```
let rec concat l = match l with
| [] -> ""
| x::xs -> x ^ (concat xs)

let concat' l =
let rec conc l acc = match l with
| [] -> acc
| x::xs -> conc xs (acc ^ x)
in
conc l ""
```

Prove that concat l produces the same as concat' l.

Intuitively this is clear as both functions start off with the same initial state
and keep joining elements together until the list is empty, the only difference being that concat' uses an inner function conc so as to store the result in an accumulator
variable. concat' is better in the sense that there is no danger of a stack overflow since the string we want to eventually return is being stored in 'acc' rather than on the stack.

formal proof: (this is extremely boring)

NTS concat l = concat' l

(Step 1) Base Case: l = []

concat l = concat [] = "" = conc [] "" = concat' [] = concat' l

the first equality is due to the definition of l in the base case
the second equality is by the program concat
the third and fourth equalities are by the program concat'

(Step 2) Assume true for list l

(Step 3) Inductive Step: Prove that equality still holds if we add an extra item to list l.

```
let item = additional generic item in list l
let l1 = item :: l
```

```
concat l1  = concat  (item :: l) = item ^ (concat l)
                                 = item ^ (concat' l)
                                 = item ^ (conc l "")
                                  = conc l item
                                 = concat' (item :: l)
                                 = concat' l1
```

first equality -> by definition of l1
second equality -> by concat function
third equality -> by step 2
fourth equality -> by concat' function
fifth equality -> by concat' function
sixth equality -> by concat' function
seventh equality -> by definition of l1

Q.E.D

Again, concat' is better in the sense that there is no danger of a stack overflow since the string we want to eventually return is being stored in 'acc' rather than on the stack.