Higher-order functions

Our discussion of functions as values and currying naturally leads to the idea of higher-order functions. In principle higher-order functions are simply functions that take as input, amongs other things, other functions. For example here is a function that "composes" two other functions:

```
let compose f g = fun x -> f (g x)
```

Question: What is the type of compose? What is another way of writing it?

Maps

The term higher-order functions is often more associated with *functions that operate on compound structures and take the action to be performed as a parameter function*. Nowhere is this best exemplified than in the List module. Lists are meant to simply hold a list of items to operate on. For example we can square all elements on a list by the following:

```
let rec square xs = match xs with
| [] -> []
| x :: xs' -> x * x :: square xs'
```

What if we wanted to triple the elements? Or add 5 to them? We could write very similar functions in this case, but the amount of repetition should suggest that we should find a better way around it. This is not unlike the situation of using for loops to operate on the elements of an array. In OCAML we can be more expressive about it: Write a function that takes as input a function to operate on lists, and returns the function we defined above. Here's how that might look like:

```
let map f =
    let rec apply_f xs = match xs with
    | [] -> []
    | x :: xs' -> f x :: apply_f xs'
    in apply_f
```

Question: What is the type of apply_f? What is the type of map?

So we might use this as follows:

```
let square = map (fun x -> x * x)       (* returns the "square a list" function *)
square [1; 4; 6]
map (fun x -> x * x) [1; 4; 6]          (* Equivalent to the line above *)
```

Before we move on let us see a slightly different, and more common, way of writing this function, which will serve as a model for the functions we will write in the future:

```
let rec map f xs = match xs with
| [] -> []
| x :: xs' -> f x :: map f xs'
```

The main point is essentially **separation of concerns**, or if you prefer a **decoupling of responsibilities**:

- map only concerns itself with how to traverse the list. It does not really care what happens to the elements on the list, nor should it. Functions should ideally try to do only one thing, and make sure they do it well.
- The parameter function f passed to map is the one that knows what should happen to the elements on the list. It also does not need to care at all about the fact that it traverses a whole list, it just needs to know what to do to an element of the list.

Higher-order functions, along with the convenience of anonymous functions, offers this powerful decoupling functionality.

The function map is just the first in a series of higher order functions we will explore. We will see more of those in the next section. For now we focus on perhaps the two most important higher-order functions, fold_left and fold_right.

Folds

Folding is simply the act of accumulating over a list. Let us think of the basic for loop:

- We declare and initialize an "accumulator".
- As we visit each element in the list, we update the accumulator based on our problem's logic of how the current element is supposed to contribute.
- When we visit every element in the list, the accumulator holds the final answer.

As a simple example, let us revisit the question of adding all numbers in the list:

```
let rec sum xs = match xs with
| [] -> 0
| x :: xs' -> x + sum xs'
```

This would be called a right-fold. Essentially there are two possible ways we could try to add all the numbers in the list, starting with an accumulator of 0: We could add them left-to-right, or right-to-left:

```
sum [1; 2; 3] ---> ((0 + 1) + 2) + 3          (* left fold *)
sum [1; 2; 3] ---> 1 + (2 + (3 + 0))          (* right fold *)
```

Question: Look back at the implementation of sum above, and make sure you understand which of these two expressions it produces.

Let's see how we can generalize the definition of sum above. Recall that our main goal is to separate the idea of iterating over the list from the operations that are meant to be performed on the elements of the list. In the above definition there are two parts that are specific to the summing operation, as opposed to the idea of the list traversal:

- The value to be returned at the empty list (in this case 0). This is effectively the initial value of the accumulator, and in general would need to be provided.

- The operation to be performed between the new value `x` and the accumulated value from recursion, `sum xs'`. We will need to be provided a function that takes these two inputs and combines them appropriately. If the values in the list have a type `'a` and the accumulator has a type `'b`, then this provided function would have type `'a -> 'b -> 'b`.

Here is how that would look like:

```
(* fold_right: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b *)
let rec fold_right f xs acc = match xs with
| [] -> acc
| x :: xs' -> f x (fold_right f xs' acc)
```

Notice how the call to `f` in the last clause takes the place of the addition in the example of `sum`.

We could now implement sum as:

```
let sum xs = fold_right (fun x acc -> x + acc) xs 0
let sum xs = fold_right (+) xs 0          (* alternative *)
```

A function that multiplies all the numbers would be:

```
let mult xs = fold_right ( * ) xs 1
```

We could even implement our function `map` as a call to `fold_right`. The accumulator starts as the empty list, and the function combines the so-far-accumulated elements, a list, with the value of the new element, to form a longer list:

```
let map f xs = fold_right (fun x rest -> (f x) :: rest) xs []
```

Left folds are only a tad harder, and they operate more in the "state recursion" way. This would normally require a helper function, but the fact that `fold_left` takes as arguments the list and accumulator gives us all the state we need.

One other unusual thing is that the function in the left fold expects its arguments in the opposite order. This is done in order to reflect the left-folding nature.See the two examples of summation earlier. It also reverse the order of the other two parameters: the initial value and the list of items. Here `'a` is the accumulator's type, and `'b` is the list elements' type:

```
(* fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a *)
let rec fold_left f acc xs = match xs with
| [] -> acc
| x :: xs' -> fold_left f (f acc x) xs'
```

Note that our implementation of `fold_left` is naturally tail-recursive, while our implementation of `fold_right` is not.

Whenever you find yourself wanting to accumulate information in a list, you should think of using a fold.