

---

## Chapter 7 Exercises

Exercises using foldr, foldl, and the composition operator

Connor Baker

## 7.9 Exercises

1. Show how the list comprehension `[f x | x <- xs, p x]` can be re-expressed using the higher order functions `map` and `filter`.

```
1 applyFilterAndFnToList :: (a -> Bool) -> (a -> b) -> [a] -> [b]
2 applyFilterAndFnToList p f xs = map f (filter p xs)
```

2. Without looking at the definitions from the standard prelude, define the following higher-order library functions on lists:

- (a) Decide if all the elements of a list satisfy a predicate:

```
1 all :: (a -> Bool) -> [a] -> Bool
```

- (b) Decide if any element of a list satisfies a predicate:

```
1 any :: (a -> Bool) -> [a] -> Bool
```

- (c) Select elements from a list while they satisfy a predicate

```
1 takeWhile :: (a -> Bool) -> [a] -> [a]
```

- (d) Remove elements from a list while they satisfy a predicate

```
1 dropWhile :: (a -> Bool) -> [a] -> [a]
```

*Note: In `Prelude` the first two of these functions are generic functions rather than being specific to the type of lists*

```
1 -- Part a
2 all :: (a -> Bool) -> [a] -> Bool
3 all p = and . map p
4
5 -- Part b
6 any :: (a -> Bool) -> [a] -> Bool
7 any p = or . map p
8
9 -- Part c
10 takeWhile :: (a -> Bool) -> [a] -> [a]
11 takeWhile _ [] = []
12 takeWhile p (x:xs)
13   | p x       = x : takeWhile p xs
14   | otherwise  = []
15
16 -- Part d
```

```

17 dropWhile :: (a -> Bool) -> [a] -> [a]
18 dropWhile _ [] = []
19 dropWhile p x'@(x:xs)
20   | p x      = dropWhile p xs
21   | otherwise = x'

```

3. Redefine the functions `map f` and `filter p` using `foldr`.

```

1 map' :: (a -> b) -> [a] -> [b]
2 map' f = foldr (\x xs -> f x : xs) []
3
4 filter' :: (a -> Bool) -> [a] -> [a]
5 filter' p =
6   foldr (\x xs ->
7     if p x then
8       x:xs
9     else
10      xs)
11   []

```

4. Using `foldl` define a function `dec2Int :: [Int] -> Int` that converts a decimal number into an integer. For example:

```

1 > dec2Int [2,3,4,5]
2 2345

```

```

1 dec2Int :: [Int] -> Int
2 dec2Int = foldl (\xs x -> 10*xs + x) 0

```

5. Without looking at the definitions from the standard prelude, define the higher-order library function `curry` that converts a function on pairs into a curried function, and, conversely, the function `uncurry` that converts a curried function with two arguments into a function on pairs. (Hint: first write down the types of the two functions.)

```

1 curry :: ((a,b) -> c) -> a -> b -> c
2 curry f a b = f (a,b)
3 -- Alternatively:
4 -- curry f = \a b -> f (a,b)
5
6 uncurry :: (a -> b -> c) -> (a,b) -> c
7 uncurry f (a,b) = f a b
8 -- Alternatively:
9 -- uncurry f = \ (a,b) -> f a b

```

6. A higher-order function `unfold` that encapsulates a simple pattern of recursion for producing a list can be

defined as follows:

```
1  unfold p h t x | px      = []
2                  | otherwise = h x : unfold p h t (t x)
```

That is, the function `unfold p h t` produces the empty list if the predicate `p` is true of the argument value, and otherwise produces a non-empty list by applying the function `h` to this value to give the head, and the function `t` to generate another argument that is recursively processed in the same way to produce the tail of the list. For example, the function `int2Bin` can be rewritten more compactly using `unfold` as follows:

```
1  int2Bin = unfold (== 0) (`mod` 2) (`div` 2)
```

Redefine the functions `chop8`, `map f` and `iterate f` using `unfold`.

```
1  unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
2  unfold p h t x
3      | p x      = []
4      | otherwise = h x : unfold p h t (t x)
5
6  chop8 :: [Int] -> [[Int]]
7  chop8 = unfold null (take 8) (drop 8)
8
9  map'' :: (a -> b) -> [a] -> [b]
10 map'' f = unfold null (f . head) tail
11
12 iterate' :: (a -> a) -> a -> [a]
13 iterate' = unfold (const False) id
```

7. Modify the binary string transmitter example to detect simple transmission errors using the concept of parity bits. That is, each eight-bit binary number produced during encoding is extended with a parity bit, set to one if the number contains an odd number of ones, and to zero otherwise. In turn, each resulting nine-bit binary number consumed during decoding is checked to ensure that its parity bit is correct, with the parity bit being discarded if this is the case, and a parity error being reported otherwise. (Hint: the library function `error :: String -> a` displays the given string as an error message and terminates the program; the polymorphic result type ensures that `error` can be used in any context.)

```
1  type Bit = Int
2
3  bin2Int :: [Bit] -> Int
4  bin2Int = foldr (\x y -> x + 2*y) 0
5
6  int2Bin :: Int -> [Bit]
7  int2Bin 0 = []
8  int2Bin n = mod n 2 : int2Bin (div n 2)
```

```

9
10 make8 :: [Bit] -> [Bit]
11 make8 bits = take 8 (bits ++ repeat 0)
12
13 encode :: String -> [Bit]
14 encode = concatMap (make8 . int2Bin . ord)
15
16 encodeWithParityBit :: String -> [Bit]
17 encodeWithParityBit s = encoded ++ [parityBit]
18     where
19         encoded    = encode s
20         parityBit = sum encoded `mod` 2
21
22 -- Using previously defined chop8 from #6
23
24 decode :: [Bit] -> String
25 decode = map (chr . bin2Int) . chop8
26
27 decodeWithParityBit :: [Bit] -> String
28 decodeWithParityBit list =
29     if last list == sum (init list) `mod` 2
30     then decode $ init list
31     else error "Parity bit doesn't match input actual string"

```

8. Test your new string transmitter program from the previous exercise using a faulty communication channel that forgets the first bit, which can be modeled using the tail function on lists of bits.

```

1 channelNoisy :: [Bit] -> [Bit]
2 channelNoisy = tail
3
4 transmitNoisy :: String -> String
5 transmitNoisy = decodeWithParityBit
6                 . channelNoisy
7                 . encodeWithParityBit

```

9. Define a function `altMap :: (a -> b) -> (a -> b) -> [a] -> [b]` that alternately applies its two argument functions to successive elements in a list, in turn about order. For example:

```

1 > altMap (+10) (+100) [0,1,2,3,4]
2 [10,101,12,103,14]

```

```

1 altMap :: (a -> b) -> (a -> b) -> [a] -> [b]
2 altMap _ _ [] = []
3 altMap f g (x:xs) = f x : altMap g f xs

```

10. Using `altMap`, define a function `luhn :: [Int] -> Bool` that implements the Luhn algorithm from the exercises in Chapter 4 for bank card numbers of any length. Test your new function using your own bank card.

```
1 luhnDouble :: Int -> Int
2 luhnDouble n
3   | n <= 5    = 2 * n
4   | otherwise = 2 * n - 9
5
6 luhn :: [Int] -> Bool
7 luhn [] = False
8 luhn ns = mod (sum $ altMap luhnDouble id ns) 10 == 0
```