Practice with pattern matching and function parameters.

The most common use of pattern-matching is in writing functions that process a list. We already saw a number of examples in that direction. The main elements of the process are as follows:

1. We handle in some special way the "base" cases of the empty list, and possibly the list of one element.
2. We handle the general case of a list with a head and a tail. This typically involves calling the function recursively onto the tail, then doing some more work with the result.

The `map` function is a good example of this process:

```
map :: (a -> b) -> [a] -> [b]
map f []       = []
map f (x:xs)   = f x : map f xs
```

Note the second case. We call `map f xs` to obtain the result for the tail of our list. Then we also compute `f x` and put it at the front of the list.

Let us also write the function `filter`: `filter` takes a predicate, which is a function of type `a -> Bool`. Then it takes a list of values, applies the predicate to them, and only returns those for which the predicate is `True`. Here's how that looks like:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p []                   = []
filter p (x:xs) | p x         = x :: filter p xs
                | otherwise = filter p xs
```

Let us look at some more examples. For instance let us write the function `take` that returns the first however many elements from a list. The logic would go like this:

1. If we are asked to take 0 or less elements, then we simply return the empty list.
2. If we are asked to take a number of elements from the empty list, then we simply return the empty list.
3. If we are asked to take `n` elements from a non-empty list, then we will take `n-1` elements from its tail, then append the head element.

Let us translate that into code:

```
take :: Int -> [a] -> [a]
take _ []                    = []
take n (x:xs) | n <= 0       = []
              | otherwise    = x : take (n-1) xs
```

Practice Problems

You are expected to do these using pattern-matching and recursion as above, and not via other means.

7. Write a function `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`. It takes a function that turns an `a` and a `b` into a value of type `c`, and also takes a list of `a`s and a list of `b`s. It then forms a list out of the result of applying the function to the corresponding pairs of elements.

8. (difficult) Write a function `splitWith :: (a -> Bool) -> [a] -> ([a], [a])` which take as input a predicate and a list, and separates the list in two lists, with the first list containing those elements for which the predicate is `True` and the second list containing those elements for which the predicate is `False`. The order of elements must be maintained within each list.