Brown Higher-Order Function Study

With this assignment we participate in the Brown University study of Elijah Rivera and Shriram Krishnamurthi. Their research question is:

> How do students approach problems with structural vs pipeline solutions (and is there a discernible difference)?

Rules: Implement the functions below by using the following pre-defined higher-order functions only:

- `map :: (a -> b) -> [a] -> [b]`
- `filter :: (a -> Bool) -> [a] -> [a]`
- `foldl :: (b -> a -> b) -> b -> [a] -> b`
- `foldr :: (a -> b -> b) -> b -> [a] -> b`
- `any :: (a -> Bool) -> [a] -> Bool`
- `all :: (a -> Bool) -> [a] -> Bool`

You will likely have to complement them with auxiliary **non-recursive** functions. You are not allowed to write any recursive functions yourself.

Assignments:

1. Write a function `externalSenders :: [Server] -> [Report]`

   An email address is composed of two parts: a username and a domain. The username helps to identify the account that sent the email. The domain helps to identify the sender's (email) organization. For example, in the email address `jane.doe@company.abc`, `jane.doe` is the username and `company.abc` is the domain.

   In many organizations, there are different security restrictions for emails which originate "in-house" (from the company domain) versus externally. Organizations want to keep track of who is sending mail to their email server, and tend to monitor external senders more closely as vectors for potential attack. Often this responsibility is outsourced to large email hosting providers, who have to do this for many domains at once. At any moment, the email hosting provider needs to know which emails in each server are coming from external senders to flag them for potential further protective action.

   Thus, an email hosting service has a list of email-servers. Each email-server is associated with one domain (a company it's servicing). There are several emails in each email-server, waiting to be processed. We represent these with the following types:

   ```
   type Username = String
   type Domain = String

   data Email = MkEmail Username Domain
   ```

```
data Server = MkServer Domain [Email]

data Report = MkReport Domain [Email]
```

(We know that these data structures should have several other fields, such as the body of the email message, but we explicitly ignore them here to reduce the testing burden.)

The function `externalSenders` should consume a list of servers and produce a list of reports. Each report corresponds to a server in the input. Its emails field contains only those emails that are not from the domain being serviced by that sender. The output should contain a report for each server in the input, even if that server has no external emails. All emails in each report should be in the same order as in the corresponding server in the input.

```
Main> let brown = MkServer "brown.edu" [MkEmail "tom" "kuleuven.be"
                                       ,MkEmail "elijah" "brown.edu"]
Main> let kuleuven = MkServer "kuleuven.be" [MkEmail "elijah" "brown.edu"
                                            ,MkEmail "tom" "kuleuven.be"]
Main> let severs = [brown, kuleuven]
Main> externalSenders servers
[MkReport "brown.edu" [MkEmail "tom" "kuleuven.be"]
,MkReport "kuleuven.be" [MkEmail "elijah" "brown.edu"]]
```

2. Write a function `unique :: [Integer] -> [Integer]` that, given a list of values, produces a list of the same values in the same order excluding duplicates. If a value is present more than once, its first instance stays and the remainder are discarded. Use `(==)` and/or `(/=)` for comparison.

```
Main> unique [1,1,2,1,3,2]
[1,2,3]
```

3. Write a function `fixShelves :: [Shelf] -> [Shelf]`.

   Most bookstores sort the books on their shelves by the author's last name. Unfortunately, some bookstore patrons do not preserve this order when browsing books, and simply place books back wherever they fit.

   Assume that bookstores keep all authors whose last name starts with the same letter on the same shelf, and those shelves are labeled with that letter. A record of which authors are on a given shelf would be represented as:

```
type Letter = Char
type Author = String

data Shelf = MkShelf Letter [Author]
  deriving Show
```

The function `fixShelves` consumes a list of `Shelf` values and produces a list of those `Shelf` values where there is at least one author who doesn't belong on that shelf. The output `Shelf` values should only contain the authors who don't belong on that shelf. `Shelf` values and the authors within those records should be in the same order in the output as they appear in the input. Do not generate empty `Shelf` values; this generates needlessly long reports, which annoys the employees.

4. Write a function `elimContainsChar :: Char -> [String] -> [String]`, which removes the strings from the given list that contain the given character. All other strings are preserved in the same order.

```
Main> elimContainsChar 'w' ["one","two","three"]
["one","three"]

Main> elimContainsChar 'e' ["one","two","three"]
["two"]

Main> elimContainsChar 'z' ["one","two","three"]
["one","two","three"]
```

5. Write a function `leetSpeak :: [String] -> [String]`, which replaces the following characters in the strings following the 1337 speak style.

a $\mapsto$ 4	A $\mapsto$ 4
e $\mapsto$ 3	E $\mapsto$ 3
i $\mapsto$ 1	I $\mapsto$ 1
o $\mapsto$ 0	O $\mapsto$ 0
t $\mapsto$ 7	T $\mapsto$ 7

```
Main> leetSpeak ["one","two","three"]
["0n3","7w0","7hr33"]

Main> leetSpeak ["leet","speak","noob"]
["l337","sp34k","n00b"]

Main> leetSpeak ["ONE","TWO","Three"]
["0N3","7W0","7hr33"]
```

6. Write a function `isPalindrome :: String -> Bool` that determines whether, after removing spaces and punctuation, the input is a palindrome. A palindrome is a string with the same letters in both forward and reverse order (ignoring capitalization). The function `isAlphaNum :: Char -> Bool` from the `Data.Char` module may come in handy.

```
Main> isPalindrome "palindrome"
False
```

```
Main> isPalindrome "rotor"
True

Main> isPalindrome "a man, a plan, a canal: panama"
True
```

7. Write a function `viableFish :: [Fish] -> [String] -> [Fish]`.

   There are strict rules about fishing in the United States. Catching and keeping a fish that is below a certain size can result in a hefty fine. However, these rules don't apply to any fish designated as an "invasive species". In fact, in many locations, releasing these fish back into the wild carries its own fine.

   A fishing boat skipper knows that they must follow the above rules for which fish to release, and wants to keep track of which fish are not subject to the above rules. At every point in time on the trip, they want to be able to determine which non-invasive fish in their catch are above the minimum size restriction (8 inches).

   Fish data are presented with the following datatype:

```
type Species = String
type Length  = Float -- in inches

data Fish = MkFish Species Length
```

   The function `viableFish` takes in a list of caught fish currently in the boat and a list of names of invasive species (represented as strings), and returns a list, in the same order as the input, of the non-invasive fish that are larger than 8 inches.

```
Main> let invasive = ["Asian Carp", "Lionfish"]
Main> let caught = [MkFish "Asian Carp" 5, MkFish "Bull Trout" 7, MkFish "Whale Shark" 180]
Main> viableFish caught invasive
[MkFish "Whale Shark" 180]
```