Assignment: Haskell, Java Stream API & Python Versions 1.2 - January 22, 2019

Instructions

This assignment is made of three parts, consisting of exercises on Haskell, the Stream API of Java and Python, respectively.

Warning: This document is subject to changes. Check that you are reading always the most recent version.

*** Changes with respect to the previous version are in bold. ***

Part 1 - Implementing multisets 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 files `testEx1.hs` and `testEx12.hs` contain some tests that can be used to check the correctness of the implemented functions. To run the tests, it is probably necessary to install the **Test.HUnit** Haskell module. Next it is sufficient to load file `testEx1.hs` or `testEx12.hs` in the interpreter, and execute `main`.

Note that solutions that do not pass such tests will not be evaluated, and a revision will be requested.

Solution format: 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`.

Solution format: 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 - A Map-Reduce framework exploiting the Java Stream API

The Map-Reduce paradigm is widely used for processing huge amounts of data in a parallel and distributed setting. In this assigment, students are required to implement a simple software framework providing the functionalities of Map-Reduce, but ignoring the aspects of parallelism and distribution. As a proof of concept, two simple working instances of the framework should be implemented as well.

For an introduction to the Map-Reduce framework see the paper MapReduce: Simplified Data Processing on Large Clusters. For a presentation of Map-Reduce as Software Framework, identifying the *hot spots*, see https://en.wikipedia.org/wiki/MapReduce#Dataflow (since we ignore the distribution aspects, you can ignore the *Partition function*.)

Solution format: An archive `MapReduce-<yourSurname>.zip` containing the Java files implementing Exercises 3, 4, and (optionally) 5. If you use `NetBeans`, please send in the archive the entire project.

Exercise 3 - The framework

Following the guidelines presented in the lesson of October 22, 2019 (see http://pages.di.unipi.it/corradini/Didattica/AP-19/index.html#framework), and more specifically the *Template Method design pattern*, implement in Java a Map-Reduce software framework providing the functionalites described in the above documentation and respecting the following constraints:

1. For `key/value` pairs, the framework must use the attached class `Pair.java` (you can change its package, but nothing else).
2. The hot spots of the framework are the methods `read`, `map`, `compare`, `reduce` and `write`.
3. The framework must use, when possible, the Stream API. For example, `map` takes a stream of key-value pairs as argument and returns a stream of key-value pairs (types of argument and result may differ, of course).

Exercise 4 - Counting words

By instantiating the framework, implement a program that counts the occurrences of words of length greater than 3 in a given set of documents, respecting the following constraints:

1. The program should ask the user for the absolute path of the directory where documents are stored. Only files ending in `.txt` should be considered.
2. The `read` function must return a stream of pairs `(fileName, contents)`, where `filename` is the name of the text file and `contents` is a list of strings, one for each line of the file. For the `read` function you can exploit the enclosed class `Reader.java` in the way you prefer.
3. The `map` function must take as input the output of `read` and must return a stream of pairs containing, for each word (of length greater than 3) in a line, the pair `(w, k)` where `k` is the number of occurrences of `w` in that line.
4. The `compare` function should compare strings according to the standard alphanumeric ordering. (The result should adhere to the standard Java conventions, see the `compareTo` method of interface `Comparable`.)
5. The `reduce` function takes as input a stream of pairs `(w, lst)` where `w` is a string and `lst` is a list of integers. It returns a corresponding stream of pairs `(w, sum)` where `sum` is the sum of the integers in `lst`.
6. The `write` function takes as input the output of `reduce` and writes the stream in a CSV (Comma Separated Value) file, one pair per line, in alphanumeric ordering. For the `write` function you can exploit the enclosed class `Writer.java` in the way you prefer.

For testing the program you can use the enclosed archive `Books.zip` which contains parts of some famous books as downloaded from the pages of the *Gutenberg Project*.

Exercise 5 - [Optional] Producing an Inverted Index

By instantiating the framework, implement a program that generates an *Inverted Index* (for words of length greater than 3). That is, given as input the absolute path of a directory, the program prints in a CSV file for each word `w` (of length greater than 3) appearing in the `.txt` documents of the directory, a line `w, filename, line` if `w` appears in line number `line` of file `filename`. The lines should be sorted in the natural way.

Part 3 - Benchmarking functions in Python, with Multithreading

You want now to benchmark your Python functions and you want to make sure that it is done in the right way (hopefully... :P)

Solution format: A single Python file called `benchmark.py` containing the solutions to the following three exercises.

Exercise 6 - A decorator for benchmarking

Define a Python decorator called `benchmark`. When invoking a function `fun` decorated by `benchmark`, `fun` is executed possibly several times (discarding the results) and a small table is printed on the standard output including the average time of execution and the variance.

The exact behaviour of the `benchmark` decorator is ruled by the following optional parameters:

- `warmups` : The number of warm-up invokations to `fun` (i.e. invokations whose timing must be ignored) (default: `warmups = 0`);
- `iter` : The number of times `fun` must be invoked and whose timing must be taken into account for the final metrics (default: `iter = 1`);
- `verbose` : Whether the execution should be verbose (i.e. if it must print the timing of each warm-up round and invokation) or not (default `verbose = False`);
- `csv_file` : A CSV file name where the benchmark information must be written. The header of the file will be in the form `run num, is warmup, timing` with the intuitive meaning (default `csv_file = None`, meaning that the benchmark information is only displayed on screen).

Exercise 7 - Testing the decorator with multithreading

Test your implementation by also evaluating the effectiveness of multhitreading in Python.

Using the `threading` module of the Standard Library and exploiting the `benchmark` decorator, write a function `test` that executes a function `f` (passed as parameter) with varying numbers of iterations and degrees of parallelism. More preciselsy, the `test` first runs `f` 16 times on a single thread, then 8 times on two threads, then 4 times on 4 threads, and finally 2 times on 8 threads. The program must write the benchmarking information for the four scenarions in corresponding files named `f_<numthreads>_<numiterations>`.

Run the program using a function that computes the `n`-th Fibonacci number in the standard, inefficent, double recursive way (choose `n` carefully) .

Discuss briefly the results in a comment in the Python file.

Exercise 8 - [Optional] Downloading and executing Python scripts

Define another decorator called `prepost` taking as parameters two URLs, the first indicating the location of a Python script to be executed before the function being decorated, the second denoting the location of a Python script to be executed after the function being decorated.

*** Test the decorator passing the URLs http://pages.di.unipi.it/corradini/Didattica/AP-19/PROG-ASS/02/pre.py and http://pages.di.unipi.it/corradini/Didattica/AP-19/PROG-ASS/02/post.py ***

*** Hint: Use the `requests` library to download the scripts (it is not part of the Standard Library, so you might need to install it, depending on the Python distribution you are using). You can download the scripts to temporary files, and then use `os.system(.)` to execute them invoking the Python interpreter. ***

Author: Andrea Corradini & Matteo Busi

Created: 2020-01-22 Wed 20:07

Validate