1. (10 points) Circle True or False for each statement.

(a) All functions defined in Coq via Fixpoint must terminate on all inputs.

   True   False

(b) The proof of an implication \( P \rightarrow Q \) is a function that uses a proof of the proposition \( P \) to produce a proof of the proposition \( Q \).

   True   False

(c) The proposition \( \text{true} = \text{false} \) is provable in Coq.

   True   False

(d) Given a function \( f \) of type \( \text{nat} \rightarrow \text{bool} \), it is possible to define a proposition that holds when \( f \) returns \( \text{true} \) for all natural numbers.

   True   False

(e) There are no empty types in Coq. In other words, for any type \( A \), there is some Coq expression that has type \( A \).

   True   False

(f) If \( H : \text{true} = \text{false} \) is a current assumption, then the tactic \text{inversion} H will solve any goal.

   True   False

(g) If \( H : S \, x = S \, (S \, y) \) is a current assumption, then the tactic \text{inversion} H will solve any goal.

   True   False

(h) If \( H : x \neq y \) is a current assumption, then the tactic \text{inversion} H will solve any goal.

   True   False

(i) If the goal is \( A \wedge B \), then the tactic \text{split} will produce two subgoals, one for \( A \) and one for \( B \).

   True   False

(j) If \( H : x_1 :: y_1 = x_2 :: y_2 \) is a current assumption, then we know that \( x_1 \) is equal to \( x_2 \).

   True   False
2. (10 points) Write the type of each of the following Coq expressions, or write “ill-typed” if it does not have one. (The references section contains the definitions of some of the mentioned functions and propositions.)

(a) \texttt{beq_nat 3 4}

(b) \texttt{3=4}

(c) \texttt{forall \ (X:Type), forall \ (x:X), x = x}

(d) \texttt{fun \ (X:Prop) => X -> X}

(e) \texttt{fun \ (x:nat) => x :: x}
3. (8 points) Recall the definition of flat_map from the homework (The ++ function is given in the references):

```
Fixpoint flat_map {X Y:Type} (f:X -> list Y) (l:list X) : (list Y) :=
  match l with
  | []       => []
  | h :: t   => (f h) ++ (flat_map f t)
  end.
```

This function applies f to each element in the list and appends the results together. For example:

```
Example test_flat_map1:
  flat_map (fun (n:nat) => [n;n;n]) [1;5;4] = [1; 1; 1; 5; 5; 5; 4; 4; 4].
```

(a) Complete the definition of the list filter function using flat_map. (You will receive no credit if your answer uses Fixpoint!)

```
Definition filter {X : Type} (test: X->bool) (l:list X) : (list X) :=
```

Your filter should satisfy the same tests as the filter we saw in class. For example:

```
Example test_filter1: filter evenb [1;2;3;4] = [2;4].
```

(b) Complete the definition of the list map using flat_map. (You will receive no credit if your answer uses Fixpoint!)

```
Definition map {X Y:Type} (f : X -> Y) (l : list X) : list Y :=
```

Again, your map should satisfy the same tests as the map we saw in class. For example:

```
Example test_map1: map (plus 3) [2;0;2] = [5;3;5].
```
4. (17 points) An alternate way to encode lists in Coq is the \texttt{jlist} type, shown below.

\begin{verbatim}
Inductive jlist (X:Type) : Type :=
  \mid j_nil : jlist X
  \mid j_one : X -> jlist X
  \mid j_app : jlist X -> jlist X -> jlist X.

(* Make the type parameter implicit *)
Arguments j_nil \{X\}.
Arguments j_one \{X\} _.
Arguments j_app \{X\} _ _.
\end{verbatim}

We can convert a \texttt{jlist} to a regular \texttt{list} with the following function:

\begin{verbatim}
Fixpoint to_list \{X : Type\} (jl : jlist X) : list X :=
  match jl with
  \mid j_nil => []
  \mid j_one x => [x]
  \mid j_app j1 j2 => to_list j1 ++ to_list j2
  end.
\end{verbatim}

(a) Note that there may be multiple \texttt{jlist}s that represent the same \texttt{list}. Demonstrate this fact by giving definitions of \texttt{example1} and \texttt{example2} such that the Lemma below (\texttt{distinct_jlists_to_same_list}) is provable (there is no need to prove it).

Definition example1 : jlist nat :=

Definition example2 : jlist nat :=

Lemma distinct_jlists_to_same_list :
  example1 <> example2 \land (to_list example1) = (to_list example2).
(b) It is also possible to define most list operations directly on the jlist representation. Complete the following function for mapping over a jlist:

\[
\text{Fixpoint } j\_\text{map} \{ X \ Y : \text{Type} \} (f : X \to Y) (x : \text{jlist } X) : \text{jlist } Y :=
\]

(c) What is the type of the expression \text{j\_one}?

(d) What is the type of the expression \text{j\_map (fun (x:nat) => beq_nat x 0)}?
(e) Your j_map function from part (b) should satisfy the following correctness lemma that states that it agrees with the list map operation. (The list map function is shown in the references.) The proof of this lemma for our definition of j_map is shown below. This proof uses an auxiliary lemma (map_app), not shown.

Lemma j_map_correct : forall (X:Type) (Y:Type) (f : X -> Y) (l:jlist X),
    to_list (j_map f l) = map f (to_list l).
Proof.
intros X Y f l. induction l as [\[|x\]|l1 IHl1 l2 IHl2].
Case "j_nil".
    simpl. reflexivity.
Case "j_one".
    simpl. reflexivity.
Case "j_app".
    simpl. rewrite IHl1. rewrite IHl2. apply map_app.
Qed.

The j_app case of the j_map correctness proof makes use of two different induction hypotheses, called IHl1 and IHl2. Circle the correct statement of IHl1 used in this case of the proof.

i. IHl1: to_list (j_app l1 l2) = map f (to_list (j_app l1 l2))
ii. IHl1: to_list (j_map f l1) = map f (to_list l1)
iii. IHl1: forall l1:jlist X. to_list (j_map f l1) = map f (to_list l1)
iv. IHl1: forall l2:jlist X. to_list (j_app l1 l2) = map f (to_list (j_app l1 l2))

Circle the statement of the lemma map_app, necessary to complete the j_app case of the j_map correctness proof.

i. Lemma map_app : forall X Y (f:X -> Y) l1 l2,
    j_map f (j_app l1 l2) = j_app (j_map f l1) (j_map f l2)
ii. Lemma map_app : forall X Y (f:X -> Y) l1 l2,
    map f (l1 ++ l2) = j_map f (j_app l1 l2)
iii. Lemma map_app : forall X Y (f:X -> Y) l1 l2,
    map f l1 ++ map f l2 = j_app (j_map f l1) (j_map f l2)
iv. Lemma map_app : forall X Y (f:X -> Y) l1 l2,
    map f l1 ++ map f l2 = map f (l1 ++ l2).
5. (12 points) Write a *careful* informal proof of the following theorem. Make sure to state the induction hypothesis explicitly in the inductive step.

Theorem: Addition is commutative. For all $x$ and $y$, $x + y = y + x$.

In your proof, you may use the following lemmas

- Lemma *plus_n_0*: 0 is a right identity for addition. i.e. for all $n$, $n + 0 = n$.
- Lemma *plus_n_Sm*: The successor of $(n + m)$ is equal to $n$ plus the successor of $m$. 

6. (13 points) In this question, we’ll consider two different implementations of the same list function—one as an inductively defined relation and one as a Fixpoint.

(a) The function \texttt{f\_repeat} takes an element \texttt{x} and a number \texttt{n} and returns a list containing \texttt{n} copies of the element. For example:

\[
\begin{align*}
\texttt{f\_repeat true 3} &= [\texttt{true; true; true}] \\
\texttt{f\_repeat 4 0} &= [] \\
\end{align*}
\]

Complete the Fixpoint definition of \texttt{f\_repeat}.

\[
\texttt{Fixpoint f\_repeat \{X : Type\} (x : X) (n : nat) : list X :=}
\]

(b) Similarly, the relation \texttt{r\_repeat} is a three place relation that holds between an element \texttt{x}, a number \texttt{n}, and a list \texttt{xs} if and only if \texttt{xs} is the list obtained by repeating the element \texttt{n} times. For example, the following are provable instances of \texttt{r\_repeat}.

\[
\begin{align*}
\texttt{r\_repeat true 3 [true; true; true]} \\
\texttt{r\_repeat 4 0 []} \\
\end{align*}
\]

Complete an Inductive definition of \texttt{r\_repeat}. Note, your answer must not use \texttt{f\_repeat}.

\[
\texttt{Inductive r\_repeat \{X : Type\} : X -> nat -> list X -> Prop :=}
\]
(c) Suppose we want to show the equivalence between the functional definition of repetition and
the relational specification. As part of that, we should prove the following lemma:

Lemma repeat_f_to_r : forall X x n (l : list X),
    f_repeat x n = l -> r_repeat x n l.

An ill-advised proof of this lemma might start as follows:

Proof.
  intros X x n l H. induction n as [|n'].
  Case "0".
    admit. (* skipping base case for now. *)
  Case "n = S n'".
    destruct l as [|x0 l0].
    SCase "l=[]". simpl in H. inversion H.
    SCase "l=x0 :: l0".

At this point, the proof state looks like the following:

SCase := "l=x0 :: l0" : String.string
Case := "S n'" : String.string
X : Type
x : X
n' : nat
x0 : X
l0 : list X
H : f_repeat x (S n') = x0 :: l0
IHn' : f_repeat x n' = x0 :: 10 -> r_repeat x n' (x0 :: 10)

What are the next steps in the proof? What is the problem with this proof attempt after
those steps have been taken? How might this problem be resolved? Be specific. (Use the
next page if you need more space.)
(Extra space for the previous problem.)
For Reference

Inductive nat : Type :=
  | O : nat
  | S : nat -> nat.

Inductive and (P Q : Prop) : Prop :=
  conj : P -> Q -> (and P Q).

Notation "P \ /
  Q" := (and P Q) : type_scope.

Inductive True : Prop :=
  I : True.

Inductive False : Prop := .

Definition not (P:Prop) := P -> False.

Notation "~ x" := (not x) : type_scope.

Notation "x <> y" := (~ (x = y)) : type_scope.

Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.

Notation "x + y" := (plus x y)(at level 50, left associativity) : nat_scope.

Fixpoint mult (n : nat) (m : nat) : nat :=
  match n with
  | 0 => 0
  | S n' => m + (mult n' m)
  end.

Notation "x * y" := (mult x y)(at level 40, left associativity) : nat_scope.

Fixpoint beq_nat (n m : nat) : bool :=
  match n, m with
  | 0, 0 => true
  | S n', S m' => beq_nat n' m'
  | _, _ => false
  end.
Fixpoint ble_nat (n m : nat) : bool :=
    match n with
    | 0 => true
    | S n' =>
        match m with
        | 0 => false
        | S m' => ble_nat n' m'
        end
    end.

Inductive beautiful : nat -> Prop :=
    b_0 : beautiful 0
| b_3 : beautiful 3
| b_5 : beautiful 5
| b_sum : forall n m, beautiful n -> beautiful m -> beautiful (n+m).

Inductive list (X:Type) : Type :=
    | nil : list X
    | cons : X -> list X -> list X.

Fixpoint app (X : Type) (l1 l2 : list X) : (list X) :=
    match l1 with
    | nil => l2
    | cons h t => cons X h (app X t l2)
    end.

Notation "x ++ y" := (app x y) (at level 60, right associativity).

Fixpoint map {X Y:Type} (f:X->Y) (l:list X) : (list Y) :=
    match l with
    | [] => []
    | h :: t => (f h) :: (map f t)
    end.

Fixpoint filter {X:Type} (test: X->bool) (l:list X) : (list X) :=
    match l with
    | [] => []
    | h :: t => if test h then h :: (filter test t)
                else filter test t
    end.