Programming I: Functional Programming in Haskell

Unassessed Exercises 4: **Higher-Order Functions** (Solutions)

These exercises are unassessed so you do not need to submit them. They are designed to help you master the language, so you should do as many as you can at your own speed.

There are probably more questions on these sheets than you may need in order to get the hang of a particular concept, so feel free to skip over some of the questions. You can always go back to them later if you need to.

Model answers to each set will be handed out throughout the course.

---

1. Rewrite the following functions so that they use either `map`, `concatMap`, `filter` or one of the family of `fold` functions. Hint: if you can't spot the trick try replacing an operator with its (prefix) form.

   (a)
   ```
   depunctuate :: [Char] -> [Char]
   depunctuate [] = []
   depunctuate (c : cs)
     | elem c ['.', ',', ':'] = depunctuate cs
     | otherwise              = c : depunctuate cs
   ```

   Solution:

   ```
   depunctuate = filter (\c -> not (elem c ['.', ',', ':']))
   ```

   (b)
   ```
   makeString :: [Int] -> [Char]
   makeString []       = []
   makeString (n : ns) = chr n : makeString ns
   ```

   Solution:

   ```
   makeString = map chr
   ```

   (c)
   ```
   enpower :: [Int] -> Int
   enpower [n]      = n
   enpower (n : ns) = enpower ns ^ n
   ```

   Solution:

   ```
   enpower = foldr1 (flip (^))
   ```

   (d)
   ```
   revAll :: [[a]] -> [a]
   revAll []       = []
   revAll (x : xs) = reverse x ++ revAll xs
   ```

> **Solution:**
>
> ```
> revAll = concatMap reverse
> ```

(e)
```
-- Built-in reverse in disguise
rev :: [a] -> [a]
rev xs = rev' xs []
  where rev' [] ys = ys
        rev' (x : xs) ys = rev' xs (x : ys)
```

> **Solution:**
>
> ```
> rev = foldl (\ys x -> x : ys) []
> ```

(f)
```
-- Built-in unzip in disguise
dezip :: [(a,b)] -> ([a],[b])
dezip [] = ([], [])
dezip ((x, y) : ps) = let (xs, ys) = dezip ps in (x : xs, y : ys)
```

> **Solution:**
>
> ```
> dezip = foldr (\(x, y) (xs, ys) -> (x : xs, y : ys)) []
> ```

2. Write a function `allSame :: [Int] -> Bool` which delivers `True` iff all elements of the given list are the same. One way to do this (not necessarily the best) is to ask whether all adjacent elements in the list are the same, i.e. first=second, second=third and so on. If you haven't already done it this way try it out using `and`, `zipWith` and `tail`.

   Hint: what happens if you zip a list with its own tail?

> **Solution:** Zipping two lists will trim to the shorter of the two lists: by zipping a list with its own tail, you will generate the list of all adjacent pairs in the list.
>
> ```
> allSame xs = and (zipWith (==) xs (tail xs))
> ```

3. The prelude function `scanl` takes a function, a base value and a list and forms the list whose n^{th} element comprises the results of folding the given function into the first n elements of the list, using the base value given. For example, `scanl (+) 0 [1,3,5,7,9]` computes the *partial prefix* sums of the list `[1,3,5,7,9]`, i.e. the list `[0,1,4,9,16,25]`. The function `scanr` is defined similarly. You can also experiment with variants `scanl1` and `scanr1` which omit the base case in the same way as `foldl1` and `foldr1`.

   (a) Use a single application of `scanl` to build the infinite list of factorials `[1,1,2,6,24,...]`

Solution: Why `scanl'` here? Well, a regular `scanl` (or `foldl`) will evaluate the accumulator parameter lazily, which in some cases is fine, and in others may build an unnecessary "tower" of work, called *thunks*. The *prime* variants of these functions will evaluate the accumulator strictly at each step, which may be much more efficient!

```
facts = scanl' (*) 1 [1..]
```

(b) Using `scanl`, `map` and `sum`, compute an approximation to e by summing the first five terms in the infinite expansion for e viz.

$$e = \frac{1}{0!} + \frac{1}{1} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

Solution: The `recip` function is the same as saying `(1/)`.

```
e = sum (map recip (take 5 facts))
```

(c) What does the function `mystery = 1 : scanl (+) 1 mystery` represent? Try evaluating `take 6 mystery`. Now explain it!

Solution: `mystery` represents the infinite list of Fibonacci numbers. This is reliant on laziness, and is known as "chasing the tail". Try peeling off a layer one by one to see what happens.

4. Write a function `squash :: (a -> a -> b) -> [a] -> [b]` which applies a given function to adjacent elements of a list, e.g. `squash f [x1, x2, x3, x4] == [f x1 x2, f x2 x3, f x3 x4]`. Implement the function using:

(a) explicit recursion and pattern matching and

Solution:

```
squash :: (a -> a -> b) -> [a] -> [b]
squash f (x : y : ys) = f x y : squash f (y : ys)
squash _ _ = []
```

(b) In terms of `zipWith` and `tail`.

Solution:

```
squash :: (a -> a -> b) -> [a] -> [b]
squash f xs = zipWith f xs (tail xs)
```

5. Write a function `converge :: (a -> a -> Bool) -> [a] -> a` which searches for convergence in a given list of values. It should apply its given convergence function to adjacent elements of the list until the function yields `True`, in which case the result is the first of the two convergent values. For example, `converge (==) [1, 2, 3, 4, 5, 5, 5]` should return `5`. If no convergence

is found before the list runs out, return the last element of the list. A precondition is that the list contains at least one element. Using `converge` and `scanl` (twice), define a (constant) function that will compute e to 5 decimal places.

Solution:

```
converge :: (a -> a -> Bool) -> [a] -> a
-- Pre: list is non-empty
converge f (x : x' : xs)
  | f x x'    = x'
  | otherwise = converge f (x' : xs)
converge _ [x] = x
```

Using this, e, can be be computed as follows (using the `facts` from earlier!):

```
e = converge lim (sums (map recip facts))
  where lim x y = abs (x - y) < 0.00001
        sums = scanl (+) 0
```

6. Write a function `limit :: (a -> a -> Bool) -> [a] -> [a]` which again checks for convergence but this time returns the list elements up to the point where the convergence function delivers `True`. This is a generalisation of the `takeWhile` function. We know that if $0 \leq r \leq 1$ then

$$\sum_{n=0}^{\infty} r^n = \frac{1}{1-r}$$

Use this to test your limit function with an expression of the form `sum (limit f (map (r ^) [0..]))` for some convergence function `f`. If no limit is found return the whole list.

Solution: Simple generalisation of the `converge` function above.

```
limit :: (a -> a -> Bool) -> [a] -> a
limit f (x : x' : xs)
  | f x x'    = [x, x']
  | otherwise = x : limit f (x' : xs)
limit _ xs = xs
```

7. The functions `any, all :: (a -> Bool) -> [a] -> Bool` apply a given predicate to each element of a given list: `any` delivers `True` if one or more applications yields `True` and `all` iff every application yields `True`. Given empty lists they deliver `False` and `True` respectively. Exploiting extensionality[1], define `any` using `or` and `map` and `all` using `and` and `map` so that they are of the form:

```
any p = ...
all p = ...
```

[1]Extensionality means that when `f x = g x` for all `x`, `f = g`. In other words, you can drop redundant arguments.

Solution:

```
any p = or . map p
all p = and . map p
```

As an explanation:

```
any p xs {- unwound definition -}
         = or (map p xs)
         {- bracketing -}
         = or ((map p) xs)
         {- definition of (.) -}
         = (or . map p) xs
any p    {- apply function extensionality -}
         = or . map p
```

8. By exploiting extensionality, define the function `isElem`, equivalent to the built-in `elem` function, using `(==)` and `any` so that it is in *point-free form*, i.e. of the form:

```
isElem = ...
```

Solution:

```
isElem = any . (==)
```

9. Sometimes we would like to compose a single-argument function with a binary function, as in `(succ . (+)) 4 7`, yielding `12`, for example. However, Haskell's `(.)` function will only compose two single-argument functions.

   (a) Define a new composition operator `<.>` that will allow compositions of the above form. What is its most general type?

   Solution:

   ```
   (<.>) :: (c -> d) -> (a -> b -> c) -> (a -> c -> d)
   (f <.> g) x y = f (g x y)
   ```

   (b) Now try writing `<.>` in point-free form. You should find you can write it using just the symbols '(', ')' and '.'.

   Hint: To write expressions involving operators in point-free form it's often useful to express the operators in their prefix form, e.g. `(.)` in the case of composition. You then need to manipulate the resulting expression so that you can cancel the arguments.

Solution:

```
(f <.> g) x y {- base definition -}
              = f (g x y)
(f <.> g)     {- move arguments onto right-hand side with lambda -}
              = \x y -> f (g x y)
              {- bracketing -}
              = \x y -> f ((g x) y)
              {- definition of (.) -}
              = \x -> f . g x
              {- bracketing -}
              = \x -> (f .) (g x)
              {- definition of (.) -}
              = (f .) . g
(<.>)         {- move arguments onto right-hand side with lambda -}
              = \f g -> (f .) . g
              {- bracketing -}
              = \f g -> ((f .) .) g
              {- function extensionality -}
              = \f -> (f .) .
              {- prefix notation on both compositions -}
              = \f -> (.) ((.) f)
              {- definition of (.) -}
              = (.) . (.)
```

(c) Define point-free versions of `any` and `all` in terms of `<.>`, i.e. versions that are of the form:

```
any = ...
all = ...
```

Solution:

```
any = or <.> map
all = and <.> map
```

(d) Now do the same, but this time using the 'vanilla' composition operator `(.)`. Try using the same hint above for point-free functions.

Solution:

```
any = (or .) . map
all = (and .) . map
```

10. Using `(.)`, `foldr`, `map` and the identity function `id`, write a function `pipeline` which given a list of functions, each of type `a -> a` will form a pipeline function of type `[a] -> [a]`. In such a pipeline, each function in the original function list is applied in turn to each element of the input (assume the

functions are applied from right to left in this case). You can imagine this as being like a conveyor belt system in a factory where goods are assembled in a fixed number of processing steps as they pass down a conveyor belt. Each process performs a part of the assembly and passes the (partially completed) goods on to the next process. Test your function by forming a pipeline from the function list `[(+ 1), (* 2), pred]` with the resulting pipeline being applied to the input list `[1, 2, 3]`. Hint: Notice that if `f :: a -> a` then `map f` is a function of type `[a] -> [a]`.

Solution:

```
pipeline :: [a -> a] -> [a] -> [a]
pipeline = map . foldr (.) id
```

11. The `foldr` function is the embodiment of structural recursion on lists, where every element of the list is processed once, and in order, with results collapsed from right-to-left. Conversely, `foldl` is the embodiment of tail-recursive structural recursion on lists, where the result is threaded through as an accumulating parameter.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k [] = k
foldr f k (x:xs) = f x (foldr f k xs)
```

```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f k [] = k
foldl f k (x:xs) = foldl f (f k x) xs
```

As it happens, these two functions are equivalent, and can be expressed in terms of each other. In the following parts, you will take steps to figure out this definition yourself.

(a) First, we will consider the function `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`.

i. Give a recursive definition of `zipWith`, remembering that it will only zip the elements up to the smaller of the two lists.

Solution:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith _ _      _      = []
```

ii. If necessary, rework your solution to only pattern match on the first list on the left-hand side. i.e., your new solution should be of the form:

```
zipWith :: (a -> b -> c) -> [a] -> ([b] -> [c])
zipWith f []     = ...
zipWith f (x:xs) = ...
```

You may find a helper function useful here, or perhaps a `case` expression!

Solution:

```
zipWith :: (a -> b -> c) -> [a] -> ([b] -> [c])
zipWith _ []     = const []
zipWith f (x:xs) = matchYs
```

```
  where matchYs :: [b] -> [c]
        matchYs []     = []
        matchYs (y:ys) = f x y : zipWith f xs ys
```

Or, using `case` expressions and lambdas:

```
zipWith :: (a -> b -> c) -> [a] -> ([b] -> [c])
zipWith _ []     = \_ -> []
zipWith f (x:xs) = \ys -> case ys of
  [] -> []
  y:ys -> f x y : zipWith f xs ys
```

iii. Notice the shape of your solution now, rewrite `zipWith` as a `foldr` iterating over the first list `xs` so that it has the shape:

```
zipWith :: (a -> b -> c) -> [a] -> ([b] -> [c])
zipWith f xs = foldr cons nil xs
```

For some definition of `cons` and `nil`. Hint: look at the type signature for the function, and start by writing out the function signatures for `nil` and `cons` in a `where` clause. These will not be like the other examples you've seen so far! Let those types guide your implementation.

Solution: The types of `nil` and `cons` both involve functions, which means that `cons` has "more arguments" than usual. Thankfully, there is only one way to construct the `cons` function making use of all its components: it needs a `[b]`, so we should give that to `next` when we've taken something off it.

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f = foldr cons nil
  where nil :: [b] -> [c]
        nil = const [] -- as above!
        cons :: a -> ([b] -> [c]) -> [b] -> [c]
        cons x next []     = []
        cons x next (y:ys) = f x y : next ys
```

In essence, the `next` *continuation* is there to feed some value to the remaining computation: we peel off an element to match with our `x` and given the rest of the next to the "next elements", so to speak.

iv. When you have something working, evaluate `zipWith (+) [1, 2] [3, 4]` *on paper* and really make sure you understand how it operates.

Solution:

```
nil :: [Int] -> [Int]
nil = const []
```

```
cons :: Int -> ([Int] -> [Int]) -> [Int] -> [Int]
cons x next []     = []                -- clause 1
cons x next (y:ys) = x + y : next ys -- clause 2

zipWith (+) [1, 2] [3, 4]   {- By definition -}
                           = (foldr cons nil [1, 2]) [3, 4]
                             {- desugar lists -}
                           = (foldr cons nil (1:2:[])) (3:4:[])
                             {- definition of foldr -}
                           = cons 1 (foldr cons nil (2:[])) (3:4:[])
                             {- definition of cons clause 2 -}
                           = 1 + 3 : ((foldr cons nil (2:[])) (4:[]))
                             {- simplify -}
                           = 4 : ((foldr cons nil (2:[])) (4:[]))
                             {- definition of cons clause 2 -}
                           = 4 : 2 + 4 ((foldr cons nil []) [])
                             {- simplify -}
                           = 4 : 6 : (foldr cons nil []) []
                             {- definition of nil -}
                           = 4 : 6 : (const []) []
                             {- simplify -}
                           = 4 : 6 : []
```

(b) In this part, you will rewrite `foldl` in terms of `foldr`.

i. Take the provided definition of `foldl` above, and rewrite it so that the argument `k` is found on the right-hand side, so it is of the form:

```
foldl :: (b -> a -> b) -> [a] -> (b -> b)
foldl f []     = ...
foldl f (x:xs) = ...
```

Solution:

```
foldl :: (b -> a -> b) -> [a] -> (b -> b)
foldl f []     = \k -> k -- or simply just id!
foldl f (x:xs) = \k -> foldl f xs (f k x)
```

ii. For the `x:xs` case, try and rewrite your right-hand side to have shape `g x (foldl f xs)` for some definition of `g` you can define in a `where` clause.

Solution:

```
foldl :: (b -> a -> b) -> [a] -> (b -> b)
foldl f []     = id
foldl f (x:xs) = g x (foldl f xs)
  where g x next k = next (f k x)
```

iii. Notice the shape of your answer to part ii, and your answers to part a. Give the definition of `foldl` in terms of a single `foldr`, so that it is of the following shape:

```
foldl :: (b -> a -> b) -> [a] -> (b -> b)
foldl f xs = foldr cons nil xs
```

For some definition of `nil` and `cons`. Hint: think *carefully* about how the accumulating parameter should be flowing through the computation.

> **Solution:**
>
> ```
> foldl :: (b -> a -> b) -> [a] -> (b -> b)
> foldl f xs = foldr cons nil xs
>   where cons x next k = next (f k x)
>         nil           = id
> ```
>
> It's *really* easy to get this definition wrong, which is why we worked through `zipWith` first to see how information flows through parameterised folds like these but where the parametric types ensure we can't really go wrong. In this case, it is really easy to accidentally write `f (next k) x`. In fact, have a play with this incorrect definition: what function have you now accidentally made?

iv. Reorganise the arguments to your `foldl` so that it matches the regular type signature of `(b -> a -> b) -> b -> [a] -> b`.

> **Solution:**
>
> ```
> foldl :: (b -> a -> b) -> b -> [a] -> b
> foldl f k xs = foldr cons id xs k
>   where cons x next k = next (f k x)
> ```

(c) Try and do the opposite! Define `foldr` in terms of a single `foldl`. Hint: be careful with the information flow, things are now flowing *backwards* compared to the previous examples!

> **Solution:**
>
> ```
> foldr :: (a -> b -> b) -> b -> [a] -> b
> foldr cons nil xs = foldl f id xs nil
>   where f prev x k = prev (cons x k)
> ```
>
> The name `prev` is more appropriate here: you are feeding information back to previous iterations. In the other examples, you were feeding it forward to the "next" ones.

You can test your solution by trying out `foldr (:) [] [1, 2]`, you should get `[1, 2]` back.

> **Solution:**
>
> ```
> f prev x = \k -> prev (x : k)
> ```

```
foldr (:) [] [1, 2]   {- By definition -}
                     = (foldl f id (1:2:[])) []
                       {- definition of foldl -}
                     = (foldl f (f id 1) (2:[])) []
                       {- definition of f -}
                     = (foldl f (\k -> id (1 : k)) (2:[])) []
                       {- simplification -}
                     = (foldl f (\k -> 1 : k) (2:[])) []
                       {- definition of foldl -}
                     = (foldl f (f (\k -> 1 : k) 2) []) []
                       {- definition of f -}
                     = (foldl f (\k -> (\k -> 1 : k) (2 : k)) []) []
                       {- simplification -}
                     = (foldl f (\k -> 1 : 2 : k) []) []
                       {- definition of foldl -}
                     = (\k -> 1 : 2 : k) []
                     = 1 : 2 : []
```

Look at the steps like `(\k -> (\k' -> 1 : k') (2 : k))` before simplification, notice how it is feeding `2 : k` to the previous iteration, where `k` is in fact the list returned by the *next* iteration!