Pattern Matching

Pattern matching is a powerful technique that allows us to follow different code paths depending on the *structure*, the *shape* of a value. For example when working with lists, a list can have either the shape `[]` of an empty list or the shape `x:xs` of a list with a first element followed by all the remaining elements. Patterns allow us to tap into those shapes.

Reading

- Sections 4.1-4.4
- Practice Problems: 4.8 1, 2, 3, 4, 5, 6

Patterns in Function Definitions

There are two common uses of patterns. The first, and most common one, is in function definitions. You can specify how the function should behave for different structures of its argument.

As a first example, recall our definition of an `Account` as a pair `(Bool, Double)`, and our definition of a `deposit` function:

```
deposit :: Double -> Account -> Account
deposit amountAdded (hasFunds, amount)
        | hasFunds  = (True, amount + amountAdded)
        | otherwise = initAccount (amountAdded - amount)
```

This function looks at the pair, and associates the first entry with the variable `hasFunds` and the second entry with the variable `amount`. The guarded expression then checks to see if `hasFunds` is `True` or not.

A pattern-matching approach would instead say: A pair of type `(Bool, Double)` has either the form `(True, aNumber)` or `(False, aNumber)` and I will take an action in each case. It looks like this:

```
deposit :: Double -> Account -> Account
deposit amountAdded (True, funds) = (True, funds + amountAdded)
deposit amountAdded (False, deficit) = initAccount (amountAdded - deficit)
```

> A **pattern** specifies the form a value may take. It can contain literals as well as variables. Its form is compared to the form of the value, and if they match then the variables get *bound to* the corresponding values.

As another example, imagine we wanted to write a `mergeAccounts` function that takes two accounts and merges them into one. It could look like this:

```
mergeAccounts :: Account -> Account -> Account
mergeAccounts (True, funds1) (True, funds2) = (True, funds1 + funds2)
mergeAccounts (False, owed1) (False, owed2) = (False, owed1 + owed2)
mergeAccounts (True, funds) (False, owed)   = initAccount (funds - owed)
mergeAccounts (False, owed) (True, funds)   = initAccount (funds - owed)
```

Rock-Paper-Scissors As a further example, let us imagine a setup for the Rock-Paper-Scissors game. We will learn later more formally about defining new types, but for now imagine that we have a new type that takes exactly three possible values: Rock, Paper or Scissors. we could then write a function beats which is given two "hand" values and returns whether the first value beats the second. In order to implement this function we need to basically say that if the two values are in one of the three winning arrangements then the result is True and otherwise the result is False. This could look like the following. We will use the backtick notation as it reads more naturally:

```
Rock     `beats` Scissors = True
Scissors `beats` Paper    = True
Paper    `beats` Rock     = True
_        `beats` _        = False
```

Notice here the last case: We have used underscores to indicate that we don't care what goes there, the result doesn't depend on in. We already captured the interesting cases earlier. This underscore is called the **wildcard** pattern.

List Patterns

We have already seen list patterns informally when we discussed an implementation for the sum function. Let us revisit that function now with all the extra knowledge we have obtained:

```
-- sum function that adds all the elements in a list.
sum :: Num t => [t] -> t
sum []     = 0
sum (x:xs) = x + sum xs
```

Let's go through this line by line:

1. The function sum has the type Num t => [t] -> t, because it takes a list of values that can be added and returns their sum. Thefore the contents of the list must have a type that has an instance of the Num class.
2. There are two lines defining the sum function, depending on the shape/structure of the list parameter.
   a. If it is an empty list, then the sum [] line matches it and the result is 0.
   b. If it is not an empty list, then we check the next formula and see if it matches it. That formula looks for a list matching the pattern (x:xs), and any non-empty list has that form. Therefore the right-hand-side of that expression will be evaluated, with the variable x bound to the first element and the variable xs bound to the list of the remaining elements.

As another example, let us write a function allTrue that is given a list of booleans and is supposed to return True if they are all True (or if the list is empty) and False if there is at least one False value in the list. We can write this with pattern-matches thus:

```
allTrue :: [Bool] -> Bool
allTrue []           = True
allTrue (True:rest) = allTrue rest
allTrue (False:_)    = False
```

Let's take a look at this one.

1. The first pattern is the same idea as before, handling the empty list case.
2. The second pattern matches any list whose first entry is literally True, followed by anything. In that case we want to simply check the rest of the list, so we recursively call the allTrue function.
3. The first pattern matches any list whose first entry is literally False. In this case the result of the function is supposed to be False regardless of what the rest of the list does. Since we don't care what value the rest of the list takes, we use the wildcard pattern for it.

Practice: Write a function anyTrue, which returns True whenever there is *at least one* True value somewhere in the list. Start by writing the type of the function and the different pattern cases you would want to consider.

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.