5 Higher-order functions

Exercise 5.1 (*Warm-up*: Type derivation, `HOF.hs`). Below a number of functions and operators are defined. What types does Haskell derive for them? Check your answer in GHCi!

```
const x y    = x

x $→ y       = y x

oper "mul" n = (*n)
oper "div" n = (n/)
oper _     _ = error "not implemented"

mapMap f xs  = map (map f) xs

without p    = filter (not . p)

on f g x y   = f (g x) (g y)
```

Exercise 5.2 (*Warm-up:* Function composition, `HOF.hs`). The function composition operator `.` creates a new function out of two existing ones. It can be defined as: `(f . g) x = f (g x)`. Explain what the following compositions do:

```
f1 = (* 5)     . (+ 1)
f2 = (+ 1)     . (* 5)
f3 = (min 100) . (max 0)
f4 = (<5)      . length
```

What changes if we change `f4` to "`(5<) . length`"? What about "`((<)5) . length`"?

Exercise 5.3 (*Warm-up*: Function parameters, `ThereCanBeOnlyOne.hs`).
This exercise illustrates a generalization pattern that is available in all useful programming languages, not just functional languages.

1. Define a function `onlyElem :: (Eq a) ⇒ a → [a] → Bool`, which test whether a given element occurs once, and only once, in a list.

2. Generalize this function to `onlyOnce :: (a → Bool) → [a] → Bool`, which tests whether a given predicate is true for one, and exactly one, element in a list:

   ```
   onlyOnce odd  [1..3] ⟹ False
   onlyOnce even [1..3] ⟹ True
   ```

3. `onlyOnce` is a more general function than `onlyElem`: Redefine `onlyElem` so it is expressed in terms of `onlyOnce`.

Exercise 5.4 (*Warm-up*: Any and all).
The list functions `any :: (a → Bool) → [a] → Bool` and `all :: (a → Bool) → [a] → Bool` are often useful. `any` tests whether a given predicate is true for some element in a list, and `all` tests whether it is true for all elements in the list.

These functions correspond to the $\exists x[Px]$ and $\forall x[Px]$ quantifiers. And of course you remember DeMorgan's laws for quantifiers: $\forall x[\neg Px] \equiv \neg\exists x[Px]$

1. Define a function `all'` which is equivalent to `all`, **using only** the standard functions `any`, `not`, the operator `(.)`, and function application.

2. If your definition looks like `all' p xs = ...`, can you change it so it looks like `all' p = ...` **without** using a lambda expression? Which style of definition do you prefer?

Exercise 5.5 (*Warm-up*: Folding exercises, `Folders.hs`). Use `foldl` or `foldr` (whichever is convenient) to re-implement the following functions. **Do not use** direct recursion.

1. `and :: [Bool] → Bool`, that determines whether all elements in a list of Booleans are true;

2. `or :: [Bool] → Bool`, that determines whether there exists a Boolean in a list that is true;

3. `elem :: (Eq a) ⇒ a → [a] → Bool`, that tests whether an element occurs in a list;

4. `maximum :: (Ord a) ⇒ [a] → a`, that calculates the largest value in a list of element. (An empty list has no maximum, so you are allowed to generate a runtime error for that case.)

5. `fromList :: (Ord a) ⇒ [a] → Tree a`, from Exercise 4.4, which constructs a tree from the elements in the provided list by repeatedly calling insert .

6. `fromBits :: [Integer] → Integer`, which converts a bit representation of an integer in *least significant bit first* order (encoded as a list of ones and zeroes) to the corresponding integer, so:

   `fromBits [1,1,0,0,1,0,0,1] ⟹ 147`

If both recursion schemes are applicable, which do you prefer? (See Exercise 5.9!)

Exercise 5.6 (*Mandatory:* Functions as objects, folds, `FunList.hs`).

1. Define the function `compose :: [a → a] → a → a`, which composes a list of functions into a single function, so, for example:

   `compose [f0,f1,f2] x = f0 (f1 (f2 x))`

   Define it twice: once using the *list design pattern*, and once using `foldr`. Use the name `compose'` (pronounced "compose *prime*") for the second definition.

2. Explain *what* the following function computes, and *how* it computes it:

   ```
   foo n = compose (map (*) [1..n]) 1
   ```

3. Define a function `foldr' :: (a → b → b) → b → [a] → b`, that is equivalent to `foldr`, but defined in terms of **only** `map` and `compose`.

Exercise 5.7 (*Mandatory:* Unfolds, `Unfold.hs`).
Use the higher-order function `unfoldr` to define: (Also see Hint 1)

1. a function `bits :: Int → [Int]` that returns the binary representation of a non-negative integer, with the *least significant bit* first, so:

   ```
   binary 147 ⟹ [1,1,0,0,1,0,0,1]
   ```

   (In this case, `bits` is the inverse of `fromBits` from Exercise 5.5.)

2. a function `zip :: [a] → [b] → [(a,b)]` implementing the function `zip` from the Prelude;

3. a function `take :: Int → [a] → [a]` implementing the function `take` from the Prelude;

4. a function `primes :: [Integer]` generating a sequence of all prime numbers. The definition can be based on the following alternative implementation

   ```
   primes = sieve [2..]
     where sieve (p:xs) = p : sieve [ n | n ← xs, n ‘mod‘ p /= 0 ]
   ```

Some list producers can return "early", that is, at some point they are able to determine completely what the rest of the list being produced should look like. Producers defined using `unfoldr`, however, *always* iterate until the supplied production function returns `Nothing`, i.e. when there are *no more elements* to produce.

For example, the "append" function could return its second argument early once it has gone through all the elements of its first argument; if you define it with `unfoldr`, however, you need to produce all the elements of the second argument as well (capping off the concatenation of both lists with `[]`).

We can define a slightly more general function, named `apo` (after the formal concept of an *apomorphism*):

```
apo :: (t → Either [a] (a, t)) → t → [a]
apo f seed = case f seed of
               Left  l      → l
               Right (a,ns) → a : apo f ns
```

Instead of returning a `Maybe`-value, the argument function of `apo` now returns an `Either` value. Termination occurs when a value `Left l` is produced. The difference with `unfoldr` is that in this case `apo` will return `l` instead of `[]`. The other case is the same as the case for `Just` in `unfoldr`.

5. Use `apo` to define the append function, `(++)`;

6. Use `apo` to redefine the function `insert` that inserts a given element in an already sorted list, i.e.;

   ```
   insert :: (Ord a) ⇒ a → [a] → [a]
   insert x [] = [x]
   insert x (y : ys)
    | x <= y     = x : y : ys
    | otherwise  = y : insert x ys
   ```

7. Give a definition of `unfoldr` in terms of `apo`;

8. *Optional:* Give a definition of `apo` in terms of `unfoldr`;

Exercise 5.8 (*Mandatory*: Functional pipelines, `WordStats.hs`).
During a tutorial, we have discussed that the function `wordFrequency :: String → [(String,Int)]` counting the number of occurrences of words, can be neatly implemented as:

```
wordFrequency = map (\x→(head x,length x)) . group . sort . words
```

This kind of function composition is similar to the UNIX concept of a *functional pipeline*: one function consumes the output of another. (Except here the pipeline is written from right to left.)

Define the following functions, and provide type specifications for them. Use pipelines and higher order functions. It would probably be too time-consuming otherwise! Functions from the `Data.List` module will be useful: https://hackage.haskell.org/package/base/docs/Data-List.html

1. `mostFrequentOfLength`, similar to `wordFrequency`, but only listing words that are of certain minimal length (provided as a parameter), in descending order of frequency.

2. `wordLengthFrequency`, which counts how frequently certain *word lengths* occur and lists the lengths and their frequencies in ascending order of length. So:

   `wordLengthFrequency "hallo hallo hello and goodbye"` ⟹ `[(3,1),(5,3),(7,1)]`

   (This can be used to empirically test Zipf's Law of Abbreviation)

3. `anagrams`, which lists all words that are anagrams of each other in a a string. Only list words *once*, and don't display words that are not anagrams of any other word. For example:

   ```
   >>> anagrams "the eastern spot is the nearest stop in earnest"
   [["eastern","nearest","earnest"],["spot","stop"]]
   ```

You can test your functions on realistically large inputs, by compiling the template file using GHC and actually running it; it will read its input from standard input:

```
$ ghc WordStats.hs
./WordStats < YOUR_INPUT_FILE
```

Note: The Haskell module `Data.Map` contains a data type commonly used for associative maps. Documentation for it can be found at https://hackage.haskell.org/package/containers/docs/Data-Map.html. Do not confuse `Map k a` (a container data type associating keys `k` with values `a` like a dictionary) with `map` (a higher-order function to apply a function to all values in a container).

Associative maps also support operations that are similar to the ones in `Data.List`. These (intentionally) share the names of the functions they are similar to. To avoid ambiguity, the definitions exported by this module are typically used with an explicit qualifier:

```
>>> import qualified Data.Map as Map
>>> Map.insert "key" "value" Map.empty
fromList [("key","value")]
>>> Map.lookup "Maastricht" (Map.fromList [("Nijmegen", 89), ("Maastricht", 1203)])
Just 1203
```

4. *Optional:* Re-implement the function `wordFrequency` using a `Data.Map String Int`, using the dictionary to efficiently count words. In particular, you may want to have a look at the functions `toList`, `fromListWith` or `insertWith`.

Exercise 5.9 (*Extra*: Laziness and folds, `ShortCircuit.hs`).
Both `&&` and the `||` inspect first the value of the first argument, and if the second argument is not needed to determine the result, it is not evaluated. So we can safely write things like:

```
infinitesimal x = x == 0 || 1 / x >= 1e10
```

The functions `and` and `or` 'lift' these conditional tests to lists; they can be expressed using both `foldl` and `foldr`; so let's do that:

```
andl = foldl (&&) True
andr = foldr (&&) True
orl  = foldl (||) False
orr  = foldr (||) False
```

1. Predict what will happen when `andl`, `andr`, `orl` and `orr` are applied to an *infinite list* of booleans: Do this using the following examples. Check your predictions!

   ```
   andl $ False : [True,  True  ..]
   andr $ False : [True,  True  ..]
   orl  $ True  : [False, False ..]
   orr  $ True  : [False, False ..]
   ```

2. For *finite* lists, these functions are equivalent to `and` and `or` respectively. But in both cases one definition is preferable over the other. Why?

Exercise 5.10 (*Extra:* Higher-order functions, `ListHOF.hs`). There are many useful functions in `Data.List` that takes other functions as arguments, besides `map` and `filter`, such as `takeWhile`/`dropWhile`, `sortOn`/`sortBy`, `zipWith`, `groupBy`... see https://hackage.haskell.org/package/base/docs/Data-List.html.

With these, define the following: (this shouldn't need a lot of code!)

1. A function `sortLength :: [String] → [String]` which sorts a list of strings based on their length, in ascending order.

2. A function `letterClump :: String → [String]` which groups together all letters, but leaves all other characters as is, so:

   ```
   letterClump "... forty two!" ⟹ [".", ".", ".", " ", "forty", " ", "two", "!"]
   ```

   (And yes, this would have been useful for Exercise 3.8.)

3. The list `fibs :: [Integer]` of all Fibonacci numbers. (If the fact that this list is infinite bothers you, start by defining it for just the first 1000 Fibonacci numbers or so.)

4. The function `zipWith'`, equivalent to `zipWith`, but defined in terms of `map`, `zip`, and `uncurry`.

Exercise 5.11 (*Extra*: Run length encoding, `RLE.hs`).
Run length encoding is a simple encoding scheme for repetitious data. It represents duplicate contiguous elements more efficiently by encoding elements as a tuple containing their value and number of repetitions. For example:

```
"Noooo...!" ⟹ [('N',1), ('o',4), ('.',3), ('!',1)]
```

Define the following functions (**without direct recursion**):

1. `encodeRLE :: (Eq a) ⇒ [a] → [(a,Int)]`, which transforms a list into a run-length encoded representation of the same list.

2. `decodeRLE :: [(a,Int)] → [a]`, which does the reverse.

You can implement these functions by using the "Think Big" approach, cobbling them together from standard functions from the `Data.List` library and function composition. But you can also make both using `foldr` and `unfoldr`.

Hints to practitioners 1. The function `unfoldr` might appear very abstract, but it actually just encodes a form of repetition in programs you should be very familiar with: that of a *productive* loop. A loop is productive if at every iteration it produces something as output, such as in this imperative program:

```
x := 0
while x <= 5:
    output (x*x)
    x := x+1
```

Which is an instance of a more general pattern:

```
x := init
while p(x):
    output f(x)
    x := update(x)
```

This schema is essentially what `unfoldr` captures. The above can be expressed in Haskell as:

```
unfoldr (\x→if p x then Just (f x, update x) else Nothing) init
```

or, equivalently (but perhaps nicer to read):

```
unfoldr go init
  where go x | p x       = Just (f x, update x)
             | otherwise = Nothing
```

Of course, the helper function `go` can be much more interesting than this, consisting of pattern matches, more than two case distinctions, etc., which would quickly become pretty hard to write elegantly using a while-loop in an imperative programming language.

(There is a similar correspondence between a for-loop and `foldl`, but **not** for `foldr`, which can have quite different behaviour—see Exercise 5.9).