Functional programming

Exercises II

Higher-order functions

1. Define the function *map :: (a -> b) -> [a] -> [b]* using:
    a. List comprehension
    b. Recursion
2. Define the function filter :: (a -> Bool) -> [a] -> [a] using:
    a. List comprehension
    b. Recursion
3. Show how the list comprehension [f x | x <- xs, p x] can be re-expressed using the higher-order functions map and filter.
4. Without looking at the definition in the prelude, define the higher-order functions *all*, *any*, *takeWhile* and *dropWhile*.
5. Redefine the functions *map f* and *filter p* using *foldr*.
6. Using *foldl,* define a function *dec2int :: [Int] -> Int* that converts a decimal number into an integer. For example: *dec2int [2,3,4,5]* gives 2345
7. Explain why the following definition is invalid
   sumSqrEven = compose [sum, map (^2), filter even]
8. Without looking at the standard implementation, define a higher-order 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: write down the types of these functions before you begin
9. 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 | p x = []
   | otherwise = h x : unfold p h t (t x)
   That is, the function unfold p h t produces the empty list if predicate p is true of the argument, and otherwise produces a non-empty list by applying the function h 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*
10. Modify the string transmission program to detect simple transmission errors using parity bits. That is, each 8-bit number produced 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 9-bit 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 reported otherwise. Hint: the library function *error :: String -> a* terminates evaluation and displays the given string as an error message.
11. Test the above modification by using a broken communication channel that forgets the first bit. This can be implemented using *tail*.
12. Define a function altMap :: (a -> b) -> (a ->b) -> [a] -> [b] that alternatively applies its two argument functions to successive elements in a list.

13. Using altMap, define a function luhn :: [Int] -> Bool that implements luhn's algorithm for numbers of arbitrary length.

Data declarations

1. Using recursion and the function add, define a function mult :: Nat -> Nat -> Nat for natural numbers
2. There is a standard library type data Ordering = LT | EQ | GT and a function compare :: Ord a -> a -> a -> Ordering that decides if one value in an ordered type is Less Than, EQual to or Greater Than another such value. Redefine occurs :: Int -> Tree -> Bool to use this function. Why is this version more efficient?
3. Define a function balance :: [Int] -> Tree that converts a non-empty list of integers into a balanced tree.
4. Consider the following type of binary trees:
   data Tree = Leaf Int | Node Tree Tree
   Let's say such a tree is balanced if the number of leaves in the left and right subtree of every node differ by at most one, with leaves themselves trivially being balanced. Define a functions balanced :: Tree -> Bool that decides if a tree is balanced. Hint: first define a function that returns the number of leaves in a tree.
5. Given the data type declaration
   data Expr = Val Int | Add Exp Expr
   define a higher-order function
   folde :: (Int -> a) -> (a -> a -> a) -> Expr -> a
   such that folde f g replaces each Val constructor in an expression by the function f, and each Add constructor by the function g.
6. Using folde, define a function eval :: Expr -> Int that evaluates an expression to an integer value, and a function size :: Expr -> Int that calculates the number of values in an expression.
7. Complete the following instance declarations:
   instance Eq a => Eq (Maybe a) where
   …
   and
   instance Eq a = > Eq [a] where
   …