Higher-order functions
Lab 4

1 Currying

We will explain the term *currying* with an example. Consider the functions `f` and `g` defined as follows:

```
f :: (Int, Int) -> Int
f (x,y) = x + y

g :: Int -> Int -> Int
g x y = x + y
```

Notice that the functions `f` and `g` have similar behaviors, yet they are different.

In Haskell, all functions have only one argument. The function `f` takes a single argument (a tuple) and returns an element of type `Int`:

```
*Main> :t (f (2,3))
(f (2,3)) :: Int
```

In Haskell, when specifying the type of a function we must keep in mind that the symbol `->` is right associative. Thus, by `g :: Int -> Int -> Int` we actually mean `g :: Int -> (Int -> Int)`. So the function `g` also receives a single argument (an integer) and returns a *function* of type `Int -> Int`

```
*Main> :t (g 2)
(g 2) :: Int -> Int
```

Further, this new function takes an integer as an argument and returns an integer as well:

```
*Main> :t ((g 2) 3)
((g 2) 3) :: Int
```

The function `g` is the *curried* form of the function `f`. This form is preferred in Haskell because it allows partial application of functions. In Haskell all functions are considered to be of the form *curried*.

Exercițiul 1.1. Write the *curried* variant for the function:

```
addThree :: (Int, Int, Int) -> Int
addThree (x,y,z) = x + y + z
```

2 Higher Order Functions

We saw in the previous section that functions in Haskell can return other functions. Furthermore, functions can take other functions as arguments. The function `process` below takes as arguments a function of type `Int -> Int` and an integer. It applies the function given as an argument over the integer and returns another integer:

```
process ::  (Int -> Int) -> Int -> Int
process f x = f x
```

A possible function call is:

```
 *Main> process (+ 2) 4
6
```

Another possibility to define the function `process` is as follows:

```
process ::  (a -> a) -> a -> a
process f x = f x
```

Here, `a` is a *type variable* that can be instantiated with any type:

```
 *Main> process (+ 2) 4
6
Main> process (&& True) False
False
```

The function is first called on arguments of types `Int -> Int` and `Int`. On the second call, the arguments are of types `Bool -> Bool` and `Bool`.

Exercițiul 2.1. Write a function that is of type `(Int -> Int) -> Int -> Int -> Int` and applies the function of type `Int -> Int` to all values between two integers given as arguments. The function will return the sum of the obtained values.

Exercițiul 2.2. Write a function that returns the composition of two functions.

Exercițiul 2.3. Write a function that receives as a parameter a list of functions and returns their composition.

Exercițiul 2.4. Write a function that calculates the sum of the elements in a list. Use the predefined list data type in the standard library.

Exercițiul 2.5. Write a function that applies a function to each element of a list and returns the resulting list.

Exercițiul 2.6. Write a function that will return the list of elements for which a function of type `a -> Bool` returns `True`.

Exercițiul 2.7. Write a function that implements the fold behavior `(foldr, foldl)` on the list defined in the previous lab.

Exercițiul 2.8. Write three functions, which receive as input parameters the root of a binary search tree and a function (`f`), which will be applied to each node in the manner `preorder, postorder, inorder`. The functions will return a list of the results of applying the function `f`. Use the binary search tree structure defined in the previous lab.

Exercițiul 2.9. Building on the previous exercise, write a **single** traversal function for a binary search tree that receives the traversal strategy (inorder, postorder, preorder, any-order) as a function.

3 Sort by comparison

Exercițiul 3.1. Implement a comparison-based sorting algorithm that receives as arguments:

1. a list `:: [a]` of elements to sort;
2. a `:: a -> a -> Bool` function to compare two elements.

You can choose which sorting algorithm you want. Don't focus on efficiency. Choose the meaning of the comparison function in a reasonable way.

Exercițiul 3.2. Implement `http://hackage.haskell.org/package/base-4.12.0.0/docs/Data-Either.html`.

Exercițiul 3.3. Implement the binary search tree discussed in the lab previously so that it can contain any type of data (which does part of the class `Ord`).

Exercițiul 3.4. Write a function that solves the search problem (sequential, binary), classically and using `foldr/foldl`.

4 Bonus: TABA

TABA (there and back again) is a programming paradigm that allows writing functions in a more efficient way than in the usual mode, by avoiding the construction of additional data structures.

Exercițiul 4.1. Write a function `fromend` that receives a list `L` and a natural number `n` that calculates the `n`-th element of the list `L`, counting from the end towards the beginning.

```
> fromend [1, 7, 5] 0
   Just 5
   > fromend [1, 7, 5] 1
   Just 7
   > fromend [1, 7, 5] 100
   Nothing
```

Exercițiul 4.2. Write a function `convolute` that receives two lists `L1` and `L2` and constructs their convolution, with the list `L2` reversed.

```
> convolute [1, 7, 5] [1, 2, 3]
   [(1, 3), (7, 2), (5, 1)]
```

Here's a way to write a function similar to `fromend`, which has the advantage that it only performs a single traversal of the list.

```
fromendaux :: [a] -> Int -> (a, Int)

fromendaux [x] index = (x, 0)
fromendaux (x:xs) index = let (x', index') = fromendaux xs index in
                            if index' == index then
                              (x', index')
                            else
                              (x, index' + 1)

fromend :: [a] -> Int -> Maybe a
fromend [] = Nothing
fromend (x:xs) index = let (x', index') = fromendaux (x:xs) index in
                         if index == index' then
                           just x'
                         else
                           Nothing
```

Exercițiul 4.3. Write an implementation of the `convolute` function that works similarly (in a single pass).

Exercițiul 4.4. Express the two functions (`convolute` and `fromend`) with the help of a `fold`. The result returned by `foldl` can be post-processed (in $O(1)$).