

---

## Chapter 7 Exercises

Higher-Order Functions

Connor Baker

Compiled on December 24, 2019 at 9:22pm

## 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`.

Solution:

```
applyFilterAndFnToList :: (a -> Bool) -> (a -> b) -> [a] -> [b]
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:

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

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

- Decide if any element of a list satisfies a predicate:

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

- Select elements from a list while they satisfy a predicate

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

- Remove elements from a list while they satisfy a predicate

```
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*

Solution:

```
-- Part a
all :: (a -> Bool) -> [a] -> Bool
all p = and . map p

-- Part b
any :: (a -> Bool) -> [a] -> Bool
any p = or . map p

-- Part c
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ [] = []
takeWhile p (x:xs)
  | p x      = x : takeWhile p xs
  | otherwise = []

-- Part d
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p (x:xs)
  | p x      = dropWhile p xs
  | otherwise = x : dropWhile p xs
```

```
dropWhile p x'@(x:xs)
  | p x      = dropWhile p xs
  | otherwise = x'
```

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

Solution:

```
map' :: (a -> b) -> [a] -> [b]
map' f = foldr (\x xs -> f x : xs) []

filter' :: (a -> Bool) -> [a] -> [a]
filter' p =
  foldr (\x xs ->
    if p x then
      x:xs
    else
      xs)
    []
```

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

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

Solution:

```
dec2Int :: [Int] -> Int
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.)

Solution:

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f a b = f (a,b)
-- Alternatively:
-- curry f = \a b -> f (a,b)

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b
-- Alternatively:
-- 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:

```
unfold p h t x | px      = []
```

```
| 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:

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

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

Solution:

```
unfold :: (a -> Bool) -> (a -> b) -> (a -> a) -> a -> [b]
unfold p h t x
  | p x      = []
  | otherwise = h x : unfold p h t (t x)

chop8 :: [Int] -> [[Int]]
chop8 = unfold null (take 8) (drop 8)

map'' :: (a -> b) -> [a] -> [b]
map'' f = unfold null (f . head) tail

iterate' :: (a -> a) -> a -> [a]
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.)

Solution:

```
type Bit = Int

bin2Int :: [Bit] -> Int
bin2Int = foldr (\x y -> x + 2*y) 0

int2Bin :: Int -> [Bit]
int2Bin 0 = []
int2Bin n = mod n 2 : int2Bin (div n 2)

make8 :: [Bit] -> [Bit]
make8 bits = take 8 (bits ++ repeat 0)

encode :: String -> [Bit]
encode = concatMap (make8 . int2Bin . ord)
```

```

encodeWithParityBit :: String -> [Bit]
encodeWithParityBit s = encoded ++ [parityBit]
  where
    encoded    = encode s
    parityBit = sum encoded `mod` 2

-- Using previously defined chop8 from #6

decode :: [Bit] -> String
decode = map (chr . bin2Int) . chop8

decodeWithParityBit :: [Bit] -> String
decodeWithParityBit list =
  if last list == sum (init list) `mod` 2
  then decode $ init list
  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.

Solution:

```

channelNoisy :: [Bit] -> [Bit]
channelNoisy = tail

transmitNoisy :: String -> String
transmitNoisy = decodeWithParityBit
               . channelNoisy
               . 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:

Solution:

```

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

```

```

altMap :: (a -> b) -> (a -> b) -> [a] -> [b]
altMap _ _ [] = []
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.

Solution:

```

luhnDouble :: Int -> Int
luhnDouble n
  | n <= 5    = 2 * n
  | otherwise = 2 * n - 9

```

```
luhn :: [Int] -> Bool
luhn [] = False
luhn ns = mod (sum $ altMap luhnDouble id ns) 10 == 0
```