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.

> **Patterns** describe a desired *structure* for a value. We can **pattern match** such a pattern to particular value. This match can succeed or fail, and if it succeeds then variables specified in the pattern get associated with corresponding parts of the value.

The simplest pattern we can probably look at is something like the following:

```
let (x, y) = (3, 4)
```

This matches the pattern `(x, y)` with the value `(3, 4)`. since the value is a tuple of length 2 and the pattern is the same, they match, and `x` is associated to `3` and `y` is associated to `4`.

On the other hand, the following would not match and produce an (unfriendly) error:

```
let (x, y) = 5
```

The simplest pattern is a single variable itself. So when we type

```
let x = 4
```

we are actually performing a pattern match of the pattern `x` with the value `4`. The pattern `x` will match any value and associated `x` to `4`.

> Patterns can be encountered in the following situations:
>
> 1. In function definitions in the place of the function parameters.
> 2. In the left-hand side of an assignment in a `where` or `let` clause. For example `let (x, y) = (3, 4)` will set `x` to `3` and `y` to `4`, and it is an example of a pattern match.
> 3. In the special `case ... of ...` syntax, which we will see in a few moments.

Here are some rules for pattern-matching:

> Pattern-matching rules:
>
> - A single variable pattern `x` matches any value `v` and associates `x` to that value.
> - An underscore _ matches any value and creates no new associations.

- A tuple pattern (p1, p2, p3, ..., pn) matches any value (v1, v2, v3, ..., vn) of the same arity *only if* each pattern pi matches the corresponding value vi. It creates the associations created by the component patterns.
- A constructor pattern, C p1 p2 ... pn will match a constructor value C v1 v2 ... vn *only if* each pattern pi matches the corresponding value vi. It creates the associations created by the component patterns.

This last one is a bit complicated. But as a simple example, imagine a type for card values: It has one variant for the numeric values, and different variants for Jack through King. We then define a function that returns the numeric value of such a card, with figure cards all having value 10:

```
data CardValue = NumValue Int | Jack | Queen | King

value :: CardValue -> Int
value (NumValue n)   = n
value Jack           = 10
value Queen          = 10
value King           = 10
```

So if we do something like value (NumValue 5) then the pattern-matching mechanism will compare the NumValue 5 value with each of the patterns in the definition of value, and it will find it matching the first pattern, NumValue n, associating the value 5 with the variable n.

List Patterns

As a further example, let's revisit lists and now look a bit under the hood. We could for example define our own list type with something like:

```
data MyList t = Empty | Cons t (MyList t) deriving (Eq, Show)
```

This is saying, that an instace of MyList, which needs a parametrized type t is:

1. either the empty list Empty, or
2. the result of putting together via the Cons constructor a value of type talong with a list of values of type t.

So this way we can create the list [2,3] as follows:

```
Cons 2 (Cons 3 Empty)
```

In Haskell we more or less have the same thing, with the changes that Empty is really [] and Cons is really the colon operator, so we can write this example as:

```
2:3:[]
```

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 the function and 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:

```
and :: [Bool] -> Bool
and []          = True
and (True:rest) = and rest
and (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:

1. Write a function or, which returns True whenever there is *at least one* True value somewhere in the list (and it should return False for the empty list. Start by writing the type of the function and the different pattern cases you would want to consider.

2. Write a function length which returns the length of a string.
3. Write a function head which returns the first element of the list (use error "empty list" for the empty list case).
4. Write a function last which returns the last element of the list (use error "empty list" for the empty list case).
5. Write a function maximum which returns the largest value of the list (use error "empty list" for the empty list case).
6. Write a function onlyEvens which given a list of integers returns whether the list contains only even numbers.
7. Write a function everyOther which given a list build a new list containing every other element from the original list. Make sure your code can handle both even and odd length lists.
8. Write a function init which given a list returns a new list containing all but the last element.
9. Write a function alternating which given a list of booleans returns True if the booleans keep alternating in value and False otherwise. For example [True, False, True] would return True but [True, False, False, True] would return False because of the two consecutive False entries.