More Practice with Pattern Matching

Implementation of `map`

Before moving on, we can now give a straightforward implementation of the map function:

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

This is a typical recursive function, and we will study recursive functions in more detail later. But in effect, we recursively build the correct list for the tail of our input, then fix it by appending the value from the head.

More Complex Patterns

Let us proceed to some more advanced pattern-matching with list patterns. One of the cool features is that you can dig deeper into a list with a single pattern. As an example, imagine we wanted to write a function allEqual that checks if all values in a list are equal to each other. The logic of it could go something like this:

1. If the list has 1 or fewer elements, then the answer is True.
2. If the list has at least two elements, then we check that the *first two* elements are equal to each other, then drop the first element and expect all the rest to equal each other.

This could look as follows:

```
allEqual :: Eq t => [t] -> Bool
allEqual []         = True
allEqual (x:[])     = True
allEqual (x:y:rest) = x == y && allEqual (y:rest)
```

Question: Why is it wrong to just say allEqual rest at the end?

Warning: It is very tempting to change the last pattern to say (x:x:rest), expecting that this would only match if the first two elements of the list match. This *does not work*. You cannot repeat variables in a pattern. It would work with literals though, like (True:True:rest).

Practice

1. Write pattern-matching definitions for the function fst that given a pair returns the first entry, and the function snd that given a pair returns the second entry. Don't forget to use wildcards for values you dont need, and to start by writing the types of the functions.

2. Using allEqual as a starting template, write a function isIncreasing that checks if a list of numbers is in increasing order, each next number in the list being larger than the ones before it.

3. Using allEqual as a starting template, write a function hasDups that given a list of elements tests if the list has any *consecutive duplicates*, i.e. if there is ever a point in the list where two consecutive elements are equal.

4. Using allEqual as a starting template, write a function addStrange that given a list of numbers looks at each pair (1st&2nd, 3rd&4th etc), and from each one picks the largest number, then adds those. If there is a single element remaining at the end, just use it.

Pattern-matching examples

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.

1. Write a function `length :: [a] -> Int` which given a list returns its length.
2. Write a function `last :: [a] -> a` which returns the last element of a non-empty list. You should not worry about its behavior on an empty list.
3. Write the function `(++) :: [a] -> [a] -> a` which concatenates two lists. Write it using `++` as an infix operator in the definition (will not need the parentheses if it is infix).
4. Write a function `init :: [a] -> [a]` which given a list returns the list without the last element. If the list is empty it should return an empty list.
5. Write a function `zip :: [a] -> [b] -> [(a, b)]` that given two lists forms pairs (tuples) out of the corresponding values (i.e. the first elements go together, the second elements go together etc). Stop when either list runs out of elements.
6. (difficult. Skip for now) Write a function `unzip :: [(a, b)] -> ([a], [b])` that basically does the opposite of `zip`.
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. Write a function `insertOrdered :: Ord t => t -> [t] -> [t]` that takes a list containing values in increasing order, possibly with duplicates, and a new element to insert into the list. It then inserts that element in the correct spot to preserve the order. For example `insertOrdered 4 [1, 3, 6] = [1, 3, 4, 6]`.
9. Write a function `searchOrdered :: Ord t => t -> [t] -> Bool` that takes a list containing values in increasing order, possibly with duplicates, and an element, and it checks to see if the element is in the list. *This function should only traverse as much of the list as it needs to.*

10. Write a function `interject :: [a] -> [a] -> [a]` that given two lists produces a new list with the values interjected. So the first value of the first list goes first, followed by the first value of the second list, followed by the second value of the first list and so on. If any list ends first, the remaining entries are formed from the remaiming elements. For example `interject [1, 2, 3] [4, 5, 6, 7, 8] = [1, 4, 2, 5, 3, 6, 7, 8]`.

11. (difficult) Write a function `splitAt :: Int -> [a] -> ([a], [a])` which takes an integer and a list, and splits the list in two at that integer and stores the two parts in a tuple. If the integer is 0 or less, then the first part of the tuple would be `[]`. If the integer is longer than the list length, then the second part of the tuple would be `[]`. Simple example: `splitAt 3 [1..5] = ([1, 2, 3], [4, 5])`

12. (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.