Let's do a little prep work first. This will allow our discussion to include some Prelude overlap.

```
import Prelude hiding(map, Functor, concat)

infixl 4 `map`
infixl 4 `multiF`
```

Now, a list of "a"s is either Nil (the empty list of any given type) or an "a" followed by another whole list of "a"s.

```
data List a = Nil
            | Cons a (List a)
```

The Prelude, of course, defined built-in syntax. Nil becomes [] and Cons becomes the infix data constructor (:). Thus, when we pattern match against a list we include patterns like (a:as) which identifies "a" as the head of the list and "as" as the tail.

If we have a list of "a"s, we may want to change them all in the same predefined way to "b"s. In other words, we will apply a function to all of them.

```
map :: (a → b) → [a] → [b]
map f [] = []
map f (a:as) = f a : map f as
```

If we have a list of lists, we may want to flatten them all into one list. We use the most obvious order.

```
concat :: [[a]] → [a]
concat [] = []
concat (as:aas) = as ++ concat aas
```

Now let's investigate a model where we want to represent uncertainty using lists

Say we have some numerical computations that can produce multiple results. For exampele, you want the possibility of working with both positive and negative square roots.

We can use lists to represent these computations, where every element in the list is a different answer. So let's write a few functions that take lists as arguments and results.

```
root1 :: [Float] → [Float]
root1 [] = []
root1 (x:xs) = [sqrt x, - sqrt x] ++ root1 xs
```

When the function has multiple arguments, we want to work on every possible combination of inputs

```
allSums :: [Float] → [Float] → [Float]
allSums [] _ = []
allSums (x:xs) ys = map addX ys ++ allSums xs ys
  where addX y = x + y

func1 :: [Float] → [Float] → [Float]
func1 [] _ = []
func1 (x:xs) ys = func1map x ys ++ func1 xs ys
  where
    func1map :: Float → [Float] → [Float]
    func1map _ [] = []
    func1map a (b:bs) = [a + b, a - b] ++ func1map a bs
```

Our final example function will turn out to be a bit different than the previous ones. See if you can guess why, but don't worry too much about it because it's pretty subtle.

```
root2 :: [Float] → [Float]
root2 [] = []
root2 (x:xs) | x < 0    = root2 xs
             | x > 0    = [sqrt x, - sqrt x] ++ root2 xs
             | x     = [0] ++ root2 xs
```

Let's look at some possible abstractions

There is very evidently a large number of possible things to abstract over, so let's shop around for some

My first instinct is to provide an easier way to do many functions at once. "map" ends up helping us a lot here.

```
multiF :: [a → b] → [a] → [b]
multiF [] _ = []
multiF (f:fs) as = map f as ++ multiF fs as
```

We recover root1 as

```
root1' = multiF [sqrt, negate · sqrt]
```

An interesting observation:

```
map' f = multiF [f]
```

Does the exact same thing that map does. Just a curiosity, or a deap connection?

Now let's look at the most powerful abstraction of the concept of multiple results

First, let's look at a function that's a bit different than the ones we've written thus far.

```
diff :: [Float] → [Float]
diff [] = [666] -- Not reached in evaluating ordinary lists
diff [a] = [2*a]
diff (a:b:xs) = a+b : diff (b:xs)
```

What is different about this function? Why, it uses multiple elements from the same list at the same time to produce its results. Our model that we want is this: a list of elements defines some number of results. Each of these results can go on to create more results if we apply more functions.

However, in diff we did not interpret our list as results. We used information that was embedded in the list itself (its order etc) and it was more like we were applying a function of two arguments across the list

So let's focus on single argument functions. The easiest way I can think of the get all of the values we want is this

```
mapMulti :: (a → [b]) → [a] → [[b]]
mapMulti f as = map f as
```

So it literally just turns out that mapMult = map, thanks to the amazing generality of map. However, this mapMulti strategy obviously does not work by itself, as it isn't composable. We turned a list into a list of lists, we want to get it back to a one dimensional list in order to preserve our formatting. Thankfully, we wrote the concat function earlier:

```
mapConcat :: (a → [b]) → [a] → [b]
mapConcat f as = concat $ map f as
```

mapConcat is the real function we're after.

Let's use mapConcat with the final versions of our root functions:

```
roots :: Float → [Float]
roots x = [sqrt x, - sqrt x]

roots' :: Float → [Float]
roots' x | x < 0   = []
         | x > 0   = [sqrt x, - sqrt x]
         | x    = [0]
```

Wow that was nice! Look at how pretty that is compared to our first implementation. Now we can officially recover "root1 = mapConcat roots" and "root2 = mapConcat roots'" whenever we need these functions

Okay that was cool, but how to we generalize our func1, which applied a two argument function over two lists? We can use brute force and implement exactly that:

```
smashWith :: (a → b → [z]) → [a] → [b] → [z]
smashWith _ [] _ = []
smashWith f (a:as) bs = concat (map (f a) bs) ++ smashWith f as bs
```

Note that we partially apply the function f to get (f a) :: b -¿ z, then we map (f a) over all the elements of "bs"

Now we have

```
func1' = smashWith (λa b → [a+b, a-b])
```

Now, should we add another version of this function for every arity we need? Certainly we could, for instance

```
combThree :: (a → b → c → z) → [a] → [b] → [c] → [z]
```

We could, but there is a better way. First think about how we could implement combThree:

```
combThree f as bs cs = let -- applyA :: [b -¿ c -¿ z]
                           applyA = map f as
                           -- applyB :: [c -¿ z]
                           applyB = multiF applyA bs
                       in multiF applyB cs
```

in other words,

```
combThree' f as bs cs = map f as `multiF` bs `multiF` cs
```

So there's our pattern that we're after. If we need to extend the arity, we'll just use `multiF` as many times as needed.

In reproducing combThree we didn't even need our buddie multiF. We can use this style (map once then multiF the rest of the way home) to implement allSums:

```
allSums' xs ys = (+) `map` xs `multiF` ys
```

And with a little cleverness, we can use multiF to implement func1 as well. This time we don't need map because we have our function(s) inside a list to begin with.

```
func1'' xs ys = [(+),(-)] `multiF` xs `multiF` ys
```

Now, if you try to implement root2 using multiF alone, you run into a real difficulty. The difference between root2 and root1 ultimately comes down to the fact that the number of results produced by root2 is allowed to depend on the input. We can implement root1 using only multiF:

```
root1'' xs = [sqrt, negate · sqrt] `multiF` xs
```

Just like in func1, we could write out a single list that represented all of the output patterns. But for root2, there are multiple output patterns (lengths of list 0 for negative input, length 1 for an input of zero, and length 2 for positive input).

Luckily, we saw earlier that our buddy concatMap overcomes this difficiency of multiF. To review, we have

```
root2' xs = concatMap root2Helper xs
  where root2Helper x | x < 0   = []
                      | x > 0   = [sqrt x, - sqrt x]
                      | x    = [0]
```

The above example seems to indicate that concatMap is fundamentally more powerful than multiF. With concatMap we can customize the shape of the resulting list, where that is not an option with multiF. We found that root2 could not be expressed with the tools map and multiF alone. The observation that multiF can be defined in terms of map and concatMap confirms this suspicion.

```
multiF' :: [a → b] → [a] → [b]
multiF' fs as = concatMap (λa → fmap ($a) fs) as

multiF'' :: [a → b] → [a] → [b]
multiF'' fs as = concatMap eval as
  where eval a = fmap (λf → f a) fs
```

That first version used slightly advanced style, so I provided two versions.