(11 points) Put an X in the True or False box for each statement.

(a) There exists a proposition $P$ such that the proposition $\neg\neg P \leftrightarrow P$ is provable.

☒ True ☐ False

(b) For every number $n$, the boolean computation `evenb n` is reflected in the truth of the proposition $\text{exists } k, n = \text{double } k$.

☒ True ☐ False

(c) Suppose we have assumption $H : P \rightarrow Q \rightarrow R \rightarrow S$ and the current goal is $S$. If we do `apply H`, then the goal will change to $P \rightarrow Q \rightarrow R$.

☐ True ☒ False

(d) The type

```
Inductive Foo : Type :=
| MakeFoo (f : Foo).
```

has an infinite number of elements.

☐ True ☒ False

(e) The result of `Compute (In 1 [1;2])` is True.

☐ True ☒ False

(f) 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

(g) Coq’s termination checker uses a conservative syntactic condition to guarantee that all `Fixpoint` definitions define total functions: there must be one of the function’s arguments that is “decreasing” at each recursive call in the function’s body, in the sense that the argument to the recursive call is a “strict subterm” of the corresponding argument to the function itself.

☒ True ☐ False

(h) $f = g \rightarrow \forall x, f x = g x$ requires an additional axiom to prove in Coq.

☐ True ☒ False
(i) \((\forall x, f \ x = g \ x) \rightarrow f = g\) requires an additional axiom to prove in Coq.

- True  ✗  False

(j) Using the tactic \texttt{injection H} when \(H : \texttt{double } n = \texttt{double } m\) is in the proof context will introduce \(n = m\) as a hypothesis.

- True  ✗  False

(k) We’ve seen situations where using the tactic \texttt{rewrite H} will have exactly the same effect as using the tactic \texttt{apply H}.

- True  ✗  False

\textit{We’d intended “False” as the correct answer, but it was pointed out that there’s a rewrite in the proof of \texttt{iff_rewrite} in IndProp.v that can be replaced by apply with exactly the same effect, because of the special way apply and rewrite and apply work with iff hypotheses.}
(16 points) Write the type of each of the following Coq expressions (write “ill typed” if an expression does not have a type). You can find the definitions of \texttt{plus} and \texttt{fold} on pages 1 and ?? in the appendix.

(a) \texttt{fun (x y z : nat) => 1 + z}
\hspace{2cm} \textit{Answer: nat \rightarrow nat \rightarrow nat \rightarrow nat}

(b) \texttt{fun (b : bool) => if true then 1 else b}
\hspace{2cm} \textit{Answer: ill typed}

(c) \texttt{forall (X : Type) (x y : X), x * y = 5}
\hspace{2cm} \textit{Answer: ill typed}

(d) \texttt{forall (X : Type) (x y : X), x = y}
\hspace{2cm} \textit{Answer: Prop}

(e) \texttt{fold plus}
\hspace{2cm} \textit{Answer: list nat \rightarrow nat \rightarrow nat}

(f) \texttt{forall (X Y : Prop), X \rightarrow Y}
\hspace{2cm} \textit{Answer: Prop}

(g) \texttt{forall (x : nat) (y : Prop), x \rightarrow y}
\hspace{2cm} \textit{Answer: ill typed}

(h) \texttt{fun x => exists y, y <= x}
\hspace{2cm} \textit{Answer: nat \rightarrow Prop}
For each of the following propositions, check “not provable” if it is not provable (in Coq’s core logic, without additional axioms), “needs induction” if it is provable only using induction, or “easy” if it is provable without using induction and without additional lemmas.

(a) In 3 [1;2;3;4;5]
   □ not provable  □ needs induction  ☒ easy

(b) forall s, In 3 ([1;2;3] ++ s)
   □ not provable  □ needs induction  ☒ easy

(c) forall s, In 3 (s ++ [1;2;3])
   □ not provable  ☒ needs induction  □ easy

(d) exists s, In 3 (s ++ [1;2;3])
   □ not provable  □ needs induction  ☒ easy

(e) exists (x y : list nat), x ++ y = y ++ x
   □ not provable  □ needs induction  ☒ easy

(f) forall n, n+5 <= n+6
   □ not provable  ☒ needs induction  □ easy

(g) forall f g, (forall x, f x = g x) -> f = g
   ☒ not provable  □ needs induction  □ easy

(h) forall x y, x * y = y * x
   □ not provable  ☒ needs induction  □ easy

(i) forall P : Prop, P \/ \~P
   ☒ not provable  □ needs induction  □ easy

(j) forall P : Prop, P -> \~\~P
   □ not provable  □ needs induction  ☒ easy

(k) forall P : Prop, P
   ☒ not provable  □ needs induction  □ easy
(a) \textbf{option Prop} \\
Answer: \\
Some True.
(b) \textbf{forall (X : Type), (X -> Prop) -> list X -> Prop} \\
Answer: \\
\@All.
(c) \textbf{nat -> nat -> nat} \\
Answer: \\
\text{plus}.
(d) \textbf{list (nat -> nat -> nat)} \\
Answer: \\
\text{[plus; mult]}.
(e) \textbf{forall (A : Type), A} \\
Answer: \\
\text{empty}.
(f) \textbf{1 <= 3} \\
Answer: \\
\text{le_S 1 2 (le_S 1 1 (le_n 1))}.
(g) \textbf{forall (A B : Type), A -> B -> A} \\
Answer: \\
\text{fun A B a b => a}.
(h) \textbf{bool * Prop} \\
Answer: \\
\text{(true, False)}. 
(6 points) For each of the following, select the tactics which will prove the theorem.

Inductive ev : nat -> Prop :=
| ev_0 : ev 0
| ev_SS (n : nat) (H : ev n) : ev (S (S n)).

Inductive le : nat -> nat -> Prop :=
| le_n : forall n, le n n
| le_S : forall n m, (le n m) -> (le n (S m)).
Notation "m <= n" := (le m n).

(a) Which of these tactic scripts would prove the following theorem (there may be zero, one, or
more; check as many as apply)?

Theorem even_term :
forall n, ev n -> ev (S (S (S (S n)))).
□
intros n H. apply (ev_SS (ev_SS H)).
□
intros n H. apply (ev_SS (S (S n)) (ev_SS n H)).
□
intros n H. apply (ev_SS n (ev_SS (S n)) H)).
□
intros n H. apply (ev_SS (S (S (S n)))
(ev_SS (S (S n)) (ev_SS (S n) (ev_SS n H))).
□
intros n H. apply (ev_SS (ev_SS (ev_SS H))).

(b) Which of these tactic scripts would prove the following theorem (check as many as apply)?

Theorem le_term :
1 <= 3.
□
apply (le_S 1 1 (le_S 2 1 (le_n 1))).
□
apply (le_S (le_S le_n)).
□
apply (le_S 2 1 (le_S 1 1 (le_n 1))).
□
apply (le_n (le_S le_S)).
□
apply (le_S 1 2 (le_S 1 1 (le_n 1))).
This problem asks you to translate mathematical ideas from English into Coq.

(a) (4 points) Complete the following definition of a property characterizing the prime natural numbers (i.e., those numbers that cannot be expressed as the product of two other numbers, both strictly greater than 1).

\[
\text{Definition prime (n: nat) : Prop :=} \\
\quad \text{not (exists x y, x > 1 \&\& y > 1 \&\& n = x * y).}
\]

\[
\text{Definition prime1 (n: nat) : Prop :=} \\
\quad \text{not (exists x y, n = (2+x) * (2+y)).}
\]

\[
\text{Definition prime2 (n: nat) : Prop :=} \\
\quad \forall n1 n2, n1 < n -> n2 < n -> 1 < n1 -> 1 < n2 -> n1 * n2 <> n.
\]

\[
\text{Definition prime3 (n: nat) : Prop :=} \\
\quad \forall x y, x > 1 -> y > 1 -> n <> x * y.
\]

Note: Our wording of the question was a bit wrong (we neglected to specify that prime numbers should also be strictly greater than 1), so we accepted answers that counted 1 as either prime or not prime.

(b) (4 points) A binary tree with labels of type A is either empty or a branch that has some value of type A along with two binary trees of type A as children. Its formal definition is provided on page 2 of the appendix. In the following example, exTree on the left corresponds to the tree on the right:

\[
\text{Example exTree : tree nat :=} \\
\quad \text{Branch 5} \\
\quad \quad \text{(Branch 2} \\
\quad \quad \quad \text{(Branch 1 Empty Empty)} \\
\quad \quad \quad \text{(Branch 4 Empty Empty))} \\
\quad \text{(Branch 9} \\
\quad \quad \text{Empty} \\
\quad \quad \text{(Branch 7 Empty Empty))}.
\]

Now, suppose we define a predicate EveryLabel of the form:

\[
\text{Fixpoint EveryLabel \{A : Type\} (P : A -> Prop) (t: tree A) : Prop :=} \\
\quad \text{match t with} \\
\quad \quad \text{| Empty => True} \\
\quad \quad \text{| Branch x l r => P x \&\& EveryLabel P l \&\& EveryLabel P r} \\
\quad \end.
\]

For example,

\[
\text{EveryLabel (fun x => x <= 10) exTree}
\]

is provable, while

\[
\text{not (EveryLabel (fun x => ev x) exTree)}
\]
is not.

A tree is called a *binary search tree* if for all nodes \((\text{Branch } x \ l \ r)\) in the tree, all values stored in the left subtree \(l\) are less than or equal to the node’s value \(x\), and all values stored in the right subtree \(r\) are greater than \(x\). For example, the tree shown above is *not* a binary search tree, but it will become one if we change the label 7 to 10.

Use \texttt{EveryLabel} to complete the definition of an inductive property \texttt{BST} that characterizes binary search trees.

\begin{verbatim}
Inductive BST : tree nat -> Prop :=
| Empty_BST : BST Empty
| Branch_BST :
  forall x l r, 
  BST l -> BST r ->
  EveryLabel (gt x) l -> EveryLabel (le x) r ->
  BST (Branch x l r).
\end{verbatim}

(c) (5 points) Continuing with binary trees, in this part, we will next define a relation \texttt{mirror} that relates two trees that are “mirror images” of each other. For example, the following should be provable:

\begin{verbatim}
Example mirror_eg :
  mirror (Branch 1 Empty (Branch 2 (Branch 3 Empty Empty) Empty))
  (Branch 1 (Branch 2 Empty (Branch 3 Empty Empty)) Empty).
\end{verbatim}

This says that the following two mirrored trees are mirrors of each other:

\begin{center}
\begin{tikzpicture}[level distance=1.5cm,sibling distance=1.5cm]
  \node {1} child {node {2} child {node {3}}}
  \node {1} at (3,0) child {node {2} child {node {3}}}
\end{tikzpicture}
\end{center}

Here is a skeleton for \texttt{mirror}; please fill in the missing part.

\begin{verbatim}
Fixpoint mirror {A: Type} (t1 t2 : tree A) : Prop :=
  match t1,t2 with
  (* FILL IN HERE *)
  | Empty, Empty => True
  | Branch a1 t11 t12, Branch a2 t21 t22 =>
    a1 = a2 /
    mirror t11 t22 /
    mirror t12 t21
  | _, _ => False
end.
\end{verbatim}
(d) (7 points) Continuing with binary trees, here is a way of representing a “path” in a binary tree.

```coq
Inductive path :=
| Stop
| GoLeft (p : path)
| GoRight (p : path).
```

In this problem, we will define a ternary relation `valueAtPath` that relates a tree and a path to the value (if any) picked out by following that path in the tree. For example, the following should both be provable, where `t` is the tree pictured on the right:

```coq
Definition t :=
  Branch 1 Empty (Branch 2 (Branch 3 Empty Empty) Empty).

Example VAP_eg1 : valueAtPath t Stop 1.

Example VAP_eg2 : valueAtPath t (GoRight (GoLeft Stop)) 3.
```

Here is a skeleton for `valueAtPath`; please fill in the missing bits.

```coq
Inductive valueAtPath {A: Type} : tree A -> path -> A -> Prop :=
| Here : forall (t1 t2: tree A) (a: A),
  (* FILL IN HERE *)
  valueAtPath (Branch a t1 t2) Stop a
| InLeft : forall (t1 t2: tree A) (b: A) (p: path) (a: A),
  (* FILL IN HERE *)
  valueAtPath t1 p a ->
  valueAtPath (Branch b t1 t2) (GoLeft p) a
| InRight : forall (t1 t2: tree A) (b: A) (p: path) (a: A),
  (* FILL IN HERE *)
  valueAtPath t2 p a ->
  valueAtPath (Branch b t1 t2) (GoRight p) a.
```
Recall the polymorphic fold function.

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

The arguments to fold are an aggregating function \(f : X \rightarrow Y \rightarrow Y\), a list \(l : list X\), and an initial value \(b : Y\); given these, it “folds through the list” applying the function \(f\) on successive elements. Here is an example use of fold, showing several evaluation steps.

\[
\begin{align*}
\text{fold app } &[[1], [], [2,3], [4]] [] \\
= &\text{app [1] (fold app } [[], [2,3], [4]] []) \\
= &... \\
= &[1,2,3,4]
\end{align*}
\]

Suppose we define a variant reverse_fold that folds a given list while reversing its elements.

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

For the same \(f\) and \(l\) as before, the output of reverse_fold would be:

\[
\begin{align*}
\text{reverse_fold app } &[[1], [], [2,3], [4]] [] \\
= &\text{reverse_fold app } [[], [2,3], [4]] (app [1] []) \\
= &\text{reverse_fold app } [[2,3], [4]] (app [] (app [1] [])) \\
= &\text{reverse_fold app } [[4]] (app [2,3] (app [] (app [1] []))) \\
= &... \\
= &[4,2,3,1]
\end{align*}
\]

Finally, recall the polymorphic rev function that reverses the elements of a list.

```ocaml
Fixpoint rev {X} (l : list X) : list X :=
  match l with
  | [] => []
  | h :: t => rev t ++ [h]
end.
```

(The definitions of fold, reverse_fold, and rev can also be found on page 2 in the appendix.)

The following theorem intuitively says that “the result of applying reverse_fold on a list \(l\) is the same as the result of applying fold to the reverse of list \(l\).”

**Theorem reverse_fold_rev_fold :**

\[
\forall X Y \quad (f : X \rightarrow Y \rightarrow Y) \quad (b : Y) \quad (l : list X), \\
\text{reverse_fold } f \ l \ b \ = \ \text{fold } f \ (\text{rev } l) \ b.
\]

On the next page, write a careful, informal proof of this theorem. If your proof uses induction, be sure to spell out the induction hypothesis explicitly. You may find it helpful to define an auxiliary lemma characterizing how fold works when applied to a list of the form \(l1 \ ++ \ l2\). If so, please
Theorem reverse_fold_rev_fold :
  forall X Y (f : X -> Y -> Y) (b : Y) (l : list X),
  reverse_fold f l b = fold f (rev l) b.

Proof: In order to prove the given theorem, we will first prove an auxiliary lemma.

Lemma fold_last_elem :
 forall X Y (f : X -> Y -> Y) (b : Y) (l : list X) (x : X),
  fold f (l ++ [x]) b = fold f l (f x b).

Proof for Lemma: (Provided here just for completeness: As the instructions say, we did not expect answers to include this.) By induction on l.

• Suppose l = []. By the definitions of ++ and fold, both fold f ([] ++ [x]) b and fold f [] (f x b) reduce to f x b.

• Suppose l = h::l' and we have the induction hypothesis

  fold f (l' ++ [x]) b = fold f l' (f x b).

We can reduce fold f ((h::l') ++ [x]) b to f h (fold f (l' ++ [x]) b) by the definitions of ++ and fold. Similarly, fold f (h::l') (f x b) reduces to f h (fold f l' (f x b)). The reduced expressions are equal by the induction hypothesis.

Proof of Theorem: We show, by induction on l, that forall (b : Y), reverse_fold f l b = fold f (rev l) b.

• Suppose l = []. By the definitions of rev, fold, and reverse_fold, both reverse_fold f [] b and fold f (rev []) b reduce to b.

• Suppose l = h::l' and we have the induction hypothesis

  forall b', reverse_fold f l' b' = fold f (rev l') b'.

We can reduce reverse_fold f (h::l') b to reverse_fold f l' (f h b) by the definition of reverse_fold. Similarly, fold f (rev (h::l')) b reduces to fold f ((rev l') ++ [h]) b, which can be rewritten using the lemma fold_last_elem to fold f (rev l') (f h b). The reduced expressions are equal by the induction hypothesis when b' is instantiated as (f a b).
Use this space for scratch work that you don't want graded. If you write something here that you do want graded, make sure there is a very clear pointer from the earlier page where you ran out of space.
For Reference

Numbers

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

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 double (n:nat) :=
match n with
| O => O
| S n' => S (S (double n'))
end.

Fixpoint evenb (n : nat) : bool :=
match n with
| O => true
| S O => false
| S (S n') => evenb n'
end.

Inductive le : nat -> nat -> Prop :=
| le_n : forall n, le n n
| le_S : forall n m, (le n m) -> (le n (S m)).
Notation "m <= n" := (le m n).

Inductive ev : nat -> Prop :=
| ev_O : ev 0
| ev_SS (n : nat) (H : ev n) : ev (S (S n)).

Pairs

Inductive prod (A : Type) (B : Type) :=
| pair : A -> B -> prod A B.

Arguments pair {A B}.
Notation "A * B" := (prod A B) (at level 40, left associativity) : type_scope.
Notation "( x , y )" := (pair x y) (at level 0) : core_scope.
Lists

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

Arguments nil {X}.
Arguments cons {X} _ _.
Notation "[ ]" := nil.
Notation "x :: l" := (cons x l) (at level 60, right associativity).
Notation "[ x ; .. ; y ]" := (cons x .. (cons y nil) ..).

Fixpoint app {X : Type} (l1 l2 : list X) : (list X) :=
  match l1 with
  | [] => l2
  | h :: t => h :: (app t l2)
  end.
Notation "x ++ y" := (app x y) (at level 60, right associativity).

Fixpoint rev {X} (l : list X) : list X :=
  match l with
  | [] => []
  | h :: t => rev t ++ [h]
  end.

Fixpoint fold {X Y: Type} (f: X -> Y -> Y) (l: list X) (b: Y) : Y :=
  match l with
  | nil => b
  | h :: t => f h (fold f t b)
  end.

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

Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] => False
  | x' :: l' => x' = x \ In x l'
  end.

Binary Trees

Inductive tree (A : Type) :=
| Empty : tree A
| Branch : A -> tree A -> tree A -> tree A.

Arguments Empty {A}.
Arguments Branch {A} _ _ _.