Exercises

Chapter 3

Exercise 1. Write a recursive function `copy :: [a] -> [a]` that copies its list argument. For example, `copy [2]` $\Rightarrow$ `[2]`. pp.52-53

Solution a

```
copy :: [a] -> [a]
copy []     = []
copy (x:xs) = x : copy xs
```

Solution b

```
copy :: [a] -> [a]
copy = map id
```

Exercise 2. Write a function `inverse` that takes a list of pairs and swaps the pair elements. For example,

```
inverse [(1,2), (3,4)] ==> [(2,1), (4,3)]
```

Solution a

```
inverse :: [a] -> [a]
inverse []          = []
inverse ((a,b):xs) = (b,a) : inverse xs
```

Solution b

```
inverse :: [a] -> [a]
inverse = map (\(a, b) -> (b, a))
```

Solution c

```
import Data.Tuple
inverse :: [a] -> [a]
inverse = map swap
```

Exercise 3. Write a function

```
merge :: Ord a => [a] -> [a] -> [a]
```

which takes two sorted lists and returns a sorted list containing the elements of each.

Solution

```
merge :: Ord a :: [a] -> [a] -> [a]
merge [] ys = ys
merge (x:xs) ys =
  let
    smaller = [ y | y <- ys, y <= x ]
    bigger  = [ y | y <- ys, y > x ]
  in
    merge xs (smaller ++ [x] ++ bigger)
```

Exercise 4. Write `(!!)`, a function that takes a natural number `n` and a list and selects the nth element of the list. List elements are indexed from 0, not 1, and since the type of the incoming number does not prevent it from being out of range, the result should be a `Maybe` type. For example,

```
[1,2,3]!!0 ==> Just 1
[1,2,3]!!2 ==> Just 3
[1,2,3]!!5 ==> Nothing
```

Solution

```
(!!) :: [a] -> Int -> Maybe a
(!!) [] _     = Nothing
(!!) (x:xs) 0 = Just x
(!!) (x:xs) n = (!!) xs (n-1)
```

Exercise 5. Write a function `lookup` that takes a value and a list of pairs, and returns the second element of the pair that has the value as its first element. Use a Maybe type to indicate whether the lookup succeeded. For example,

```
lookup 5 [(1,2),(5,3)] ==> Just 3
lookup 6 [(1,2),(5,3)] ==> Nothing
```

Solution

```
lookup :: Int -> [(Int, a)] -> Maybe a
lookup _ [] = Nothing
lookup n ((k,v):xs)
  | n == k    = Just v
  | otherwise = lookup n xs
```

Exercise 6. Write a function that counts the number of times an element appears in a list.

Solution

```
count :: Eq a => [a] -> a -> Int
count []     n  = 0
count (x:xs) n
  | x == n    = 1 + count xs n
  | otherwise = count xs n
```

Exercise 7. Write a function that takes a value `e` and a list of values `xs` and removes all occurrences of `e` from `xs`.

Solution

```
remove :: Eq a => [a] -> a -> [a]
remove []     _ = []
remove (x:xs) e
  | e == x      = remove xs e
  | otherwise   = x : remove xs e
```

Exercise 8. Write a function

```
f :: [a] -> [a]
```

that removes alternating elements of its list argument, starting with the first one. For examples, `f [1,2,3,4,5,6,7]` returns `[2,4,6]`.

Solution

```
alternating :: [a] -> [a]
alternating (x:[])    = []
alternating (x:xx:[]) = xx : []
alternating (x:xx:xs) = xx : alternating xs
```

Exercise 9. Write a function `extract :: [Maybe a] -> [a]` that takes a list of `Maybe` values and returns the elements they contain. For example, `extract [Just 3, Nothing, Just 7] = [3, 7]`.

Solution

```
extract :: [Maybe a] -> [a]
extract []              = []
extract ((Nothing):xs) = extract xs
extract ((Just x):xs)  = x : extract xs
```

Exercise 10. Write a function

```
f :: String -> String -> Maybe Int
```

that takes two strings. If the second string appears within the first, it returns the index identifying where it starts. Indexes start from 0. For example,

```
f "abcde" "bc" ==> Just 1
f "abcde" "fg" ==> Nothing
```

Solution

```
f :: String -> String -> Maybe Int
f text query = search text query 0
  where tokenOf       = take (length query)
        search [] _ _ = Nothing
        search t q i  = if tokenOf t == q
                        then Just i
                        else search (tail t) q (i+1)
```

Exercise 11. Write `foldrWith`, a function that behaves like `foldr` except that it takes a function of three arguments and two lists. p.56

Solution

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

Exercise 12. Using `foldr`, write a function `mappend` such that

```
mappend f xs = concat (map f xs)
```

Solution

```
mappend :: [a] -> [a]
mappend f xs = foldr ff [] xs
  where ff e acc = f e ++ acc
```

Exercise 13. Write `removeDuplicates`, a function that takes a list and removes all of its duplicate elements.

Solution

```
removeDuplicates :: [a] -> [a]
removeDuplicates = foldr addUniq []
  where addUniq acc e = if e `elem` acc
                        then acc
                        else e : acc
```

Exercise 14. Write a recursive function that takes a value and a list of values and returns `True` if the value is in the list and `False` otherwise.

Solution

```
onList :: Eq a => a -> [a] -> Bool
onList _ [] = False
onList e (x:xs) = if e == x
                  then True
                  else onList e xs
```

Exercise 15. Write a function that takes two lists, and returns a list of values that appear in both lists. The function should have type pp.59-60

```
intersection :: Eq a => [a] -> [a] -> [a]
```

(This is one way to implement the intersection operation on sets; see Chapter 8.)

Solution

```
intersection :: Eq a => [a] -> [a] -> [a]
intersection xs ys = foldr inter [] xs
  where inter x acc
          | x `elem` acc = acc
          | x `elem` ys  = x : acc
          | otherwise    = acc
```

Exercise 16. Write a function that takes two lists, and returns `True` if all the elements of the first list also occur in the other. The function should have type `isSubset :: Eq a => [a] -> [a] -> Bool`. (This is one way to determine whether one set is a subset of another; see Chapter 8.)

Solution

```
isSubset :: Eq a => [a] -> [a] -> Bool
isSubset xs ys = foldl inYs True xs
  where inYs acc x = x `elem` ys && acc
```

Exercise 17. Write a recursive function that determines whether a list is sorted.

Solution

```
isSorted :: Ord a => [a] -> Bool
isSorted []         = True
isSorted (_:[])     = True
isSorted (x:xx:xs) = if x < xx
                     then isSorted (xx:xs)
                     else False
```

Exercise 18. Show that the definition of `factorial` using `foldr` always produces the same result as the recursive definition given in the previous section.

Solution

```
-- recursive definition
factorial :: Int -> Int
factorial 0 = 1
factorial n = factorial (n-1) * n

factorial 3
= (factorial 2) * 3
= ((factorial 1) * 2) * 3
= (((factorial 0) * 1) * 2) * 3
= ((1 * 1) * 2) * 3
= (1 * 2) * 3
= 2 * 3
= 6

-- foldr definition
factorial :: Int -> Int
factorial n = foldr (*) 1 [1..n]

factorial 3
= foldr (*) 1 [1, 2, 3]
= 1 * foldr (*) 1 [2, 3]
= 1 * (2 * foldr (*) 1 [3])
= 1 * (2 * (3 * foldr (*) 1 []))
= 1 * (2 * (3 * 1))
= 1 * (2 * 3)
= 1 * 6
= 6
```

Exercise 19. Using recursion, define `last`, a function that takes a list and returns a `Maybe` type that is `Nothing` if the list is empty.

Solution

```
last :: [a] -> Maybe a
last []     = Nothing
last (x:[]) = Just a
last (x:xs) = last xs
```

Exercise 20. Using recursion, write two functions that expect a string containing a number that contains a decimal point (for example, `23.455`). The first function returns the whole part of the number (i.e., the part to the left of the decimal point). The second function returns the fractional part (the part to the right of the decimal point).

Solution

```
whole :: String -> String
whole ""     = ""
whole (s:ss) = if s == '.'
               then ""
               else s : (whole ss)

decimal :: String -> String
decimal = reverse . whole . reverse
```