Functional Programming Warmup

Background info

Base cases:

```
foldl f z [] = z
foldr f z [] = z
map f [] = []
```

Recursive definitions:

```
foldl f z (x:xs) = foldl f (f z x) xs
foldr f z (x:xs) = f x (foldr f z xs)
map f (x:xs) = f x : map xs
```

Group exercises

Write the following functions.

1. *Concat*:
   Write a function that, given a list *l* of lists, concatenates all the lists together. This function should *not* be passed to `fold()` or `map()`; instead, it should be implemented with a combination of `fold()` and `map()`.

   Example: `concat [ [a,b,c], [d,e,f], [g], [h,i] ] -> [a,b,c,d,e,f,g,h,i]`

2. *Group*:
   Write a function that, given a list *l* of key/value pairs, outputs a list of lists, where each sublist is of the form `(key, [v1, v2,...])`. There are two possible implementations; one uses a hash table and one does not. This function should *not* be passed to `fold()` or `map()`; instead, it should be implemented with a combination of `fold()` and `map()`.

   Example: `group [ [k1,a], [k2,b], [k1,c], [k1,d], [k3,e], [k2,f] ] -> [ [k1,[a,c,d]], [k2,[b,f]], [k3,[e]] ]`

3. *Bonus Question: Partition*:
   Write a function `f` that, given a list of integers and a value `k`, can be folded over the list such that the result of the fold is two lists partitioned around the value `k`. Feel free to hardcode the value `k` in function `f`.

Example: Let `k = 3`. Then `foldl f z [8,2,6,1,3,6,2,0] -> ([2,1,2,0], [3,8,6,6])`

Normally, hardcoding values is bad. However, since this is Haskell, I don't care that you hardcode `k` into `f` as it's trivially fixable. Why is that? What two language features makes this fix easy?

4. *Bonus Question: Composition*:
   Given 2 functions `f` and `g`, the "." operator in Haskell will create a third function that is the composition of the two. Write the code to do this (should be really short).