Exercise 1 (Tail-recursion)**.**

1. Write a tail-recursive factorial function.
2. Write a tail-recursive exponentiation function.
3. Write a tail-recursive function which concatenates two lists.

*4. Write a function

```
maxima (f : nat -> nat) (l r : nat) : nat
```

which computes the number of local maxima of f in the range $[l, r]$. You may assume that f is injective.

Exercise 2 (Higher-order functions)**.**

1. Define the list concatenation function `List.app` using `List.fold_right`.
2. Using `List.fold_left` or `List.fold_right`, define a function

```
lcomp {A} (l : list (A -> A)) : A -> A
```

which composes all functions in a list.

For example,

```
lcomp [fun x => x + 1; fun x => x * 2; fun x => x - 1]
```

should be a function which given x computes $2(x-1)+1$.

***Exercise 3** (Dependently typed expressions)**.**

Consider the following dependent version of the `expr` type from the lecture. Here instead of an arbitrary type, the argument to `expr` is an ordinary data structure (an element of `type` representing a type). An advantage is that now we can match on values in `type`.

```
Inductive type := Nat | Bool.

Definition tyeval (ty : type) : Type :=
  match ty with Nat => nat | Bool => bool end.

Inductive expr : type -> Type :=
| var : nat -> expr Nat
| plus : expr Nat -> expr Nat -> expr Nat
| equal : expr Nat -> expr Nat -> expr Bool
| const (A : type) : tyeval A -> expr A
| ite (A : type) : expr Bool -> expr A -> expr A -> expr A.

Definition store := nat -> nat.
```

Define a function `eval {A} (s : store) (e : expr A) : tyeval A` which evaluates such expressions.

***Exercise 4** (Vectors)**.**

Recall the definition of vectors from the lecture.

```
Inductive vector (A : Type) : nat -> Type :=
| nil : vector A 0
| cons (_ : A) (n : nat) : vector A n -> vector A (S n).
```

1. Define a function `vlength {A n} (v : vector A n) : nat` which computes the length of a vector.
2. Define a `map` function for vectors.