Standard higher-order list functions

There is a rich set of functions on lists, that you can find in the List module[1]. We will implement these from scratch in this section. Most of these functions can be extended from lists to other "container types".

- map has a function act on each element of the list, and produces a new list with the results.
- filter takes as input a *predicate* (a function returning true/false) and returns a list containing only those values that return true.
- fold_left accumulates the values in the list, given an initial value and a function describing how to accumulate each new term.
- fold_right does the same, but starting from the other end.
- iter calls a function with no return value ('a -> unit) on each element of the list.
- for_all takes a predicate and a list and returns whether all elements of the list would return true.
- exists takes a predicate and a list and returns whether at least one element of the list would return true.
- find takes a predicate and a list and searches for the first element in the list satisfying the prediate.
- partition takes a predicate and a list and returns two lists, one holding the values that the predicate returns as true and one holding those the predicate returns as false.

We have already seen map, fold_left, fold_right. We will now implement filter in a couple of different ways, both directly and using fold_right.

```
(* filter: ('a -> bool) -> 'a list -> 'a list *)
let rec filter p xs = match xs with
| [] -> []
| x :: xs' -> if p x
              then x :: filter p xs'
              else filter p xs'

let filter p xs = fold_right (fun x acc -> if p x then x :: acc else acc) xs []
let filter p xs =
    let add_if_true x acc = if p x then x :: acc else acc
    in fold_right add_if_true xs []
```

Question: Could we have used fold_left? Try it!

Before some practice problems, let us implement iter both via direct recursion and using folds:

```
(* iter: ('a -> unit) -> 'a list -> unit *)
let rec iter f xs = match xs with
| [] -> ()
| x :: xs' -> (f x; iter f xs')
```

[1] http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

```
let iter f xs = fold_left (fun x () -> f x) () xs

(* To test. Should print in that order *)
iter print_endline ["hi"; "there"; "you"];;
```

Note that using fold_right here would not be right: It would have performed the applications in the reverse list order, usually not the desired effect.

Practice problems

1. Implement the function rev that reverses a list, using fold_left.
2. Implement for_all using fold_left or fold_right (ideally do it both ways). It should have type ('a -> bool) -> 'a list -> bool.
3. Implement exists using fold_left or fold_right (ideally do it both ways). It should also have type ('a -> bool) -> 'a list -> bool.
4. Implement partition both with a direct recursion and using fold_right. It should have type ('a -> bool) -> 'a list -> 'a list * 'a list.