ListBag` described above, implement the predicate `wf` that applied to a `ListBag` returns `True` if and only if the argument is well-formed. Check that the inferred type is `wf :: Eq a => ListBag a -> Bool`.

Important: All the operations of the present exercise that return a `ListBag bag` must ensure that the result is well-formed, i.e., that `wf bag == True`.

- Implement the following constructors:
  1. `empty`, that returns an empty `ListBag`
  2. `singleton v`, returning a `ListBag` containing just one occurrence of element `v`
  3. `fromList lst`, returning a `ListBag` containing all and only the elements of `lst`, each with the right multiplicity
- Implement the following operations:
  1. `isEmpty bag`, returning `True` if and only if `bag` is empty
  2. `mul v bag`, returning the multiplicity of `v` in the `ListBag bag` if `v` is an element of `bag`, and `0` otherwise
  3. `toList bag`, that returns a list containing all the elements of the `ListBag bag`, each one repeated a number of times equal to its multiplicity
  4. `sumBag bag bag'`, returning the `ListBag` obtained by adding all the elements of `bag'` to `bag`

Testing: The attached file `testEx1.hs` contains some tests that can be used to check the correctness of the implemented functions. To run the test, it is probably necessary to install the **Test.HUnit** Haskell module. Next it is sufficient to load `testEx1.hs` in the interpreter, and execute `main`. Some more tests will be released in the following days. **Note that solutions that do not pass such tests will not be evaluated, and a revision will be requested.**

Expected output: A Haskell source file called `Ex1.hs` containing a Module (see Section "Making our own modules") called `Ex1`, defining the data type `ListBag` (copy it from above) and *at least* all the functions described above. The module can include other functions as well, if convenient. **Note:** The file has to be adequately commented, and each function definition must be preceded by its type, as inferred by the Haskell compiler.

Exercise 2: Mapping and folding

The goal of this exercise is to experiment with class constructors by adding some functions to the module developed for Exercise 1.

1. Define an instance of the constructor class `Foldable` for the constructor `ListBag` defined in Exercise 1. To this aim, choose a minimal set of functions to be implemented, as described in the documentation of `Foldable`. Intuitively, folding a `ListBag` with a binary function should apply the function to the elements of the multiset, ignoring the multiplicities.
2. Define a function `mapLB` that takes a function `f :: a -> b` and a `ListBag` of type `a` as an argument, and returns the `ListBag` of type `b` obtained by applying `f` to all the elements of its second argument.
3. Explain (in a comment in the same file) why it is not possible to define an instance of `Functor` for `ListBag` by providing `mapLB` as the implementation of `fmap`.

Testing: Also for this exercise a test file will be provided, to be used for checking the code.

Expected output: A Haskell source file `Ex2.hs` containing a module called `Ex2`, which imports module `Ex1` and includes **only** the new functions defined for this exercise. **Note:** The file has to be adequately commented, and each function definition has to be preceded by its type, as inferred by the Haskell compiler.

Part 2: Programming in Java 8 with the Stream API

The following exercises require using the files `oscar_age_female.csv` and `oscar_age_male.csv` that store information about the winners of the Oscar prize. These are text files where each line is a record storing the following fields in csv format:

```
Index, Year, Age, Name, Movie
```

where the field *Year* denotes the year when the Oscar was won, and *Age* the age the winners had when they were awarded. The other fields are self-explanatory.

Exercise 3: Basic structures

Write a class `WinnerImpl` that

1. implements the simple interface `Winner` (attached) that represents a record of the winner database
2. contains the static method `loadData` that given a `String[]` containing the paths of some winner databases returns a `Collection<Winner>` representing the content of the databases. Note that the first line of each file describes the format of the fields and that the values of fields `Name` and `Movie` are enclosed in double quotes, e.g., "Coquette". Your solution must suitably deal with these cases.

Expected output: A Java source file `WinnerImpl.java` providing the required functionalities. **Note:** The file has to be adequately commented.

Exercise 4: Methods using the Stream API

Write a class `WinnerOperations` that contains the following **static** methods, which should make extensive use of the Stream API:

1. method `oldWinners` that given a `Stream<Winner>` returns a new `Stream<String>` containing the names of the winners that are *older* than 35 *sorted alphabetically*.
2. method `extremeWinners` that given a `Stream<Winner>` returns a `Stream<String>` containing the names of all the youngest and of all the oldest winners, *sorted in inverse alphabetical ordering*.
3. method `multiAwardedFilm` that given a `Stream<Winner>` returns a `Stream<String>` containing the title of films who won two prizes. The elements of the stream must be in *chronological order*, i.e. in increasing order of year of the corresponding records in the database.
4. method `runJobs` that given a `Stream<Function<Stream<T>,Stream<U>>>` of jobs and a `Collection<T> coll` returns a `Stream<U>` obtained by concatenating the results of the execution of all the jobs on the data contained in `coll`.
5. method `main`, that reads the databases of Winners `oscar_age_female.csv` and `oscar_age_male.csv` using method `loadData` of `WinnerImpl`, and prints the result of invoking methods `oldWinners`, `extremeWinners` and `multiAwardedFilm` on the databases by exploiting `runJobs`.

Expected output: A Java source file `WinnerOperations` providing the required functionalities. The execution of the program should print the expected result assuming that the two `csv` files are in the same directory of the `class` file. **Note:** The file has to be adequately commented.

Part 3 - Functional programming in Haskell [Optional]

The next exercises build over those presented in Part 1.

Exercise 5: Multisets as Monad [Optional]

The goal of this exercise is to understand how multisets can be equipped with a monadic structure. The resulting monad represents conceptully both a container (a bag of distinct elements, each one with its multiplicity), and a computational effect (a function `a -> ListBag b` is a non-deterministic function returning the multiset of possible results).

1. Define the operations `returnLB` and `bindLB`, taking inspiration from the operations of the `Monad` type constructor.
2. Try to define an instance of `Monad` for `ListBag` using the functions just defined. Discuss whether this is possible or not, and if not what conditions have to be released in order to obtain an instance of `Monad`
3. Write some tests for the functions just implemented.

Expected output: A Haskell source file `Ex5.hs` containing a module called `Ex5`, which imports modules `Ex1` and `Ex2`, and includes **only** the new functions defined for this exercise, the requested comments, and the tests. **Note:** The file has to be adequately commented, and each function definition has to be preceded by its type, as inferred by the Haskell compiler.

Exercise 6: Abstract Data Type [Optional]

Consider abstract data types as defined here, and in particular in Section 3.1. The goal of this exercise is to re-engineer the definition of `ListBag` of Exercise 1 and Exercise 2 as specific instances of an Abstract Data Type Constructor called `MultiSet`.

1. Define a new constructor class `MultiSet` defining an abstract data type with the same constructors and operations as `ListBag`
2. Make `ListBag` an instance of `MultiSet`
3. Provide a second, different instance of `MultiSet`, by either exploiting a new concrete representation of multisets or by reusing some data structure provided by the Haskell API.
4. Write a simple function manipulating `Multiset`, and show that it can be applied to both implementations.

Expected output: A Haskell source file `Ex6.hs` containing a module called `Ex6`, which imports modules `Ex1` and `Ex2`, and includes **only** the new definitions developed for this exercise. **Note:** The file has to be adequately commented, and each function definition has to be preceded by its type, as inferred by the Haskell compiler.

Author: Andrea Corradini & Matteo Busi

Created: 2018-12-02 Sun 17:51

Validate