1. Consider the definition of $\text{nat}$:

\[
\text{Inductive nat : Set := O : nat | S : nat \rightarrow nat.}
\]

(a) What are the constructors of $\text{nat}$?

(b) Describe the elements of $\text{nat}$.

(c) Give the type of $\text{nat\_ind}$.

2. Consider the following definition:

\[
\text{Inductive A : Set :=} \\
\quad | \ a : A \rightarrow A \\
\quad | \ b : A \rightarrow A \rightarrow A.
\]

How many elements does the set $A$ have?

3. (a) Consider the definition of $\text{natlist}$ for lists of natural numbers:

\[
\text{Inductive natlist : Set :=} \\
\quad | \ nil : natlist \\
\quad | \ cons : nat \rightarrow natlist \rightarrow natlist.
\]

Give the type of $\text{natlist\_ind}$, which is used to give proofs by induction.

(b) Give the definition of an inductive predicate $\text{last\_element}$ such that $(\text{last\_element } n \ l)$ means that $n$ is the last element of $l$.

4. (a) Give the inductive definition of the datatype $\text{natbintree}$ of binary trees with unlabeled nodes and natural numbers at the leafs.

(b) The Coq function for appending two lists is defined as follows:

\[
\text{Fixpoint append (l k : natlist) {struct l} : natlist :=} \\
\text{match l with} \\
\quad \text{nil \Rightarrow k} \\
\quad | \ \text{cons } n \ l' \Rightarrow \text{cons } n \ (\text{append } l' \ k)
\text{end.}
\]

In what argument is the recursion? Why is the recursive call (intuitively) safe?
(c) Give the definition of a recursive function `flatten : natbintree -> natlist` which flattens a tree into a list that contains the nodes from left to right.
You may use `append`.

(d) Give a recursive definition of a function `count` that takes as input a `natbintree` and gives as output the number of nodes of the tree.

5. Consider the definition of an inductive predicate for even:

```coq
Inductive even : nat -> Prop :=
| even_zero : even 0
| even_greater : forall n:nat, even n -> even (S (S n)).
```

(a) What is the type of `even 0`?
(b) Give an inhabitant of `even 0`.
(c) Give an inhabitant of `even 2`.

6. (about program extraction) What is the type of the function that can be extracted from the proof of the following theorem:

```coq
forall l : natlist,
\{l' : natlist | Permutation l l' /\ Sorted l'\}.
```

7. This exercise is concerned with dependent types. We use the following definition in Coq:

```coq
Inductive natlist_dep : nat -> Set :=
| nil_dep : natlist_dep 0
| cons_dep : forall n : nat,
          nat -> natlist_dep n -> natlist_dep (S n).
```

(a) What is the type of `natlist_dep`?
(b) What is the type of `natlist_dep 2`?
(c) Describe the elements of `natlist_dep 2`.

(b) Suppose we want to define a function `nth` that takes as input a list and gives back the `n`th element of that list. How can dependent lists be used to avoid errors?

8. Give the type of `append_dep`, the function that appends two dependent lists. We use the following definition:
Fixpoint append_dep
  (n1:nat) (n2:nat)
  (l1:natlist_dep n1) (l2:natlist_dep n2)
  {struct l1} : natlist_dep (n1 + n2) :=
match l1 in (natlist_dep n1) return (natlist_dep (n1 + n2)) with
| nil_dep => l
| cons_dep p h t => cons_dep (p + n2) h (append_dep p t n2 l)
end.

9. Give the type of reverse_dep, the function that reverses a dependent list.
We use the following definition:

Fixpoint reverse_dep
  (n:nat) (l:natlist_dep n) {struct l} :
  natlist_dep n :=
match l in (natlist_dep n) return (natlist_dep n) with
| nil_dep => nil_dep
| cons_dep p h t =>
  eq_rec (plus p 1) (fun n => natlist_dep n)
  (append_dep p (reverse_dep p t) 1 (cons_dep 0 h nil_dep))
  (S p) (P p)
end.

where

P : forall p : nat, p + 1 = S p

so P is a proof that p + 1 equals S p for all p.

10. Consider the following two terms:

reverse_dep (plus n1 n2) (append_dep n1 n2 l1 l2)
append_dep n2 n1 (reverse_dep n2 l2) (reverse_dep n1 l1)

(Here n1 and n2 have type nat, the term l1 has type natlist_dep n1,
the term l2 has type natlist_dep n2.)

What are the types of the above terms?
Are the types convertible?