CS181-N: Higher-order functions
This worksheet is for practice and is worth no course credit.

1 Maps

1.1 Say hello!

The function `sayHello` takes a list of names and produces a list of greetings, as in:

```haskell
> sayHello []
[]
> sayHello ['Yuji','Emiliano','Helen']
['Hello, Yuji!', 'Hello, Emiliano!', 'Hello, Helen!']
```

Write the type of `sayHello`.

Write two implementations of `sayHello`: one using natural recursion and one using `map`.

1.2 Make it a...

What type does the following function `d` have? Don’t just use the Haskell type checker—try to figure it out on your own, on paper.

```haskell
d [] = []
d (x:xs) = (x+x):d xs
```

\[ d :: \]

What would you call this function?  

Rewrite `d` using `map`.  

2 Filters

2.1 United we stand

The function \texttt{indivisible} takes a number and a list of numbers, returning those numbers in the list that are indivisible by the first input:

\begin{verbatim}
> indivisible 5 [1..10]
[1,2,3,4,6,7,8,9]
> indivisible 2 [1..10]
[1,3,5,7,9]
\end{verbatim}

There are many different types we can assign to \texttt{indivisible}; write a type that will allow at least the above examples to work.

Write two implementations of \texttt{indivisible}: one using natural recursion and one using \texttt{filter}.

2.2 A rock and a hard place

The function \texttt{between} takes a lower and upper bound and a list, returning those elements of the list that are between (inclusive) the two bounds.

\begin{verbatim}
> between 5 7 [1..10]
[5,6,7]
> between 'g' 'l' ['a'..'z']
"ghijkl"
\end{verbatim}

Write the type of \texttt{between}.

Write two implementations of \texttt{between}: one using natural recursion and one using \texttt{filter}.
3  Folds

Many functions in the Prelude reference Foldable t; for the purposes of this homework, please just use lists.

3.1  Long story short

The length function computes the length of a list, returning an Int. Write three versions of length: one using natural recursion, one using foldr, and one using foldl.

3.2  It’s so nice we’ll say it twice

The function stutter takes a list and returns a list with each item appearing twice:

> stutter []
[]
> stutter [1..4]
[1,1,2,2,3,3,4,4]

Write two versions of stutter: one using natural recursion, one using foldr.
3.3 You’ve got this down backwards and forwards

Write reverse four ways: using natural recursion, using accumulating recursion, using foldr, and using foldl.
3.4 Two great tastes that taste great together

Write two implementations of \( \text{concatMap} :: (a \rightarrow [b]) \rightarrow [a] \rightarrow [b] \): one using natural recursion and one using \( \text{foldr} \). Make sure you pass over the list only once.

3.5 Two great tastes that taste weird together

Just as \( \text{concatMap} f \) behaves like \( \text{concat} \ . \ \text{map} f \) (but passes over the list only once), the function \( \text{reverseMap} f \) behaves like \( \text{reverse} \ . \ \text{map} f \) (but passes over the list only once). Write two implementations of \( \text{reverseMap} :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \): one using natural recursion and one using \( \text{foldl} \).
3.6 Better keep ’em separated

In Data.List, the function \texttt{intercalate :: \([a] \rightarrow [[a]] \rightarrow [a]\)} is useful for text processing, as in:

\[
> \texttt{intercalate ", " \["A one","a two","a one two three"]}
\]

“A one, a two, a one two three”

Write two implementations of \texttt{intercalate}: one using natural recursion and one using \texttt{foldr}.

3.7 Mean means average

Write a function \texttt{mean} that computes the arithmetic mean of a list of numbers. There are many ways to write this function... so say which type yours has. Your function should be \(O(n)\) and pass over the list only once.

\[
\texttt{mean :: }
\]

Write \texttt{mean} two ways: using natural recursion and using either \texttt{foldr} or \texttt{foldl}. Make sure your functions agree on every input!
3.8 Laundry

What type does the following function \( b \) have?

\[
\begin{align*}
& b \_ \; [] = \text{error "nope"} \\
& b \; x \; (y:z) = \text{foldr} \; x \; y \; z
\end{align*}
\]

\( b :: \quad \) 

What would you call this function? 

4 Composing higher-order functions

In these questions, you can’t define your function \textit{directly in terms} of a higher-order function. You might have to use more than one higher-order function to get the answer, or you might have to pre- or post-process your data.

4.1 Saying it twice, again

Write a version of \texttt{stutter} (Problem 3.2) that uses \texttt{map} and \texttt{concat}. 
4.2 Conjunction junction

Write a version of the Prelude function `all :: (a->Bool) -> [a] -> Bool` using `map`, `filter`, and `length`.

Write another version of `all` using `filter` and `null`.

4.3 Ducks in a row

Write a function `isSorted` that determines whether a list is sorted ascending:

```haskell
> isSorted [1..10]
True
> isSorted ['a'..'z']
True
> isSorted "algebra"
False
> isSorted "sty"
True
```

Use `foldl` (and whatever else is handy). Your solution should be $O(n)$ and pass over the list only once.
4.4 A change of key

The function `transpose` “ pivots” a list of lists, as in:

```haskell
> transpose [[1,2,3],[4,5,6]]
[[1,4],[2,5],[3,6]]
> transpose [[1..5],[10],[],[20..30]]
[[1,10,20],[2,21],[3,22],[4,23],[5,24],[25],[26],[27],[28],[29],[30]]
```

Write `transpose`. Make it as concise and clear as possible, using higher-order functions (and other Prelude functions) as necessary. Remember: the most powerful tool isn’t always the right one for the job—try writing this function *without* a fold.
4.5 Taking attendance

Write three versions of \texttt{elem :: Eq a => a -> [a] -> Bool}: one using natural recursion, one using a fold of your choice, and one using a combination of \texttt{map, filter}.

4.6 Delete your account

Write two versions of \texttt{delete :: Eq a => a -> [a] -> [a]}: one using natural recursion, one using \texttt{foldr}. Unlike the \texttt{delete} in the Prelude, your function should delete \textit{all} occurrences of its first input in the list given as the second input.

4.7 A man, a plan, a canal: Haskell

Write a function \texttt{palindrome :: Eq a => [a] -> Bool} that determines whether a given list is a palindrome, i.e., the same backwards and forwards. (Look at the type—this is for arbitrary \(a\), not just characters, so there’s no need to try to get rid of punctuation or spaces or whatever.) Don’t let the space fool you—the answer can be quite short.

Quarter-point bonus: write a palindrome in English.
4.8 The heart of the matter

The function `nub` de-duplicates a list, as in:

>` nub [1..10]
>   [1,2,3,4,5,6,7,8,9,10]
>` nub ([1..10] ++ [20,22..30] ++ [1..10])
>   [1,2,3,4,5,6,7,8,9,10,20,22,24,26,28,30]
>` nub ([20,22..30] ++ [1..10] ++ [41,43..49] ++ [1..10])
>   [20,22,24,26,28,30,1,2,3,4,5,6,7,8,9,10,41,43,45,47,49]

Write three versions of `nub`: one using natural recursion, one using `foldr`, and one using `foldl`. You may use any function from the Prelude except for `reverse`—please solve this problem without reversing the list in any way.