More Practice with Pattern Matching

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).

Now let's do another complex example: The method unzip takes a list of pairs and returns individual lists for each component. For example unzip [(1, 2), (3, 4)] = ([1, 3], [2, 4])

Implementing this function shows an important recursive function pattern: We perform a *pattern-match* on the result of the recursive call, then adjust the resulting values. In this case imagine what unzip would have to do to be implemented recursively in our example above:

1. The next value is the pair (x, y) = (1, 2)
2. The recursive call would have split up the list [(3, 4)] into the two lists (xs, ys) = ([3], [4])
3. All we now need to do is combine the x with the xs and the y with the ys.
4. We also need to handle the case of the empty list, by returning a pair of empty lists.

```
unzip :: [(a, b)] -> ([a], [b])
unzip [] = ([], [])
unzip ((x, y):rest) = (x:xs, y:ys)
    where (xs, ys) = unzip rest
```

As one last example, let's implement the list concatenation operator, ++. This is an example of writing a definition for an infix operator. This is also an example where the input has two lists. We recurse on the first list:

“‘haskell (++) :: [a] -> [a] -> [a] [] ++ ys = ys (x : xs) ++ ys = x : (xs ++ ys)

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.

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. 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].

7. 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.*

8. 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].

9. 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])`