Assignment 2: Haskell, Java Stream API & Python
Version 1.0 - December 6, 2021

This assignment is made of three parts, consisting of exercises on Haskell, the Stream API of Java and Python, 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: Dictionaries in Haskell

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

```
data Dictionary a b = Dict [(a, [b])]
            deriving (Show)
```

Therefore a `Dictionary` contains a list of pairs whose first component is the key, and the second component is the list of elements associated with that key. A `Dictionary` is **well-formed** if it does not contain two different pairs `(k, vs)` and `(k', vs')` with `k = k'`.

Exercise 1: Constructors and operations

The goal of this exercise is to write an implementation of dictionaries represented concretely as elements of the type constructor `Dictionary`.

- Implement the following constructor:

- `empty`, that returns an empty `Dictionary`

- Implement the following operations:

- `insert dict k v`, returning a dictionary obtained by inserting in `dict` the element `v` with key `k`.

- `lookup dict k`, returning a value of type `Maybe [b]`, which is the list of elements with key `k`, if such list exists in `dict`, and `Nothing` otherwise.

- `keys dict`, returning a list containing the keys of `dict`.

- `values dict`, returning a single list containing all the values present in `dict`.

- `merge dict1 dict2`, returning the `Dictionary` obtained by merging the contents of the two dictionaries.

- Define `Dictionary` to be an instance of type class `Eq`, implementing equality according to the following specification: two dictionaries are equal if they contain the same keys, possibly not in the same order, and if for each key `k` the two sets of values associated with `k` are equal (that is, the corresponding lists contain the same values, disregarding the ordering and possible repetitions).

Important: All the operations of the present exercise that return a `Dictionary` must ensure that the result is *well-formed*, as defined above.
Solution format: A Haskell source file called `Dictionary.hs` containing a Module (see Section "Making our own modules") called `Dictionary`, defining the data type `Dictionary` (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 dictionaries

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

1. Define a function `readDict` that reads a text file whose name is passed as argument (as a string), and returns a new `Dictionary` after adding each word of the file using its *ciao* as key.
2. Define a function `writeDict` that given a dictionary and a file name, writes in the file, one per line, each key of the dictionary together with **the length** of the list of values associated with the key.
3. Define a function `main :: IO()` which does the following:
    a. Using `readDict`, from directory `aux_files` it loads files `anagram.txt`, `anagram_s1.txt`, `anagram_s2.txt` and `margana2.txt` in corresponding dictionaries, that we call `d1`, `d2`, `d3` and `d4` respectively;
    b. Exploiting also the functions imported from `Dictionary.hs`, it checks the following facts and prints a corresponding comment:
        i. Dictionaries `d1` and `d4` are not equal, but they have the same keys (ignoring the ordering);
        ii. Dictionary `d1` is equal to the merge of dictionaries `d2` and `d3`;
    c. Finally, using `writeDict` it writes dictionaries `d1` and `d4` 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 `TestDict.hs` with the functions described above.
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.

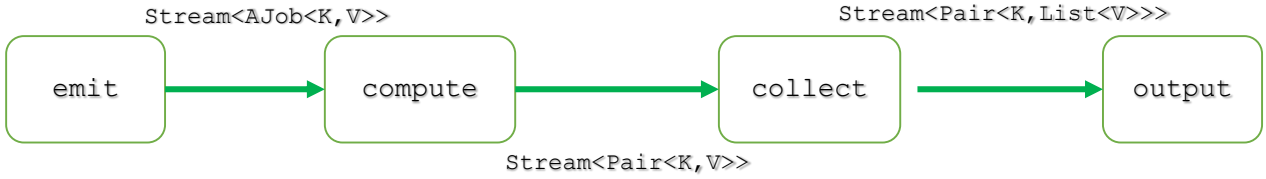
Part 2: A job scheduler exploiting the Java Stream API

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-21/index.html - framework), and more specifically the ***Strategy 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.

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.

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, by developing 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` (or `count_anagrams.csv`, in CSV format).

For testing the program you can use the files in the enclosed archive `Books.zip` which contains parts of some famous books as downloaded from the pages of the *Gutenberg Project*. (before the site became inaccessible from Italy, see *Raffaele Angius, Perché il Progetto Gutenberg sarà sotto sequestro per sempre*).

Part 3 - Benchmarking functions in Python, with Multithreading

In the last lesson on Python and the GIL I reported a claim from the literature: ***"Two threads calling a function may take twice as much time as a single thread calling the function twice"***.

This assignment requires to write a parametric decorator that can be used to check if this claim is true, at least on your machine. The decorator should measure the execution time of invoking several times a function in parallel on a given number of threads.

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

Exercise 5 - A decorator for multi-threaded benchmarking

Define in Python a parametric decorator called `bench`. When invoking a function `fun` decorated by `bench`, `fun` is executed several times in parallel on several threads (discarding the results), and the whole is repeated a few times returning the average time of execution and the variance.

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

- `n_threads`: The number of threads (default: `n_threads = 1`)
- `seq_iter`: The number of times `fun` must be invoked in each thread (default: `seq_iter = 1`)
- `iter`: The number of times the whole execution of the `n_threads` threads, each invoking `seq_iter` times `fun`, is repeated. For each execution, the execution time has to be computed (default: `iter = 1`)

The decorated function must return a dictionary of the following shape, containing the function name and the tuple of arguments, the values of the three parameters just described, as well as the mean value and the variance of the execution times of the `iter` iterations:

{'fun': 'fib_run', 'args': (28,), 'n_threads': 2, 'seq_iter': 8, ‘iter’:5, 'mean': 3.382881900004577, 'variance': 0.0032192657625461194}

Use the `threading` module to exploit Python mutithreding, `perf_counter()` from module `time` for computing the execution time, and module statistics for computing mean value and variance.

Exercise 6 – Exploiting the decorator

Exploit the `bench` decorator to evaluate the effectiveness of multhitreading in Python. Write a function `test` that has three parameters: an integer `iter`, a function `fun`, and a tuple of arguments `args`. Function `test` must execute `fun` on `args` with varying numbers of iterations and degrees of parallelism. More precisely, `test` must invoke `bench` a first time to execute 16 times `fun` on `args` on a single thread (passing `iter` as further parameter), then, similarly, to run `fun` 8 times on two threads, then 4 times on 4 threads, and finally 2 times on 8 threads. The program must write the information returned by the four invocations of `bench` in a file named `<fun>_<args>_<n_threads>_<seq_iter>`, or something similar.

Run the test on simple functions with different degrees of usage of the CPU, for example here are two extreme cases you can use:

```
def just_wait(n):         # NOOP for n/10 seconds
    time.sleep(n * 0.1)

def grezzo(n):             # CPU intensive
    for i in range(2**n):
        pass
```

Discuss the results of the experimentation in a comment at the end of the Python file.
