Name: 

CAS ID (e.g., abc01234@pomona.edu): 

I encourage you to collaborate. Please record your collaborations below.

Most solutions using higher-order functions be written in a single-line. Some solutions may take as many as four or five lines, but any more and you’re off the scent.

Feel free to use Prelude definitions that help... but don’t make the question trivial.

Each question is worth one point.

Please turn in your work as a printout of this sheet, not on separate paper. If you would rather typeset your work, I can give you the LaTeX... but you’ll learn more by writing it by hand.

Collaborators: 
1 Maps

1.1 Say hello!

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

> 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 double

The function doubleAll takes a list of arithmetic expressions (as in HW02) and produces another list of arithmetic expressions that will evaluate to double the value. For example:

> doubleAll [Num 3,Neg (Num 5)]
[Times (Num 2) (Num 3),Times (Num 2) (Neg (Num 5))]

Write the type of doubleAll.

Write two implementations of doubleAll: one using natural recursion and one using map.
2 Filters

2.1 United we stand

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

> indivisible 5 [1..10]
[1,2,3,4,6,7,8,9]
> indivisible 2 [1..10]
[1,3,5,7,9]

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

Write two implementations of indivisible: one using natural recursion and one using filter.

2.2 A rock and a hard place

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

> between 5 7 [1..10]
[5,6,7]
> between 'g' 'l' ['a'..'z']
"ghijkl"

Write the type of between.

Write two implementations of between: one using natural recursion and one using 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 \texttt{reverse} four ways: using natural recursion, using accumulating recursion, using \texttt{foldr}, and using \texttt{foldl}.
3.4  Two great tastes that taste great together

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

3.5  Two great tastes that taste weird together

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

In Data.List, the function `intercalate :: [a] -> [[a]] -> [a]` is useful for text processing, as in:

```haskell
> intercalate " , " ["A one","a two","a one two three"]
"A one, a two, a one two three"
```

Write two implementations of `intercalate`: one using natural recursion and one using `foldr`.

4 Composing higher-order functions

In these questions, you can’t define your function *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 `stutter` (Problem 3.2) that uses `map` and `concat`.

4.2 Conjunction junction

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

Write a function \texttt{isSorted} that determines whether a list is sorted ascending:

\begin{verbatim}
> isSorted [1..10]
True
> isSorted ['a'..'z']
True
> isSorted "algebra"
False
> isSorted "sty"
True
\end{verbatim}

Use \texttt{foldl} (and whatever else is handy). Your solution should be $O(n)$ and pass over the list only once.

4.4 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 \textit{say which type yours has}. Your function should be $O(n)$ and pass over the list only once.

Write \texttt{mean} two ways: using natural recursion and using either \texttt{foldr} or \texttt{foldl}. 

4.5 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],[1],[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.
4.6  Taking attendance

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

4.7  Delete your account

Write two versions of `delete :: Eq a => a -> [a] -> [a]`: one using natural recursion, one using `foldr`. Unlike the `delete` in the Prelude, your function should delete all occurrences of its first input in the list given as the second input.
4.8 A man, a plan, a canal: Haskell

Write a function `palindrome :: Eq a => [a] -> Bool` that determines whether a given list is a palindrome, i.e., the same backwards and forwards.

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

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

```haskell
> 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`—you can solve this without reversing the list in any way.