Homework 1
Types for programs and proofs
due 19 September 2013, 13.15

These exercises are all about Agda programming.

1. (a) Define "exclusive or" using pattern matching!

   ```
   xor : Bool -> Bool -> Bool
   ```

   (b) Define the factorial function using pattern matching!

   ```
   factorial : Nat -> Nat -> Nat
   ```

2. In Haskell, the reverse function on lists is a polymorphic function of type

   ```
   reverse :: [a] -> [a]
   ```

   This is so called *ML polymorphism*.

   (a) Define the polymorphic reverse function in Agda! This can be done by quantifying over `a : Set`! This is a form of *explicit polymorphism*; since you explicitly need to quantify over all "sets" ("small types").

   (b) Define another polymorphic reverse function where `a : Set` is an *implicit argument*! With implicit arguments you can write functions in Agda with types looking much like the way you would write them in Haskell.

   (c) In section 3.1 of "Dependent types at work" the type `Vec A n` of vectors of length `n` is defined. Define the reverse function on vectors, so that its type expresses that the length of the output vector is the same as the length of the input vectors. You can choose whether to define `Vec A n` either as a recursive family or as an inductive family as explained in "Dependent types at work" .

3. In section 3.2 in "Dependent Types at Work" the type `Fin n` of finite sets with `n` elements is define. Do the two exercises at the end of that section

   (a) Write a new lookup function `_!!_` so that it has the following type:

   ```
   _!!_ : {A : Set}{n : Nat} -> Vec A (succ n) -> Fin (succ n) -> A
   ```

   This will eliminate the empty vector case, but which other cases are needed?

(b) Give an alternative definition of `Fin n` as a recursive family, that is, define it by induction on `n` using pattern matching!

4. The primitive recursion operator `natrec` is a polymorphic higher order function which takes a base case and a step case and returns a function defined by primitive recursion with that base case and step case. See section 2.5 in "Dependent Types at Work".

   (a) Define the factorial function in terms of `natrec`!