# Lab Exercise 3 (Higher Order Functions for List Comprehension)

## Task

After familiarizing yourselves with haskell functions, lets move on to list comprehension functions. These functions are probably functional programming's most well-known contribution to other paradigms. Although these functions are already built in inside haskell, you are going to make your own versions of it. **Create a haskell file (".hs") containing the following functions below.** One of them (my\_filter) is already written for you but please still include this function in your haskell file.

### Lists in Haskell

Before you start implementing the functions in this exercise, you need to understand how lists work in Haskell. Haskell lists are written like a list in math. The Haskell list:

[1,2,3,4,5]

is basically equivalent to the list:

One of the most important things about lists is that you can access the list as a whole and you can access the elements inside the list. One thing you can do is you can concatenate lists using the **++** function:

ghci> [1,2,3,4,5] ++ [6,7,8]  
[1,2,3,4,5,6,7,8]

You can find the length of the list using the **length** function:

ghci> length [1,2,3,4,5]  
5

These are the different ways you can access the elements of the list and the sublists in the list:

**head** takes a list and returns the first element of the list

ghci> head [1,2,3,4,5]  
1

**tail** takes a list and returns the same list except the first element of the list

ghci> tail [1,2,3,4,5]  
[2,3,4,5]

**init** takes a list and returns the same list except the last element of the list

ghci> init [1,2,3,4,5]  
[1,2,3,4]

**last** takes a list and returns the last element in the list

ghci> last [1,2,3,4,5]  
5

Using these functions you can traverse a list using the head-tail recipe. For example, if you want to add 1 to each of the elements of the array:

addone :: [Int] -> [Int]  
addone l =   
 if length l == 0 then []  
 else [(head l) + 1] ++ addone (tail l)

Let's dissect this function one by one, for the first line you can see the type signature [Int]->[Int] meaning addone accepts a list of Ints and returns a list of Ints. Any type a surrounded by brackets is a list of as ([a] is a list of a's, [Char] is a list of Chars, [[Int]] is a list of [Int]s or a list of list of Ints).

In the second line were binding the list we are passing to l.

The third line refers to the base case, this happens when the list is empty. When writing the base case, think about the most simple possible list the function may be applied to. The simplest case would be an empty list. When the list is empty we return [] which refers to an empty list.

And finally the last line refers to the general case. Here we are see the subexpression [(head l) + 1] which is a list containing one element namely, the first element of the list plus 1. And then we are concatenating this one element list to the result of the call addone (tail l) which is a recursive call to the rest of add one to the tail of l (or the rest of l). Assuming addone works perfectly, the recursive call addone (tail l) will return the tail of the l but the elements are added with one. By concatenating [(head l) + 1] with the result of this recursive call, we complete the desired result.

* Implement the higher order functions, my\_map,my\_filter, and my\_foldl and my\_foldr, my\_zip
  + **my\_map :: (a -> b) -> [a] -> [b]** - The map function accepts a function and a list (with elements of type A) . It returns the list (with elements of type B): . The new list map produces is a list which is the image of l from the function f.
  + **my\_filter :: (a -> Bool) -> [a] -> [a]** - The filter function accepts a predicate and a list . filter returns a new list such that the contents satisfy is true, retaining the order it appears in .
  + BONUS (here's my\_filter solved for you, use this as a guide):
  + my\_filter :: (a -> Bool) -> [a] -> [a]  
    my\_filter p l =   
     if length l == 0 then []  
     else (if p (head l) then [head l] else []) ++ my\_filter p (tail l)
  + The notable part of this my\_filter's body is the last. The non-base case clause evaluates the following line
  + (if p (head l) then [head l] else []) ++ my\_filter p (tail l)
  + The first part is the if-then-else clause (if p (head l) then [head l] else []) which evaluates to either the list containing the first element of l ([head l]) or an empty list ([]). If the first element (head l) satisfies the predicate p (therefore the if clause contains the expression p (head l) ), then the if-then-else clause evaluates to [head l] otherwise it evaluates to []. Whatever, the if-then-else clause evaluates to is then concatenated to the result of the recursive call to the tail of l (my\_filter p (tail l)). Every time the function recurses, the first element is either concatenated or not concatenated to the rest of the list, this filtering out all elements that do not satisfy the predicate.
  + **my\_foldl :: (a -> a -> a) -> a -> [a] -> a** - The foldl function accepts a function , a list and an initial value . The foldl function returns the value .
  + **my\_foldr :: (a -> a -> a) -> a -> [a] -> a** - The foldr function accepts a function , a list and an initial value . The foldr function returns the value
  + **my\_zip :: (a -> b -> c) -> [a] -> [b] -> [c]** - THe zip function accepts a function and two lists , and returns a new list,
* Without using loops (use the functions above instead), write the following functions.
  + Given a list of numbers, return the sum of the squares of the numbers
  + Given three lists, a list of first names, A, a list of middle names B, and a list of surnames C. Return a list of whole name strings (list of chars) ([firstname] [middle initial]. [lastname]) where the length of the string (including spaces and period) is an even number. Example
  + ghci> wholeName ["Foo", "Bar", "Foo"] ["Middle", "Center", "Name"] ["Lastn", "Surname", "Abcd"]  
    ["Foo M. Lastn", "Bar C. Surname"]
  + ("Foo N. Abcd" is filtered out because it has 11 characters)

## Assessment Criteria

* Completeness of haskell functions - 35

**Deadline January 11, 2022**