Proofs from Homework 3
Recall extensionality is
For all f,g: A → B. f = g if and only if for all x:A. f x = g x

The id function is defined as follows:
id x = x
and in general has the type id:: a → a in Haskell (or in our mathematical notation id: A → A)

Proof 1: Prove ∀f : A → B. (f o id) = f
Choose an arbitrary function f : A → B
Proof by extensionality. We must show that:
   For all x: A. (f o id) x = f x
Pick an arbitrary x in A.
Show
   (f o id) x = f x
   f (id x) = f x by definition of o (compose)
   f x = f x by definition of id

   The two functions are equal by extensionality.

Analysis of type of id. In this case the id function had the type A→ A. This is because the function f required an element of type A as its argument and more importantly the element x had type A. Since id assumes the type of whatever its argument is and returns something of the same type, id::A → A.

Proof 2: Prove ∀f : A → B. id o f = f
Choose an arbitrary function f : A → B
Proof by extensionality. We must show that:
   For all x: A. (id o f) x = f x
Pick an arbitrary x in A.
Show
   (id o f) x = f x
   id(f x) = f x by definition of o (compose)
   f x = f x by definition of id

   The two functions are equal by extensionality.

Analysis of type of id. In this case the id function had the type B→ B. This is because the function f returns an element of type B. Hence whatever (f x) returned would be of type B and id assumes the type of its argument.
Type inference questions:

1. \( f [] = [] \)
   \[ f (x:xs) = [x] ++ f xs \]

   \( f :: [a] \rightarrow [a] \)
   
   We know that the argument and the resulting types are lists because the function takes a list as an argument and in both cases constructs a list on the right hand side. In this case, since we have no information about the elements in the list (actually this is just an identity function on lists) we can only say the lists contain some type \( a \).

2. \( f x y = x \mod y \)

   \( f :: (\text{Num } a) \Rightarrow a \rightarrow a \rightarrow a \) (also \( f :: \text{Int } \rightarrow \text{Int } \rightarrow \text{Int} \) would be acceptable)

   In this case we see, on the left, that \( f \) takes two arguments \( x \) and \( y \) one at a time. On the right hand side we see the modulo function, hence we know that the resulting type is numeric but that it is not transforming the type of its arguments at all.

3. \( f x y (z:zs) = (g x y z) : f x y zs \)

   here \( g \) is some function previously defined and has the type: \( g :: a \rightarrow b \rightarrow c \rightarrow d \)

   \( f :: a \rightarrow b \rightarrow [c] \rightarrow [d] \)

   In this case we do not have very specific information about the function \( g \) except its type and the fact that it takes three arguments. We can however get the type of \( f \) from the information.

   The first thing we see is, on the left, that \( f \) takes 3 arguments \( x, y, z:zs \). The last element being a list. We can match the variables \( x \) and \( y \) very quickly on the right hand side when they are applied to \( g \). Since we know \( g \) takes an element of type \( a \) and then type \( b \) we can determine that \( x :: a \) and \( y :: b \). Next \( g \) takes an element of type \( c \). In the implementation, we pass \( z \), the head of the list \( (z:zs) \) to the function \( g \). So \( z :: c \) and then \( (z:zs) :: [c] \).

   We can determine the result type of the function \( f \) to be \( [d] \) because
   
   1) \( g \) applied to three arguments of types \( a, b, \) and \( c \) result in a value of type \( d \)
   2) The result of \( g x y z \) is cons'd onto the list created by the recursive call to \( f \) applied to the same first two arguments and the tail of the list \( (z:zs) \), namely \( zs \).

4. \( f x y \)

   | \( y == 0 \) = Error “No division by zero”
   | \( y > 0 \) = \( x / y \)

   \( f :: (\text{Num } a, \text{Ord } a) \Rightarrow a \rightarrow a \rightarrow a \) (any numeric variation of this is acceptable)

   Like 2, \( f \) is taking two arguments \( x \) and \( y \) one at a time. In the case analysis, if the second argument is 0 then we return an error. Recall that Error assumes any type (like bottom). In the
case where $y > 0$ (hence the Ord type class constraint) it returns the result of dividing $x$ by $y$. The actual result would depend on the actual type of $a$ (Int, Float,...).