Functional Programming – Series 2

Exercise 1. Define the following functions which should work the same as the corresponding originals in Haskell: `myfilter`, `myfoldl`, `myfoldr`, `myzipWith`. Also give their types.

Exercise 2. Persons are registered in a database by their: name, age, sexe, place of residence. Assume that a person is uniquely determined by his/her name.

a. Give the type of such a database in which the data of each person are stored as a four-tuple. Create an example database of this type to test the functions you have to write below.

b. Define functions to extract the separate data of one person from a four-tuple.

c. Write in *three* ways — with recursion, with list comprehension, and with higher order functions — a function to increase the age of all persons with n years.

d. Write — again in three ways — a function that yields the names of all women between 30 and 40 years.

e. Write a function — one way is sufficient — which yields the age of a person who is given by his/her name. It should be possible that the name contains small letters and/or capital letters (import *Data.Char* for the functions *toLower*, *toUpper*).

f. Sort the database by age. You may not use predefined Haskell functions such as *sortWith*, only *sort* is allowed (from the module *Data.List*). Note that *sort* also works for tuples.

Exercise 3.

a. The *Sieve of Erathostenes* is a technique to generate the list of all prime numbers. It works as follows: it starts with the list of all natural numbers, starting with 2. Take the first number and remove from the rest of the list all multiples of this first nymber. Repeat this procedure for the first number from the remaining list, etcetera.

Write a function *sieve* that implements the Sieve of Erathostenes.

Next, write some functions which use the function *sieve*:

1. a function which tests whether a given natural number is a prime number,
2. a function which delivers the first n prime numbers,
3. a function which yields all prime numbers smaller than n.

Remark. Note that these functions use infinite lists and lazy evaluation.

b. Define a function *dividers* which yields the list of all dividers of a given natural number m. Using this function, define an alternative function to determine whether a given number is prime.

Exercise 4. A three-tuple (a, b, c) is a *Pythagorean Triple* if $a^2 + b^2 = c^2$. Well known examples of such triples are (3,4,5) and (5,12,13).

a. Write a function *pyth* which generates all Pythagorean Triples (a, b, c) such that all a, b, c are smaller than a given natural number n.

Check whether your function still works if you drop the requirement "smaller than n".

b. Write a variant *pyth′* of *pyth* which delivers the list of those Pythagorean Triples that are essentially different. For example, (3,4,5) and (4,3,5) are the same triple. Also (3,4,5) and (6,8,10) are essentially the same triple.

Exercise 5.

a. A list of numbers is *increasing* if every next number in the list greater is than the previous number. Write a function *increasing* which checks whether a list is increasing.

b. A list is *weakly increasing* if every next number in the list is greater than the average of all previous numbers. Write a function *weakIncr* which checks whether a list is weakly increasing.

Exercise 6. We start with some definitions:

- A list *xs* is called a *sublist* of a list *ys* if all elements of *xs* occur *consecutively* in *ys*.
- A list *xs* is called a *partial* sublist if all members of *xs* occur in *ys* in the same order as they occur in *xs*, but not necessarily consecutive.

For example, the list $[2, 4, 6]$ is a sublist of the list $[0, 2, 4, 6, 8]$ and a partial sublist of the list $[1, 2, 3, 4, 5, 6, 7]$.

Note that every sublist is a partial sublist as well, but the opposite need not be the case.

Exercises:

a. Write a function *sublist* which checks whether a list *xs* is a sublist of a list *ys*,

b. Ibid for partial sublist.

Exercise 7. In this exercise you have to write five different sorting algorithms:

a. The *bubble sort* algorithm sorts a list by repeatedly going through the list, while on the way comparing and swapping two consectuive elements if they are in the wrong order. After one pass through the list, the largest element will be "bubbled" to the end.

In the next pass through the list, this last element may be ignored, etcetera. The list is fully sorted if the remaining list to be bubbled does not contain more than one elemnet anymore.

Write a function *bsort* which implements this process, using a funtion *bubble* which implements a single pass through the list.

Possible optimization (not compulsory): sorting is ready when no swapping of two elements is necessary anymore.

b. The *min-max* algorithm takes the minimum and the maximum of a list and puts them in front and at the end (respectively). Repeating this process for the list from which the minimum and the maximum are removed, will sort the list.

Write a function *mmsort* which implements this algorithm.

Hint: use the function \\ from the module *Data.List*.

c. The algorithm *insertion sort* builds up the sorted list as a separate list next to the list that has to be sorted. Initially this eparate list is empty, and the sorting process is performed by inserting the elements from the original list one by one into this separate list.

Write a function *isort* which implements this algorithm by first writing a function *ins* which inserts one element into an already sorted list and by exploiting *foldl* or *foldr*.

Remark: you may not use the already existing function *insert* from *Data.List*).

d. The *merge sort* algorithm splits a list in two halves, sorts these two halves, and merges them together again. Sorting both halves should be done in the same way, i.e., recursively.

First write a function *merge* that merges two already sorted lists into a sorted list, and then write a function *msort* that implements the merge sort algorithm, using the function *merge*.

e. The *quick sort* algorithm splits a list into two lists: one contains the elements that are smaller than or equal to the head of the list, the other contains the elements that are bigger than the head of the original list. Repeating this process to both resulting lists and concatenating them together again will sort the original list.

Write the function *qsort*.