Assignment 2: Haskell and the Java Stream API
Version 1.1 - December 29, 2022

This assignment is made of two parts, consisting of exercises on Haskell, and on the Java Stream API, respectively. It is distributed with an archive `aux_files.zip` containing some auxiliary files.

This document is subject to changes. Check on the course web page that you are now reading the most recent version.

Premise: the *"ciao"* of a string

This definition will be used in some of the exercises below. Given a string `str`, we define its *ciao* (*characters in alphabetical order*) as the string having the same length of `str` and containing all the characters of `str` in lower case and alphabetical order. As an example, the *ciao* of "Hello" is "ehllo". A *ciao string* is a string that is equal to its *ciao*. Clearly, two strings have the same *ciao* if and only if each one is an anagram of the other.

Part 1: Multisets in Haskell

This assignment requires you to implement a type constructor providing the functionalities of *multisets* (also known as *bags*), that is, collections of elements where the order does not count, but each element can occur several times. Your implementation must be based on the following concrete Haskell definition of the `MSet` type constructor:

```
data MSet a = MS [(a, Int)]
          deriving (Show)
```

MS value constructor that has 1 field = a list of pairs (a , int)

Therefore an `MSet` contains a list of pairs whose first component is an element of the multiset, and the second component is its *multiplicity*, that is the number of occurrences of such element in the multiset. An `MSet` is **well-formed** if for each of its pairs `(v,n)` it holds `n > 0`, and if it does not contain two pairs `(v,n)` and `(v',n')` such that `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 `MSet`.

- Implement the following constructors:
  - `empty`, that returns an empty `MSet`
- Implement the following operations:
  - `add mset v`, returning a multiset obtained by adding the element `v` to `mset`. Clearly, if `v` is already present its multiplicity has to be increased by one, otherwise it has to be inserted with multilplicity 1.
  - `occs mset v`, returning the number of occurrences of `v` in `mset` (an `Int`).
  - `elems mset`, returning a list containing all the elements of `mset`.
  - `subeq mset1 mset2`, returning `True` if each element of `mset1` is also an element of `mset2` with the same multiplicity at least.

  - `union mset1 mset2`, returning an `MSet` having all the elements of `mset1` and of `mset2`, each with the sum of the corresponding multiplicites.

- Class Constructor Instances
  - Define `MSet` to be an instance of the class constructor `Eq`, implementing equality as follows: two multisets are equal if they contain the same elements with the same multiplicity, regardless of the order.
  - Define `MSet` to be an instance of the constructor class `Foldable`. To this aim, choose a minimal set of functions to be implemented, as described in the documentation of [`Foldable`](). Intuitively, folding a multiset with a binary function should apply the function to the elements of the multiset, ignoring the multiplicities.
  - Define a function `mapMSet` that takes a function `f :: a -> b` and an `MSet` of type `a` as arguments, and returns the `MSet` of type `b` obtained by applying `f` to all the elements of its second argument. Explain (in a comment in the same file) why it is not possible to define an instance of `Functor` for `MSet` by providing `mapMSet` as the implementation of `fmap`.

Important: All the operations of the present exercise that return an `MSet` must ensure that the result is *well-formed*, as defined above. Your code should not use the Haskell module Data.MultiSet or other similar modules, but it can use the functions of the Prelude.

Solution format: A Haskell source file called `MultiSet.hs` containing a Module (see Section "Making our own modules") called `MultiSet`, defining the data type `MSet` (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: Testing multisets

The goal of the exercise is testing the implemented functionalities. In a file named `TestMSet.hs`, import `MultiSet.hs` and

1. Define a function `readMSet` that reads a text file whose name is passed as argument (as a string), and returns a new `MSet` containing the *ciao* of all the words of the file, each with the corresponding mutiplicity.
2. Define a function `writeMSet` that given a multiset and a file name, writes in the file, one per line, each element of the multiset with its multiplicity in the format “`<elem> - <multiplicity>`”.
3. Define a function `main :: IO()` which does the following:
   a. Using `readMSet`, from directory `aux_files` it loads files `anagram.txt`, `anagram_s1.txt`, `anagram_s2.txt` and `margana2.txt` in corresponding multisets, that we call `m1`, `m2`, `m3` and `m4` respectively;
   b. Exploiting also the functions imported from `MultiSet.hs`, it checks the following facts and prints a corresponding comment:
      i. Multisets `m1` and `m4` are not equal, but they have the same elements;
      ii. Multiset `m1` is equal to the union of multisets `m2` and `m3`;

c. Finally, using `writeMSet` it writes multisets `m1` and `m4` to files `anag-out.txt` and `gana-out.txt`, respectively.

For reading and writing files you can use the functions `readFile` and `writeFile` of the Haskell Prelude (https://hackage.haskell.org/package/base-4.16.0.0/docs/Prelude.html).

Solution format: A Haskell source file `TestMSet.hs` with the functions described above, which can be executed using `runghc`
(see https://downloads.haskell.org/~ghc/9.0.1/docs/html/users_guide/runghc.html)

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.

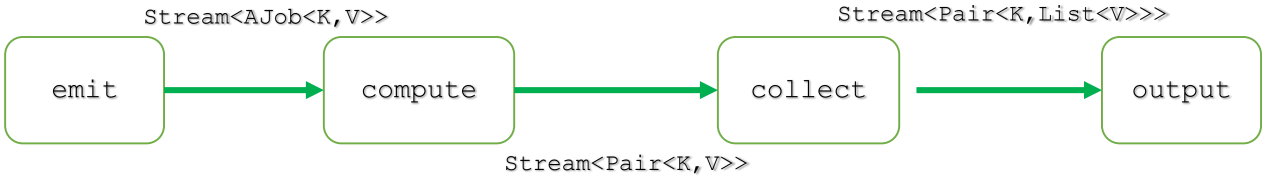
Part 2: A job scheduler exploiting the Java Stream API

In this assignment, students are required to implement a simple software framework providing the functionalities of a job scheduler, but ignoring the aspects of parallelism and distribution. More precisely, the framework includes an *emitter* of jobs, a *compute* phase executing the jobs, a *collect* stage grouping them, and an *output* action printing the results in a suitable format. As a proof of concept, a simple working instance of the framework should be implemented as well.

Exercise 3: The framework

Following the guidelines presented in the lesson of November 16 2021, *On Designing Software Frameworks*, (see http://pages.di.unipi.it/corradini/Didattica/AP-22/index.html#framework), and more specifically the ***Template Method design pattern***, implement in Java a `JobScheduler` software framework, respecting the following specifications:

1. The framework must be generic, using type variables `K` and `V` for the types of `keys` and `values` respectively.
2. For `key/value` pairs, the framework must use the class `Pair.java` from `aux_files.zip` (you can change its package, but nothing else).
3. Jobs will be instances of (subclasses) of the abstract class `AJob.java`, also enclosed, containing the abstract method `execute` with no parameter and returning a stream of `key/value` pairs.



4. The framework must include the following methods, conceptually composed as in the picture:
   - `emit`, which generates a stream of jobs;
   - `compute`, which executes the jobs received from `emit` by invoking `execute` on them, and returns a single stream of `key/value` pairs obtained by concatenating the output of the jobs;
   - `collect`, which takes as input the output of `compute` and groups all the pairs with the same keys in a single pair, having the same key and the list of all values;

- `output`, which prints the result of `collect` in a convenient way.

5. Methods `compute`, `collect` and `main` must be frozen spots of the framework, while `emit` and `output` must be hot spots.

Exercise 4: Counting anagrams

Write a program that given the absolute path of a directory prints the number of anagrams of all the words contained in a set of documents in that directory. **The program must be an instance of the framework of the previous point.** You should ignore all words of less than four characters, and those containing non-alphabetic characters. Also, uppercase and lowercase letters should not be distinguished.

Here are some guidelines:

1. Create a subclass of `AJob` having a constructor that accepts the name of a file as parameter; the `execute` method must read the file, and it must return a stream containing all pairs of the form `(`*`ciao`*`(w), w)` where `w` is a word of the file satisfying the above properties.
2. `Emit` asks the user for the absolute path of a directory where documents are stored. It visits the directory and creates a new job for each file ending with `.txt` in that directory.
3. `Output` should write the list of ciao keys and the number of words associated with each key, one per line, in file `count_anagrams.txt`, in the format “<ciao_key> - <num>”).

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

Solution format: An archive `JobScheduler.zip` containing the Java files implementing Exercises 3 and 4, suitably commented. If you use `NetBeans`, please send in the archive the entire project.