CIS 500: Software Foundations **Midterm I**

February 23, 2016

(Standard and Advanced versions together)

Name (printed): ______________________________

Username (login id): ______________________________

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: ______________________________ Date: ____________________

Directions: This exam booklet contains both the standard and advanced track questions. Questions with no annotation are for *both* tracks. Other questions are marked "Standard Only" or "Advanced Only". *Do not do the questions intended for the other track.*

Mark the box of the track you wish to follow.

☐ Standard

1	/10
2	/15
3	/8
4	/12
5	ADVANCED ONLY/–
6	/18
7	/17
Total	/80

☐ Advanced

1	/10
2	/15
3	STANDARD ONLY/–
4	STANDARD ONLY/–
5	/20
6	/18
7	/17
Total	/80

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

(a) For any `x` and `y` of type `X`, it is possible to define a proposition that holds when `x` is equal to `y`.

True False

(b) A *polymorphic* type is one that is parameterized by a type argument by using the universal quantifier `forall`. For instance: `forall (X:Type), list X -> list X` is a polymorphic type.

True False

(c) Coq is a constructive logic, which implies that it is not possible to prove (without using extra axioms) the law of excluded middle: `forall P : Prop, P \/ ~P`.

True False

(d) The axiom of *functional extensionality* states that

```
forall (A B:Type) (f g: A -> B), f = g <-> (forall x : A, f x = g x)
```

True False

(e) In Coq, the proposition `False` and the boolean `false` are logically equivalent—i.e. one can prove `False <-> false`.

True False

(f) There is exactly one *canonical proof* of the proposition `beautiful 0` according to the inductive definition of `beautiful : nat -> Prop` given in the appendix.

True False

(g) There are infinitely many *canonical proofs* of the proposition `le 3 4` (or, equivalently, `3 <= 4`) according to the inductive definition of `le : nat -> nat -> Prop` given in the appendix.

True False

(h) If the term `(In 3 [1;2;3])` is the goal of your proof state, using the tactic `simpl` will simplify it to `True`. (The definition of `In` is given in the appendix.)

True False

(i) In Coq all functions terminate (i.e. they cannot go into an infinite loop on any input).

True False

(j) A boolean function `f : nat -> bool` *reflects* a proposition `P : nat -> Prop` exactly when `forall (n:nat), (f n = true) <-> P n`.

True False

2. (15 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) `beq_nat 3`

(b) `3=4 -> False`

(c) `fun (X:Type) => fun (l:list X) => X :: l`

(d) `forall (x:nat), beq_nat x 3 = false`

(e) `fun (x:nat) => b_3`

Note: `b_3` is one of the constructors for the inductively-defined proposition `beautiful` shown in the appendix.

3. [**Standard Only**] (8 points) For each of the types below, write a Coq expression that has that type or write "Empty" if there are no such expressions. (The references section contains the definitions of `<=` and other functions and propositions.)

(a) `forall (X Y:Type), list X -> list Y`

(b) `(nat -> nat) -> nat`

(c) `3 <= 3`

(d) `4 <= 3`

4. [**Standard Only**] (12 points) For each of the given theorems, which set of tactics is needed to prove it besides `intros` and `reflexivity`? If more than one of the sets of tactics will work, choose the smallest set. Note that each proof should be completed directly, without the help of any lemmas.

(a) `Theorem mult_0_l : forall n:nat, 0 * n = 0.`

  i. `induction` and `rewrite`
  ii. `rewrite` and `simpl`
  iii. `inversion`
  iv. no additional tactics are necessary

(b) `Theorem distinct_nats : ~(3 = 4).`

  i. `induction` and `rewrite`
  ii. `unfold not` and `rewrite`
  iii. `unfold not` and `inversion`
  iv. no additional tactics are necessary

(c) `Lemma or_comm : forall P Q : Prop,  P \/ Q -> Q \/ P.`

  i. `inversion` and `apply`
  ii. `inversion`, `split`, and `apply`
  iii. `inversion`, `left`, `right` and `apply`
  iv. no additional tactics are necessary

(d)
```
Lemma app_assoc : forall X (l1 l2 l3: list X),
                    l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3.
```

  i. `simpl`, `rewrite`, and `induction l1`
  ii. `simpl`, `rewrite`, and `induction l2`
  iii. `simpl`, `rewrite`, `induction l2`, and `generalize dependent l1`
  iv. `simpl`, `rewrite`, and `induction l3`

5. [**Advanced Only**] (20 points) Write a *careful* informal proof of the following theorem. Make sure to state the induction hypothesis explicitly in the inductive step.

> **Theorem** *Dichotomy* : For all natural numbers `n` and `m`, either `n <= m` or `m <= n`.
>
> In your proof, you may use the following two lemmas:
> **Lemma** `le_0_n`: For all natural numbers `n`, `0 <= n`.
> **Lemma** `le_n_S`: For all natural numbers `n` and `m`, if `n <= m` then `S n <= S m`.
>
> **Proof:**

6. (18 points) Consider the following datatype of inductively-defined *binary trees*, which are either empty, or nodes containing a data element of type `X` and a left tree and a right tree.

```
Inductive tree X : Type :=
| empty : tree X
| node : tree X -> X -> tree X -> tree X.

(* Make the type parameter implicit. *)
Arguments empty {X}.
Arguments node {X} _ _ _.
```

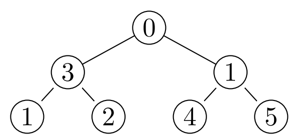
It is helpful to define a helper function called `leaf` that builds a one-node tree:

```
Definition leaf {X} (x:X) := node empty x empty.
```

Using `leaf` we can build a bigger tree like `t1` defined below:

```
Definition t1 : tree nat :=
  node (node (leaf 1) 3 (leaf 2)) 0 (node (leaf 4) 1 (leaf 5)).
```

Pictorially, we might draw `t1` like this: (note that we don't depict the `empty` constructors)



(a) The following function produces the *in-order traversal* of the elements in the nodes of a tree:

```
Fixpoint in_order {X} (t:tree X) : list X :=
  match t with
    | empty => []
    | node lt x rt => (in_order lt) ++ [x] ++ (in_order rt)
  end.
```

Which of the following is the result of `Eval compute in (in_order t1)`?

i. `[0;1;1;2;3;4;5]`
ii. `[0;3;1;2;4;1;5]`
iii. `[1;3;2;0;4;1;5]`
iv. `[1;4;2;5;3;1;0]`

(b) Complete the following definition of `tree_map`, which, like the `map` function for lists, applies a function `f` to each element in the tree. Your solution should pass the tests given below.

```
Fixpoint tree_map {X Y} (f:X -> Y) (t:tree X) : tree Y :=
```

```
Example tm_test_1 : tree_map (fun x => x + 2) empty = empty.
Proof. simpl. reflexivity. Qed.

Example tm_test_2 : tree_map (fun x => x + 2) t1 =
                    node (node (leaf 3) 5 (leaf 4)) 2 (node (leaf 6) 3 (leaf 7)).
Proof. simpl. reflexivity. Qed.

Example tm_test_3 : tree_map (beq_nat 1) (node (leaf 1) 1 (leaf 2)) =
                    (node (leaf true) true (leaf false)).
Proof. simpl. reflexivity. Qed.
```

(c) Consider the partial proof of the following (true!) theorem, which shows the relationship between `tree_map` and `in_order` in terms of the usual list `map` function:

```
Lemma tree_map_in_order : forall X Y (f:X -> Y) (t:tree X),
                              map f (in_order t) = in_order (tree_map f t).
Proof.
  intros X Y f.
  induction t.
  - simpl. reflexivity.
  -  (* HERE! *)
```

What will the proof state look like at the point marked `(* HERE! *)`? (choose one)

i.
```
X : Type
Y : Type
f : X -> Y
============================
 map f (in_order empty) = in_order (tree_map f empty)
```

ii.
```
X : Type
Y : Type
f : X -> Y
t2 : tree X
x : X
t3 : tree X
IHt1 : map f (in_order t2) = in_order (tree_map f t2)
IHt2 : map f (in_order t3) = in_order (tree_map f t3)
============================
 map f (in_order (node t2 x t3)) = in_order (tree_map f (node t2 x t3))
```

iii.
```
X : Type
Y : Type
f : X -> Y
t2 : tree X
IHt : map f (in_order t2) = in_order (tree_map f t2)
============================
 map f (in_order t2) = in_order (tree_map f t2)
```

iv.
```
X : Type
Y : Type
f : X -> Y
x : X
IHt1 : forall t2, map f (in_order t2) = in_order (tree_map f t2)
IHt2 : forall t3, map f (in_order t3) = in_order (tree_map f t3)
============================
 map f (in_order (node t2 x t3)) = in_order (tree_map f (node t2 x t3))
```

(d) From the proof state marked `(* HERE! *)`, which tactic would be used for the next step of the proof? (choose one)

   i. `intros`
   ii. `simpl`
   iii. `rewrite`
   iv. `induction`

(e) To complete the proof of `tree_map_in_order` requires a helper lemma. Which of the following is sufficient ? (choose one)

   i.
```
Lemma map_cons : forall (A B : Type) (f : A -> B) (x:A) (l : list A),
        map f (x :: l) = (f x) :: map f l.
```

   ii.
```
Lemma map_app : forall (A B : Type) (f : A -> B) (l l' : list A),
        map f (l ++ l') = map f l ++ map f l'.
```

   iii.
```
Lemma in_order_map_id : forall (X : Type) (t : tree X),
        in_order t = map (fun x => x) (in_order t).
```

   iv.
```
Lemma in_order_app : forall (X : Type) (t1 t2 : tree X),
        (in_order t1) ++ (in_order t2) = in_order (t1 ++ t2).
```

7. (17 points) Consider the following inductive definition:

```
Inductive inserted {X : Type} : X -> list X -> list X -> Prop :=
| ins_first : forall x l, inserted x l (x::l)
| ins_later : forall x y l1 l2, inserted x l1 l2 -> inserted x (y::l1) (y::l2).
```

The idea is that `inserted x l1 l2` holds exactly when `l2` is just the list `l1` with the element `x` inserted somewhere inside it.

(a) Choose the proof strategy that best fits the lemma proposed below, or select "not provable" if you think the lemma is false. (The definition of `In` is given in the appendix.)

```
Lemma In_inserted : forall (X : Type) (x : X) l,
                      In x l -> exists l', inserted x l' l.
```

i. Induction on the list `l`.
ii. Induction on the hypothesis `In x l`.
iii. Induction on the hypothesis `inserted x l' l`.
iv. *not provable*

(b) Choose the proof strategy that best fits the lemma proposed below, or select "not provable" if you think the lemma is false. (The definition of `In` is given in the appendix.)

```
Lemma inserted_In : forall (X : Type) (x : X) l1 l2,
                      inserted x l1 l2 -> In x l2.
```

i. Induction on the list `l1`.
ii. Induction on the list `l2`.
iii. Induction on the hypothesis `inserted x l1 l2`.
iv. *not provable*

(c) A list `l1` is a *permutation* of another list `l2` if `l1` and `l2` have exactly the same elements (with each element occurring exactly the same number of times), possibly in different orders. For example, the following lists (among others) are permutations of the list `[1;1;2;3]`:

```
[1;1;2;3]
[2;1;3;1]
[3;2;1;1]
[1;3;2;1]
```

On the other hand, `[1;2;3]` is not a permutation of `[1;1;2;3]`, since `1` does not occur twice. Complete the following inductively defined relation in such a way that `permutation l1 l2` is provable exactly when `l1` is a permutation of `l2`. Your definition should make use of the `inserted` proposition defined earlier.

```
Inductive permutation {X:Type}  : list X -> list X -> Prop :=
```

(d) The following Coq function counts the number of occurrences of a given natural number `n` within a list.

```
Fixpoint count (n:nat) (l:list nat) : nat :=
  match l with
    | [] => 0
    | x::tl => if beq_nat x n then 1 + (count n tl) else (count n tl)
  end.
```

Using `count`, formulate a lemma that characterizes the correctness of your definition of `permutation`. You *do not* have to prove the lemma, just state it.

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 or (P Q : Prop) : Prop :=
  | or_introl : P -> or P Q
  | or_intror : Q -> or P Q.

Notation "P \/ Q" := (or 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.
```

```
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).

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


Fixpoint ble_nat (n m : nat) : bool :=
  match n with
  | O => true
  | S n' =>
      match m with
      | O => 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 In {A : Type} (x : A) (l : list A) : Prop :=
  match l with
  | [] => False
  | x' :: l' => x' = x \/ In x l'
  end.

Fixpoint length (X:Type) (l:list X) : nat :=
  match l with
    | []     => 0
    | h :: t => S (length X t)
  end.
```

```
Fixpoint index {X : Type} (n : nat)
         (l : list X) : option X :=
  match l with
    | [] => None
    | h :: t => if beq_nat n O then Some h else index (pred n) t
  end.
```

```
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 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.
```