Functional Programming
Exercises Week 8
(for December 6, 2013)

Numbers in parentheses refer to the 6th edition of the course notes.
Exercises marked with ⋆ are optional and can be ignored.

1. Read Chapter 7 of the lecture notes.

2. (Exercise 7.1) Which of the following functions is tail recursive, which one is not?

```
let rec map f = function []    -> []
                       | x::xs -> f x::map f xs

let rec foldl f b = function []    -> b
                           | x::xs -> foldl f (f b x) xs

let rec foldr f b = function []     -> b
                           | x::xs -> f x (foldr f b xs)
```

3. (Exercise 7.2) Implement a *tail recursive* function `rev_append_tl` such that

`rev_append_tl [1;2;3] [4;5;6]` = `[3;2;1;4;5;6]`

Hint: Your function should have the type `'a list -> 'a list -> 'a list`

4. (Exercise 7.4) Consider the function `Lst.replicate`

```
let rec replicate n x =
 if n < 1 then [] else x::replicate (n-1) x
```

Is this function tail recursive? If yes, justify your answer, otherwise give a tail recursive implementation.

5. (Exercise 7.9) Use *tupling* to implement a version of `Lst.span`

```
let span p xs = (take_while p xs, drop_while p xs)
```

that just needs one list traversal to compute its result.

⋆. (Exercise 7.12) Consider '`@`' in OCaml, i.e.,

```
let rec (@) xs ys = match xs with []     -> ys
                                | x::xs -> x::(xs @ ys)
```

a) From Lemma 6.3 we know that '`@`' is associative. However, internally OCaml has to evaluate the operator either left- or right-associative. What is the choice for '`@`' in OCaml? Give evidence.

Hint: Which of the following reductions is performed by OCaml: Either

`[1]@[2]@[3]` → `[1;2]@[3]` → `[1;2;3]`

or

`[1]@[2]@[3]` → `[1]@[2;3]` → `[1;2;3]` ?

b) Determine the number of computation steps OCaml needs to evaluate

i. `(([1]@[2])@` ⋯ `)@[`n`]` (left-associative)

ii. `[1]@([2]@(` ⋯ `@[`n`]))` (right-associative)

Hint: Recall that OCaml adopts an eager evaluation strategy.