Lecture 10: Applicative

Exercise 10.1

In the spirit of the `liftA*` functions, implement the following to lift an unboxed function and apply it to a boxed list of arguments.

```
liftAN :: Applicative f => ([a] -> b) -> f [a] -> f b
```

How useful is this function?

Exercise 10.2 Unsurprisingly, there is also an instance of `Applicative` for `Either a`. Provide an instance definition, and compare it to the definition in the Haskell sources.

***Exercise 10.3** Perhaps surprisingly, given the foregoing, there is *not* an `Applicative` instance for `(,) a`. Why not?

Exercise 10.4

Write an expression in applicative style that computes the same result as:

```
[ (x,y,z) | x <- [1..3], y <- [1..3], z <- [1..3] ]
```

***Exercise 10.5**

Consider the following two, very similar looking calculations:

```
> [(+),(*)] <*> pure 2 <*> pure 3
[5,6]
> ZipList [(+),(*)] <*> pure 2 <*> pure 3
ZipList {getZipList = [5,6]}
```

The results of these computations (modulo syntactic noise around `ZipList`) are identical, but the computational patterns that produce these results are quite different. Explain the difference.

Exercise 10.6

There are several additional operators defined to improve readability when writing programs in applicative style:

```
(<$)   :: Functor f => a -> f b -> f a
(*>)   :: Applicative f => f a -> f b -> f b
(<*)   :: Applicative f => f a -> f b -> f a
(<**>) :: Applicative f => f a -> f (a -> b) -> f b
```

We won't often use them in our examples. But, similar to our discussion of `foldMap` and `foldr` last time, it can be helpful to think about how to implement such polymorphic functions based only on their types and what we know about the type classes that are mentioned in their constraints.

Try implementing these functions before peeking at them in the libraries.