# Project 3. Finger exercises: implementing Haskell (and other) functions in Haskell

### [Learn You a H](http://learnyouahaskell.com/chapters)askell

[6. Higher Order Functions (2nd half)](http://learnyouahaskell.com/higher-order-functions)

[7. Modules](http://learnyouahaskell.com/modules).

[8. Making your own types and typeclasses](http://learnyouahaskell.com/making-our-own-types-and-typeclasses)

### Videos: ([link to all Drake videos](https://www.youtube.com/playlist?list=PLAYqRAte9oRIChcPR_DD4uc8mCR6d3RiJ))

[7. Modules (4:41)](https://www.youtube.com/watch?v=OEC_XbzXmYY&list=PLAYqRAte9oRIChcPR_DD4uc8mCR6d3RiJ&index=12http://www.youtube.com/watch?v=vuT8ts_neZw)

[8a.. Defining](http://www.youtube.com/watch?v=OfxCm_OarIg) types (5:15)

[8b.. Defining](http://www.youtube.com/watch?v=OfxCm_OarIg) typeclasses (5:49)

**Total**: 15:45

In this project you will (re-)implement some existing Haskell functions (and a **while** loop) in Haskell.

1. For example, here are two ways to implement map using recursion. You should be able to explain how they both work.

-- The first version uses straightforward recursion.

myMap1 :: (a -> b) -> [a] -> [b]

myMap1 \_ [] = []

myMap1 f (x:xs) = (f x) : myMap1 f xs

> myMap1 (\x -> x+3) [1, 2, 3]

[4,5,6]

**Question:** What would happen if we reversed the order of the two myMap1 clauses?

**Question:** Is myMap1 tail recursive? Why or why not?

-- The second version implements map in terms of foldr.

myMap2 :: (a -> b) -> [a] -> [b]

myMap2 f xs = foldr (\x accum -> (f x : accum)) [] xs

> myMap2 (\x -> x+3) [1, 2, 3]

[4,5,6]

**Question:** Show how this works by stepping through the execution.

Implement the following functions in Haskell. Construct test cases to demonstrate correctness.

1. myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]  
   **Possible question:** If you use recursion, does the order of the clauses matter? Why or why not?
2. myFoldl :: (b -> a -> b) -> b -> [a] -> b  
   Use recursion and apply the function as an argument to the recursive call.  
   In other words, compute the accumulator as you walk down the list recursively.   
   **Question:** Is this tail recursive. Why or why not?   
   **Question:** What is the relationship between the value produced by the base case and the initial function call? That is, assume you make a call like this:  
   > myFoldl fn accInit list  
   and assume that when the base case is reached it returns value  
   What is the relationship (if any) between value and myFoldl fn accInit list?
3. myFoldr :: (a -> b -> b) -> b -> [a] -> b   
   Solve this in two ways,
   1. Use recursion. Compute the accumulator as you *return* from the recursion.

**Question:** Is this tail recursive. Why or why not?

**Question:** What is the relationship between the value produced by the base case and the initial accumulator value in the initial function call?

* 1. Reverse the list and then use foldl or myFoldl. You will find the function flip useful. If you want to use it, write your own myFlip function and use that.

1. myCycle :: [a] -> [a]   
   The recursive implementation has no base case.

**Question:** Such a situation would produce an infinite loop in Java. Why doesn’t this lead to an infinite loop in Haskell?

**Question:** What happens with the following? Explain why.

> cyc12 = myCycle [1,2] - - Does this result in an infinite loop?  
> take 5 cyc12 - - Does this result in an infinite loop?

**Question:** Walk through the step-by-step evaluation of

> take 5 cyc12

You may assume that take is implemented as follows

take 0 \_ = []

take \_ [] = []

take n (x:xs) = x : take (n-1) xs

and that (++) is implemented as follows.

[] ++ ys = ys

(x:xs) ++ ys = x:(xs ++ ys)

Here are two examples of what I want you to do.

1. The sequence of steps for take 5 [1, 2, 3] ++ [4, 5, 6] is:

> take 5 [1, 2, 3] ++ [4, 5, 6] – apply the recursive ++ case to get:

* take 5 (1 : ([2 : 3] ++ [4, 5, 6])) -- apply the recursive take case to get:
* 1 : take 4 ([2 : 3] ++ [4, 5, 6]) -- apply the recursive ++ case to get:
* 1 : take 4 (2 : ([3] ++ [4, 5, 6])) -- apply the recursive take case to get:
* 1 : 2 : take 3 ([3] ++ [4, 5, 6]) -- apply the recursive ++ case to get:
* 1 : 2 : take 3 (3:([] ++ [4, 5, 6])) -- apply the recursive take case to get:
* 1 : 2 : 3 : take 2 ([] ++ [4, 5, 6]) -- apply the base ++ case to get:
* 1 : 2 : 3 : take 2 [4, 5, 6] -- apply the recursive take case to get:
* 1 : 2 : 3 : 4 : take 1 [5, 6] -- apply the recursive take case to get:
* 1 : 2 : 3 : 4 : 5 : take 0 [6] -- apply the base take case to get:
* 1 : 2 : 3 : 4 : 5 : [] -- Nothing left to evaluate. We’re done.

It is because Haskell uses this evaluation strategy that it is considered lazy. This is what lazy means: perform a step in expression evaluation only when doing so is the highest level action that can be taken at that point.

1. Assume you want to find the smallest value in a list — and that you don't have a general min or foldl function. One approach would be to sort the list and take the first element.

> head $ insertionSort [7, 3, 4, 1, 3, 6, 3, 4]

1

That sounds very inefficient—especially since insertion sort is O(n2). But with lazy evaluation, it may be a reasonable choice.

-- We can define insertionSort as follows. (We will use the

-- auxiliary function iSort to give us accumulator.)

insertionSort :: Ord a => [a] -> [a]

insertionSort xs = iSort xs []

**where**

iSort :: Ord a => [a] -> [a] -> [a]

iSort (x:xs) sorted = iSort xs $ insert x sorted

iSort [] sorted = sorted

-- Insert an element into a list.

insert :: Ord a => a -> [a] -> [a]

insert elt [] = [elt]

insert elt (x:xs)

| elt < x = elt:x:xs

| otherwise = x : insert elt xs

Here are the steps in evaluating:

> head $ insertionSort [7, 3, 4, 1, 3, 6, 3, 4]

Haskell performs each evaluation step at the highest possible level. In the evaluation below I’ve highlighted the function that is applied to get to the next line.

* **The > line to line a):** the insertionSort clause is applied.
* **Line a) to line i):** the first iSort clause is applied repeatedly.
* **Line i) to line j):** the second iSort clause is applied.
* **Line j) to line k):** the first insert clause is applied.
* **Line k) to line r):** the second insert clause is applied repeatedly.
* **Line r) to line s):** the head function is applied.

> head $ insertionSort [7, 3, 4, 1, 3, 6, 3, 4]

1. head $ iSort [7, 3, 4, 1, 3, 6, 3, 4] []
2. head $ iSort [3, 4, 1, 3, 6, 3, 4] (insert 7 [])
3. head $ iSort [4, 1, 3, 6, 3, 4] (insert 3 (insert 7 []))
4. head $ iSort [1, 3, 6, 3, 4] (insert 4 (insert 3 (insert 7 [])))
5. head $ iSort [3, 6, 3, 4] (insert 1 (insert 4 (insert 3 (insert 7 []))))
6. head $ iSort [6, 3, 4] (insert 3 (insert 1 (insert 4 (insert 3 (insert 7 [])))))
7. head $ iSort [3, 4] (insert 6 (insert 3 (insert 1 (insert 4 (insert 3 (insert 7 []))))))
8. head $ iSort [4] (insert 3 (insert 6 (insert 3 (insert 1 (insert 4 (insert 3 (insert 7 [])))))))
9. head $ iSort [] (insert 4 (insert 3 (insert 6 (insert 3 (insert 1 (insert 4 (insert 3 (insert 7 []))))))))
10. head $ insert 4 (insert 3 (insert 6 (insert 3 (insert 1 (insert 4 (insert 3 (insert 7 [])))))))
11. head $ insert 4 (insert 3 (insert 6 (insert 3 (insert 1 (insert 4 (insert 3 [7]))))))
12. head $ insert 4 (insert 3 (insert 6 (insert 3 (insert 1 (insert 4 (3:[7]))))))
13. head $ insert 4 (insert 3 (insert 6 (insert 3 (insert 1 (3:(insert 4 [7]))))))
14. head $ insert 4 (insert 3 (insert 6 (insert 3 (1:3:(insert 4 [7])))))
15. head $ insert 4 (insert 3 (insert 6 (1:(insert 3 (3:(insert 4 [7]))))))
16. head $ insert 4 (insert 3 (1:(insert 6 (insert 3 (3:(insert 4 [7]))))))
17. head $ insert 4 (1:(insert 3 (insert 6 (insert 3 (3:(insert 4 [7]))))))
18. head $ 1:(insert 4 (insert 3 (insert 6 (insert 3 (3:(insert 4 [7]))))))
19. 1

The computational complexity is O(n): walk down the list using iSort and then walk back up using insert. As we walk back up the list, insert “bubbles” the smallest value to the top of the expression—which is the first element of the list—at which point head extracts it—even though the rest of the expression/list is not yet evaluated.

1. Function composition: the compose function.

g `compose` f should be the same as g . f

compose :: (b -> c) -> (a -> b) -> (a -> c)

Be sure you have the right type: two functions as argument; a function as the result.

If h = g `compose` f then h x = (g . f) x = g (f x)

**Question:** Given g:: b -> c, f:: a -> b, and h = g `compose` f,  
what is the type of h? (**Hint:** the type of h is *not* the same as the type of compose.)

1. Write a function functionPairs that takes a function f and returns a function that when given a list of numbers returns a list of pairs, whose first element is from the input list and whose second element is the function f applied to the first element. (You may assume that f :: Int -> Int). For example,

> let f1 x = x^2

> let sqrPairs = functionPairs f1

> sqrPairs [1 .. 10]

[(1,1),(2,4),(3,9),(4,16),(5,25),(6,36),(7,49),(8,64),(9,81),(10,100)]

> let f2 x = x^3 – x^2 + x

> let cubeMinusX2PlusXPairs = functionPairs f2

> cubeMinusX2PlusXPairs [1 .. 9]

[(1,1),(2,6),(3,21),(4,52),(5,105),(6,186),(7,301),(8,456),(9,657)]

When you define functionPairs, be sure you declare its type.

Construct two answers. Use zip in one and map in the other. (The zip solution also uses map. The map solution does not use zip.)

1. Implement a **while** loop.

**while** loops usually look something like this. (In the following, *state* refers to the variables the **while** loop uses in its body and when testing whether to continue the loop.)

init state -- Initialize the state.

**while** (shouldContinue state) -- while some condition holds on the state,

{ -- run the body of the loop on the state

bodyFn state -- producing a new state.

} -- Return a result extracted from the final

-- value of the state.

Your **while** function will take three arguments:

1. the current value of the *state* (the initial value when called);
2. the *shouldContinue* test, to be applied to the *state*; and returning a Bool;
3. the *bodyfn*, to be applied to the *state* and returning a *state*.

Your **while** function will have this type.

while :: state -> -- The initial value of the state

(state -> Bool) -> -- A shouldContinue test

(state -> state) -> -- A bodyFn

(state -> result -> -- A function to build the return value

result -- The return value

**Question:** Is your **while** function tail recursive. Why or why not?

The following example generates a list of the first n integers squared.

* The *state* is a pair whose first element is the index and whose second element is the list we are generating. Its initial value is (1, []).
* The *shouldContinue* function returns *True* or *False* depending on whether the *index* is less than or equal to *n*.
* The *bodyFn* adds *index^2* to the front of the list of squares and increments the *index*. (We generate the list in reverse. At the end we reverse it.)

nSquares:: Int -> [Int]

nSquares n =

while (1, [])

(\(index, \_) -> index <= n) -- n is the nSquares argument.

(\(index, list) -> (index + 1, index^2 : list))

(reverse . snd) -- Extract the second element of

-- the tuple and reverse it.

> nSquares 15

[1,4,9,16,25,36,49,64,81,100,121,144,169,196,225]

**Question:** Explain how nSquares works. What is the state? What do its components represent? (The answers are above.)

**Extra credit (one grade step)**

1. Use your **while** function to implement map. As sketched below, we expect **while** to return a pair whose first element is the reverse of the answer.

myMap3 :: (a -> b) -> [a] -> [b]

myMap3 f xs =

while <initialState> <shouldContinueFn> <bodyFn> <returnValueFn>

> myMap3 (\*2) [1, 2, 3]

[2,4,6]

1. Use your **while** function to implement foldl. As sketched below, we expect **while** to return a pair whose first element is the answer.

myWhileFoldl :: (b -> a -> b) -> b -> [a] -> b

myWhileFoldl f init xs =

while <initialState> <shouldContinueFn> <bodyFn> <returnValueFn>

> myWhileFoldl (\*) 1 [1 .. 5]

120

1. Use your **while** function to generate the first n Fibonacci numbers.

nFibs :: Int -> [Int]

nFibs n = <your definition using while>

> nFibs 10

[1,1,2,3,5,8,13,21,34,55]

1. Use your **while** function to implement the [sieve of Eratosthenes](https://www.khanacademy.org/computing/computer-science/cryptography/comp-number-theory/v/sieve-of-eratosthenes-prime-adventure-part-4https:/en.wikipedia.org/wiki/Sieve_of_Eratosthenes), and use that to generate the first n primes. (Not the same as all primes less than or equal to n.)

nPrimes :: Int -> [Int]

nPrimes n = <your definition using while>

> nPrimes 15

[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]

1. The following list comprehension acts like a loop nested within another loop.

doubleLoop outer inner = [(oIndex, iIndex) | oIndex <- [1 .. outer],

iIndex <- [1 .. inner]]

> doubleLoop 3 5

[(1,1),(1,2),(1,3),(1,4),(1,5),(2,1),(2,2),(2,3),(2,4),(2,5),

(3,1), (3,2),(3,3),(3,4),(3,5)]

Produce the same result using a nested **while**. The nested **while** should appear within the *bodyFn* of the outer **while**. (Once you figure out how to do it, it’s not that complex. But it is a lot messierthan the list comprehension approach!)

doubleWhile outer inner =

while <initialState> <shouldContinueFn> <bodyFn> <returnValueFn>

> doubleWhile 4 6 == doubleLoop 4 6

True - - for all arguments