



1. Consider the definition of `nat`:

```
Inductive nat : Set := 0 : nat | S : nat->nat.
```

- (a) What are the constructors of `nat`?
- (b) Describe the elements of `nat`.
- (c) Give the type of `nat_ind`.

**Answer:**

The definition of `nat` :

```
Inductive nat : Set := 0 : nat | S : nat->nat.
```

- (a) The constructors of `nat` are `0` and `S`.
- (b) The elements of `nat` are `0`, `(S 0)`, `(S (S 0))`, etcetera, and represent the natural numbers.
- (c) `nat_ind`:  

```
forall P : nat -> Prop,  
P 0 ->  
(forall n : nat, P n -> P (S n)) ->  
forall n : nat, P n
```

2. Consider the following definition:

```
Inductive A : Set :=  
| a : A -> A  
| b : A -> A -> A.
```

How many elements does the set `A` have?

**Answer:**

The set `A` does not have any elements (intuitively, there is no way to start).

3. (a) Consider the definition of `natlist` for lists of natural numbers:

```

Inductive natlist : Set :=
| nil : natlist
| cons : nat -> natlist -> natlist.

```

Give the type of `natlist_ind`, which is used to give proofs by induction.

**Answer:**

The type of `natlist_ind`:

```

natlist_ind :
  forall P : natlist -> Prop,
  P nil ->
  (forall (n : nat) (n0 : natlist), P n0 -> P (cons n n0)) ->
  forall n : natlist, P n

```

- (b) Give the definition of an inductive predicate `last_element` such that `(last_element n l)` means that `n` is the last element of `l`.

**Answer:**

A definition of `last_element`:

```

Inductive last_element (n:nat) : natlist -> Prop :=
|last_one : last_element n (cons n nil)
|last_more : forall h:nat, forall t:natlist,
  last_element n t -> last_element n (cons h t).

```

4. (a) Give the inductive definition of the datatype `natbintree` of binary trees with unlabeled nodes and natural numbers at the leaves.

**Answer:**

An inductive definition of the datatype `natbintree`:

```

Inductive natbintree : Set :=
| natleaf : nat -> natbintree
| natnode : natbintree -> natbintree -> natbintree.

```

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

```

Fixpoint append (l k : natlist) {struct l} : natlist :=
  match l with
  | nil => k
  | cons n l' => cons n (append l' k)
  end.

```

In what argument is the recursion? Why is the recursive call (intuitively) safe?

**Answer:**

The recursion is in the first argument.

The recursion is (intuitively) safe because `l'` is a smaller term than `l`.

- (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`.

**Answer:**

```
Fixpoint flatten (b : natbintree) {struct b} : natlist :=
  match b with
  | natleaf n => (cons n nil)
  | natnode l r => (append (flatten l) (flatten r))
  end.
```

- (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.

**Answer:**

A definition of `count`:

```
Fixpoint count (b: natbintree) {struct b} : nat :=
  match b with
  | natleaf n => 0
  | natnode l r => S (plus (count l) (count r))
  end.
```

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

```
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`?

**Answer:**

The type of `even 0` is `Prop`.

- (b) Give an inhabitant of `even 0`.

**Answer:**

An inhabitant of `even 0` is `even_zero`.

- (c) Give an inhabitant of `even 2`.

**Answer:**

An inhabitant of `even 2` is `(even_greater 0 even_zero)`.

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

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

**Answer:**

`natlist -> natlist`.