Before beginning the exam, please write your PennKey (login ID) at the top of each even-numbered page (so that we can find things if a staple fails!).
(6 points) Put an X in the True or False box for each statement.

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

(b) If foo is an inductive type and \( f : \text{nat} \rightarrow \text{foo} \) is one of its constructors, and if \( H : f \ x = f \ y \) is an assumption in the current context, then injection \( H \) will introduce an assumption that \( x = y \).
☑ True ☐ False

(c) If \( X \) is an inductive type with at least one constructor, then there is at least one constructor expression \( e \) such that \( e : X \).
☐ True ☑ False

(d) If \( S \ (n + m) = n + S \ m \) is the goal in the current proof state, then reflexivity will solve the goal.
☐ True ☑ False

(e) For any number \( n \), if the goal in the current proof state is \( S \ n = 0 \), then using the tactic discriminate will solve the goal.
☐ True ☑ False

(f) For any propositions \( P \) and \( Q \), if \( H : P \rightarrow Q \) is an assumption in the current context and the goal in the current proof state is \( P \), then apply \( H \) will solve the goal.
☐ True ☑ False
Write the type of each of the following Coq expressions (write “ill typed” if an expression does not have a type). For part e you can find the definition of repeat on page 2 in the appendix.

(a) 1 + 1 = 2
   \textit{Answer: Prop}

(b) \texttt{fun (x y : nat) => x + 1 = y}
   \textit{Answer: nat -> nat -> Prop}

(c) \texttt{forall (X : Type) (x : X), x}
   \textit{Answer: ill typed}

(d) \texttt{fun (m : nat) => fun (n : nat) => m * m + n}
   \textit{Answer: nat -> nat -> nat}

(e) \texttt{repeat 3}
   \textit{Answer: nat -> list nat}

(f) \texttt{forall (X : Type), forall (Y : Type), forall (x : X), x = x}
   \textit{Answer: Prop}

(g) \texttt{fun (m : nat) => m :: m :: m}
   \textit{Answer: ill typed}

(h) \texttt{match [1;2] with [] => true | x::l => x+1 end}
   \textit{Answer: ill typed}
Standard Track Only] (16 points) For each of the types below, write a Coq expression that has that type or write “empty” if there are no such expressions.

(a) list (list nat)
    Answer: [[0]]

(b) nat -> list nat -> nat
    Answer: fun n => fun l => n

(c) bool -> bool -> bool
    Answer: andb

(d) forall (X : Type) (l1 l2 : list X), list X
    Answer: app

(e) forall (X : Type), X -> X -> X
    Answer: fun X (x1 x2 : X) => x1

(f) Prop -> (bool -> bool)
    Answer: fun P => negb

(g) Prop * Prop
    Answer: (True,False)

(h) list (forall X, X -> list X)
    Answer: [fun X => fun x:X => [x]] (or just nil!)
Recall the polymorphic fold and ++ functions (see page 2 in the appendix).

The following theorem intuitively says that “fold commutes with ++”—that is, if we have two lists, \(l_1\) and \(l_2\), a function \(f\), and a starting value \(b\), then either we can append the lists together and then fold \(f\) over them or else we can fold \(f\) over \(l_2\) and then fold \(f\) again over \(l_1\), using the previous result as the starting point.

Theorem app_comm_fold :
\[
\forall \{X Y\} (f: X\to Y\to Y) l_1 l_2 b, \\
fold f (l_1 ++ l_2) b = fold f l_1 (fold f l_2 b).
\]

Write a careful, informal proof of this theorem. If your proof uses induction, be sure to spell out the induction hypothesis explicitly.

Proof: By induction on \(l_1\).

- Suppose \(l_1 = []\). By the definitions of ++ and fold, both \(\text{fold } f (l_1 ++ l_2) b\) and \(\text{fold } f l_1 (\text{fold } f l_2 b)\) reduce to \(\text{fold } f l_2 b\).
- Suppose \(l_1 = x::l_1'\) and we have the induction hypothesis

\[
\text{fold } f (l_1' ++ l_2) b = \text{fold } f l_1' (\text{fold } f l_2 b).
\]

We can reduce \(\text{fold } f ((x::l_1') ++ l_2) b\) to \(f x (\text{fold } f (l_1' ++ l_2) b)\) By the definitions of ++ and fold. Similarly, \(\text{fold } f (x::l_1') (\text{fold } f l_2 b)\) reduces to \(f x (\text{fold } f l_1' (\text{fold } f l_2 b))\). The reduced expressions are equal by the induction hypothesis.
This problem asks you to translate mathematical ideas from English into Coq.

(a) (5 points) The concept of “prefix of a list” can be defined as an inductive relation as follows:

```
Inductive prefix {X : Type} : list X -> list X -> Prop :=
| prefix_nil : forall l, prefix [] l
| prefix_cons : forall t l h, prefix t l -> prefix (h::t) (h::l).
```

For example, these are all the prefixes of [1;2;3] (i.e., all the lists ls such that \texttt{prefix ls [1;2;3]} is provable):

- \[\]
- \[1\]
- \[1;2\]
- \[1;2;3\]

Conversely, a \textit{suffix} of a list is a substring that occurs at the end of the larger string. For example, here are all the suffixes of \([1;2;3]\):

- \[\]
- \[3\]
- \[2;3\]
- \[1;2;3\]

Complete the following inductive definition of the suffix relation, where \texttt{suffix s l} indicates that \texttt{s} is a suffix of \texttt{l}.

```
Inductive suffix {A : Type} : list A -> list A -> Prop :=
| suffix_id : forall l, suffix l l
| suffix_tail : forall s t h, suffix s t -> suffix s (t ++ [h]).
```

(b) (8 points) A \textit{binary tree} with labels of type \texttt{A} is either \texttt{empty} or a \texttt{branch} that has some value of type \texttt{A} along with two binary trees of type \texttt{A} as children. Its formal definition is provided on page 4 of the appendix. For example, the definition

```
Example exTree : tree nat :=
  Branch 5
  (Branch 2
    (Branch 1 Empty Empty)
    (Branch 4 Empty Empty))
  (Branch 9
    Empty
    (Branch 7 Empty Empty)).
```
represents the following binary tree:

A subtree of a binary tree is either the tree itself or a subtree of its child. For example, here are all the subtrees of `exTree`:

Give an inductive definition of the subtree relation on binary trees, where `subtree s t` indicates that `s` is a subtree of `t`.

**Inductive subtree {A : Type} : tree A -> tree A -> Prop :=**

**Answer:**

| subtree_id : forall t, subtree t t |
| subtree_lt : forall s l a r, subtree s l -> subtree s (Branch a l r) |
| subtree_rt : forall s r a l, subtree s r -> subtree s (Branch a l r). |

*Or*

| subtree_id : forall t, subtree t t |
| subtree_lt : forall s l a r, subtree (Branch a l r) s -> subtree l s |
| subtree_rt : forall s r a l, subtree (Branch a l r) s -> subtree r s. |
(4 points) Recall the definition of the \texttt{In} relation between values of type \( A \) and lists with elements of type \( A \):

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

Suppose we want to define an analogous relation \texttt{NotIn} such that \texttt{NotIn} \( x \) \( l \) is true just when \( x \) does not appear in \( l \).

Below are several candidates for how we might define this relation. Check the boxes next to those that are correct—i.e., the ones where \texttt{NotIn} \( x \) \( l \) is provable exactly when \( \neg (\text{In} x l) \) is.

- \(\checkmark\) \texttt{Fixpoint NotIn \{A : Type\} (x : A) (l : list A) : Prop :=}
  \begin{array}{l}
  \text{match l with} \\
  \text{| [] => False} \\
  \text{| x' :: l' => x' <> x \land NotIn x l'}
  \end{array}

- \(\square\) \texttt{Inductive NotIn \{A : Type\} : A -> list A -> Prop :=}
  \begin{array}{l}
  | NotInHere : forall x x' l', x <> x' -> NotIn x (x' :: l') \\
  | NotInThere : forall x x' l', NotIn x l' -> NotIn x (x' :: l').
  \end{array}

- \(\square\) \texttt{Definition NotIn \{A : Type\} := fun (x : A) (l : list A) =>}
  \begin{array}{l}
  \text{forall x', In x' l -> x <> x' .}
  \end{array}

- \(\square\) \texttt{Definition NotIn \{A : Type\} := fun (x : A) (l : list A) =>}
  \begin{array}{l}
  \text{forall x', In x' l \lor x <> x' .}
  \end{array}

7
(9 points)

(a) Suppose Coq’s current goal state looks like this:

\[
\begin{align*}
P, Q, R : \text{Prop} \\
H_1 : P \lor Q \\
H_2 : Q \land R \\
H_3 : \forall S : \text{Prop}, S \rightarrow P \lor S \\
\end{align*}
\]

\[
\begin{align*}
P \lor \neg Q \\
\end{align*}
\]

i. If we give the command “\texttt{split}”, what will happen?

- Error
- Nothing (no error, but no change to the state)
- “No more subgoals”
- Goal changes to \( P \)
- Goal changes to \( \neg Q \)

ii. If we give the command “\texttt{apply H3}”, what will happen?

- Nothing (no error, but no change to the state)
- “No more subgoals”
- Goal changes to \( P \lor S \)
- Goal changes to \( P \)
- Goal changes to \( \neg Q \)
- Goal changes to \( Q \)

iii. If we give the command “\texttt{destruct H1}”, what will happen?

- Nothing (no error, but no change to the state)
- “No more subgoals”
- \( H_1 \) is replaced by two hypotheses, \( H : P \) and \( H_1 : Q \)

\( \checkmark \) Current goal is replaced by two subgoals, one with hypothesis \( H : P \) and one with hypothesis \( H : Q \)
(b) Suppose Coq’s current goal state looks like this:

\[
\begin{align*}
\text{m, n : nat} \\
\text{H : forall x, x + m = m + x} \\
\text{R : forall a b c, S a = b + S c \rightarrow a = b + c} \\
\text{=============================} \\
\text{S (m + n) = n + S m}
\end{align*}
\]

i. If we give the command “\text{induction n as \[| n' \text{ IHn'}\]”}, we will get two subgoals. What will the second one look like?

\begin{itemize}
  \item New hypothesis \text{IHn'} : S (m + n') = n' + S m
  \item New hypothesis \text{IHn'} : forall m, S (m + n') = n' + S m
  \item New hypothesis \text{IHn'} : forall m, S (m + n') = S n' + S m
  \item New hypothesis \text{IHn'} : S (m + S n') = S n' + S m
  \item New hypothesis \text{IHn'} : forall m, S (m + S n') = S n' + S m
\end{itemize}

and the goal changes to

\begin{itemize}
  \item \text{forall m, S (m + S n') = S n' + S m}
  \item \text{forall m, S (m + n') = n' + S m}
\end{itemize}

ii. If we give the command “\text{rewrite -> H}”, what will happen?

\begin{itemize}
  \item Error
  \item Nothing (no error, but no change to the state)
  \item Goal changes to \text{S (n + m) = n + S m}
  \item Goal changes to \text{S (n + m) = S m + n}
  \item Goal changes to \text{S (m + n) = S m + n}
\end{itemize}

iii. If we give the command “\text{apply R}”, what will happen?

\begin{itemize}
  \item Error
  \item Nothing (no error, but no change to the state)
  \item Goal changes to \text{m + n = n + m}
  \item Goal changes to \text{S (S (m + n)) = n + S (S m)}
  \item No more subgoals
\end{itemize}

9
In this problem you will help write a simple regular expression matching function, \texttt{re\_match}. The expression \texttt{re\_match s re} should yield 	exttt{true} if and only if the “string” \texttt{s} (a list of \texttt{nat}s) matches the regular expression \texttt{re}. For example:

\begin{verbatim}
re_match [1; 3] (App (Union (Char 1) (Char 2)) (Char 3)) = true
re_match [1; 2; 3] (App (Union (Char 1) (Char 2)) (Char 3)) = false
\end{verbatim}

Complete the recursive definition of the function \texttt{re\_match} on the following page, filling in the branches for \texttt{EmptySet}, \texttt{Char}, \texttt{App} and \texttt{Union}. Your answers may use any function defined in the appendix. In particular, the following functions might be useful:

- \texttt{list\_nat\_eq : list nat -> list nat -> bool} checks whether two lists are equal.
- \texttt{existsb : forall X : Type, (X -> bool) -> list X -> bool} checks whether at least one element in a list \texttt{l : list X} satisfies a boolean property \texttt{p : X -> bool}.
- \texttt{forallb : forall X : Type, (X -> bool) -> list X -> bool} checks whether all elements of a list \texttt{l : list X} satisfy a boolean property \texttt{p : X -> bool}.
- \texttt{splits : forall X : Type, list X -> list (list X * list X)} enumerates all possible ways to split a list \texttt{s} into two lists \texttt{(s1, s2)} such that \texttt{s1 ++ s2 = s}. For example:

\begin{verbatim}
splits [1; 2; 3] = [([], [1; 2; 3]); ([1], [2; 3]); ([1; 2], [3]); ([1; 2; 3], [])]
\end{verbatim}

The function \texttt{splits\_many} that is used in the \texttt{Star} case is a kind of \texttt{n}-way generalization of \texttt{splits}. You don’t need to understand it in detail, but it enumerates all possible ways to split a list \texttt{s} into one or more nonempty lists \texttt{s1}, \ldots, \texttt{sn} such that \texttt{s1 ++ \ldots ++ sn = s}. For example:

\begin{verbatim}
splits\_many [1; 2; 3] = [[[1]; [2]; [3]];
[[1]; [2; 3]];
[[1; 2]; [3]];
[[1; 2; 3]]
\end{verbatim}

\begin{verbatim}
Fixpoint re_match (s : list nat) (re : reg\_exp) : bool :=
  match re with
  | EmptySet => false
  | EmptyStr => list\_nat\_eq s []
  | Char n => list\_nat\_eq s [n]
  | App re1 re2 =>
    existsb (fun '(s1, s2) => re_match s1 re1 && re_match s2 re2)
    (splits s)
  | Union re1 re2 => re_match s re1 || re_match s re2
\end{verbatim}
| Star re’ =>
  list_nat_eq s [] ||
  existsb (fun ss’ => forallb (fun s’ => re_match s’ re’) ss’)
  (splits_many s)
end.
For Reference

Booleans

Inductive bool : Type :=
| true
| false.

Definition negb (b : bool) : bool :=
  match b with
  | true => false
  | false => true
  end.

Definition andb (b c : bool) : bool :=
  match b with
  | true => c
  | false => false
  end.

Definition orb (b c : bool) : bool :=
  match b with
  | false => c
  | true => true
  end.

Notation "b && c" := (andb b c) (at level 50, left associativity) : bool_scope.
Notation "b || c" := (orb b c) (at level 40, left associativity) : bool_scope.

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 eqb (n m : nat) : bool :=
match n, m with
| 0, 0 => true
| S n', S m' => eqb n' m'
| _, _ => false
end.
Notation "x =? y" := (eqb x y)(at level 70) : nat_scope.

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_0 : ev 0
| ev_SS : forall n : nat, 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 repeat (n count : nat) : list nat :=
match count with
| 0 => nil
| S count' => n :: (repeat n count')
end.
Fixpoint In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] => False
  | x' :: l' => x' = x \ In x l'
  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 forallb {X : Type} (test : X -> bool) (l : list X) : bool :=
  match l with
  | [] => true
  | x :: l' => andb (test x) (forallb test l')
  end.

Fixpoint existsb {X : Type} (test : X -> bool) (l : list X) : bool :=
  match l with
  | [] => false
  | x :: l' => orb (test x) (existsb test l')
  end.

Fixpoint list_nat_eq (s1 s2 : list nat) : bool :=
  match s1, s2 with
  | n1 :: t1, n2 :: t2 => (n1 =? n2) && list_nat_eq t1 t2
  | [], [] => true
  | _, _ => false
  end.

(* Enumerate all ways to split a list into two. *)
Fixpoint splits {X : Type} (s : list X) : list (list X * list X) :=
  match s with
  | [] => [([]), []]
  | h :: t => ([], s) :: map (fun '(s1, s2) => (h :: s1, s2)) (splits t)
  end.

(* Enumerate all ways to split a list into _any number_ of _nonempty_ lists. *)
Fixpoint splits_many {X : Type} (s : list X) : list (list (list X)) :=
  match s with
  | [] => []
  | h :: [] => [[s]]
  | h :: t =>
    map (fun t' => [h] :: t') (splits_many t) ++
    map (fun t' =>
        match t' with
        | [] => t'
        | u :: w => (h :: u) :: w
      end) (splits_many t)
  end.
Regular Expressions

Inductive reg_exp {T : Type} : Type :=
| EmptySet : reg_exp
| EmptyStr : reg_exp
| Char : T -> reg_exp
| App : reg_exp -> reg_exp -> reg_exp
| Union : reg_exp -> reg_exp -> reg_exp
| Star : reg_exp -> reg_exp.

Inductive exp_match {T: Type} : list T -> reg_exp -> Prop :=
| MEmpty : exp_match [] EmptyStr
| MChar : forall x, exp_match [x] (Char x)
| MApp : forall s1 re1 s2 re2,
        exp_match s1 re1 ->
        exp_match s2 re2 ->
        exp_match (s1 ++ s2) (App re1 re2)
| MUnionL : forall s1 re1 re2,
          exp_match s1 re1 re2,
          exp_match s1 (Union re1 re2)
| MUnionR : forall re1 s2 re2,
          exp_match s2 re2 ->
          exp_match s2 (Union re1 re2)
| MStar0 : forall re, exp_match [] (Star re)
| MStarApp : forall s1 s2 re,
            exp_match s1 re ->
            exp_match s2 (Star re) ->
            exp_match (s1 ++ s2) (Star re).

Notation "s =~ re" := (exp_match s re) (at level 80).

Binary Trees

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

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