Assignment 2: Haskell & Python, Version 1.0 - December 21, 2024

This assignment is made of two parts, consisting of exercises on Haskell and on 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: 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)
```

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 must be adequately commented, and each function definition has to be preceded by its type, as inferred by the Haskell compiler.

Exercise 3: Decorators in Python

The lecturer does not appreciate the fact that in Python *type hints* (also known as *annotations*) are ignored by the interpreter. On the other hand, he likes Python decorators. Therefore, he started writing some decorators to perform some very simple type checking. To this aim, he used the Python module **inspect**, supporting reflection.

Unfortunately, the file containing the decorators (called **`type_checking_decorators.py`**) was lost, and the lecturer is left only with some test files and logs. Your assignment consists of writing the three lost decorators (called **`print_types`**, **`type_check`** and **`bb_type_check`**) so that the tests can be executed again producing essentially the same output. The test files are **`test1.py`**, **`test2.py`** and **`test3.py`** in directory **`aux_files`**. The log file **`logN.txt`** (with N ∈ {1,2,3}) shows the effect of executing **`python3 testN.py`**. In the log files, lines starting with **`'==>'`** are printed by the code of the test file, the others are printed by the decorator.

You can assume that the annotated functions only use positional arguments. As it can be seen from the examples, if a parameter does not have a type hint then its type is **`<class 'inspect._empty'>`**. Here is a description of the three decorators.

Decorator **`print_types`** uses introspection to print, for each argument of the decorated function, the type hint and the actual parameter of the function call, with its type. Here is an example of a simple function **`foo`** decorated with **`print_types`**, and the expected output of an invocation (the lines in yellow are printed by the decorator; see file **`log1.txt`** for more examples).

```
>>> from type_checking_decorators import print_types
>>> @print_types
... def foo(x:int, y, z:float) -> str:
...     return str(x) + str(y) + str(z)
...
>>> foo(5,' is ','similar to 5.0')
Formal par 'x':<class 'int'>; actual par '5':<class 'int'>
Formal par 'y':<class 'inspect._empty'>; actual par ' is ':<class 'str'>
Formal par 'z':<class 'float'>; actual par 'similar to 5.0':<class 'str'>
Result type <class 'str'>; actual result '5 is similar to 5.0':<class 'str'>
'5 is similar to 5.0'
```

Decorator **`type_check`** refines the behaviour of **`print_types`** as follows: (1) it only prints something for one parameter or for the result of the function call if there is a disagreement between the value type and a type hint, otherwise nothing is printed; (2) each parameter is identified by its position in the parameter list starting from 0, not by its name. Here I show the effect of decorating the same function foo above with **`print_types`**. Note that when invoked, nothing is printed for the first parameter (because 5 is of type int), for the second (because y has no type hint), and for the result, which is a string as required; instead, the third parameter is not a float but a string, thus a message is printed. See file **`log2.txt`** for more examples.

```
>>> from type_checking_decorators import type_check
>>> @type_check
... def foo(x:int, y, z:float) -> str:
...     return str(x) + str(y) + str(z)
...
>>> foo(5,' is ','similar to 5.0')
Parameter '2' has value 'similar to 5.0', not of type '<class 'float'>'
'5 is similar to 5.0'
```

The third decorator **`bb_type_check`** (for “bounded-blocking type check”) enriches **`type_check`** in two ways: 1) if there is at least one disagreement between a value type and a type hint for one parameter, the function is *blocked*, i.e. it is not invoked at all and the decorated function returns **`None`**, without checking the result type; 2) the decorator can block the function at most **`max_block`** times, where **`max_block`** is a parameter of the decorator; after that, the function is invoked even if there is a type mismatch for one parameter, then the type of the result is checked against the type hint, if any, and the function result is printed. Example of use of the decorator (see file **`log3.txt`** for more examples):

```
>>> from type_checking_decorators import bb_type_check
>>> @bb_type_check(2)
... def add(x:int,y:float)-> str:
...     return x + y
...
>>> add(2.3,2.3)
Parameter '0' has value '2.3', not of type '<class 'int'>'
Function blocked. Remaining blocks: 1
>>> add(2.3,2.3)
Parameter '0' has value '2.3', not of type '<class 'int'>'
Function blocked. Remaining blocks: 0
>>> add(2.3,2.3)
Parameter '0' has value '2.3', not of type '<class 'int'>'
Result is '4.6', not of type '<class 'str'>'
4.6
```

Solution format: A single file called **`type_checking_decorators.py`** containing the definition of the decorators **`print_types`**, **`type_check`** and **`bb_type_check`**, such that

- Running **`python3 testN.py`** (with N ∈ {1,2,3}) in a directory containing **`type_checking_decorators.py`** does not raise any exception/error, and
- The output of **`python3 testN.py`** (with N ∈ {1,2,3}) is essentially the same of the corresponding file **`logN.txt`**.