CIS 500 — Software Foundations
Midterm I

February 18, 2009

Name: _____

Email: _____

	Score
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
Total	

Instructions

- This is a closed-book exam: you may not use any books or notes.
- You have 80 minutes to answer all of the questions.
- The exam is worth 80 points. However, questions vary significantly in difficulty, and the point value of a given question is not always exactly proportional to its difficulty. Do not spend too much time on any one question.
- Partial credit will be given. All correct answers are short. The back side of each page may be used as a scratch pad.
- Good luck!

1. (5 points) Consider the following Coq function:

```
Fixpoint concatMap (X Y : Set) (f : X → list Y) (l : list X)
                   {struct l} : list Y :=
  match l with
  | nil    => nil
  | h :: t => (f h) ++ (concatMap _ _ f t)
  end.
```

(a) What is the type of `concatMap`? (I.e., what does `Check concatMap` print?)

(b) What does

```
Eval simpl in (concatMap _ _ (fun x => x) [[1,2],[3,4]]).
```

print?

(c) What does

```
Eval simpl in (concatMap _ _ (fun x => [x+1,x+2]) ([1,2])).
```

print?

2. (5 points)

   (a) Fill in the definition of the Coq function `elem` below.

   Given a type `X`, an equality-testing function `eq` for `X`, an element `e` of type `X`, and a list `l` of type `list X`, the expression `elem X eq e l` returns `true` if and only an element `eq`-equal to `e` appears in the list. For example, `elem nat beq_nat 2 [1,2,3]` yields `true` (because `beq_nat 2 2 = true`) while `elem nat beq_nat 5 [1,2,3]` yields `false`.

```
Fixpoint elem (X : Set) (eq : X → X → bool) (e : X) (l : list X)
              {struct l} : bool :=
```

   (b) Why do we need to pass an equality-testing function `eq` as an argument to `elem` instead of just using = to test for equality?

3. (6 points) Fill in the definition of the Coq function `nub` below.

   Given a type `X`, an equality function `eq` for `X`, and a list `l` of type `list X`, the expression `nub X eq l` yields a list that retains only the last copy of each element in the input list. For example, `nub nat beq_nat [1,2,1,3,2,2,4]` yields `[1,3,2,4]`.

```
Fixpoint nub (X : Set) (eq : X → X → bool) (l : list X)
             {struct l} : list X :=
```

4. (5 points)

   (a) Briefly explain the use and behavior of the `apply` tactic.

   (b) Briefly explain the use and behavior of the `apply ... in ...` tactic.

5. (6 points) Recall the Coq function `repeat`:

```
Fixpoint repeat (X : Set) (n : X) (count : nat) {struct count} : list X :=
  match count with
  | O => nil
  | S count' => cons n (repeat _ n count')
  end.
```

Consider the following partial proof:

```
Lemma repeat_injective : forall (X : Set) (x : X) (n m : nat),
  repeat _ x n = repeat _ x m →
  n = m.
Proof.
  intros X x n m eq. induction n as [|n'].
  Case "n = 0". destruct m as [|m'].
    SCase "m = 0". reflexivity.
    SCase "m = S m'". inversion eq.
  Case "n = S n'". destruct m as [|m'].
    SCase "m = 0". inversion eq.
    SCase "m = S m'".
      assert (n' = m') as H.
        SSCase "Proof of assertion".
```

Here is what the "goals" display looks like after Coq has processed this much of the proof:

```
2 subgoals

  SSCase := "Proof of assertion" : String.string
  SCase := "m = S m'" : String.string
  Case := "n = S n'" : String.string
  X : Set
  x : X
  n' : nat
  m' : nat
  eq : repeat X x (S n') = repeat X x (S m')
  IHn' : repeat X x n' = repeat X x (S m') → n' = S m'
  ============================
   n' = m'

subgoal 2 is:
 S n' = S m'
```

This proof attempt is not going to succeed. Briefly explain why and say how it can be fixed. (Do not write the repaired proof in detail—just say briefly what needs to be changed to make it work.)

6. (5 points) Suppose we make the following inductive definition:

```
Inductive foo (X : Set) (Y : Set) : Set :=
  | foo1 : X → foo X Y
  | foo2 : Y → foo X Y
  | foo3 : foo X Y → foo X Y.
```

Fill in the blanks to complete the induction principle that will be generated by Coq.

```
foo_ind
     : forall (X Y : Set) (P : foo X Y → Prop),

       (forall x : X, ____________________________________) →

       (forall y : Y, ____________________________________) →

       (_____________________________________________________) →

        ____________________________________________________
```

7. (6 points)

Consider the following induction principle:

```
bar_ind
     : forall P : bar → Prop,
       (forall n : nat, P (bar1 n)) →
       (forall b : bar, P b → P (bar2 b)) →
       (forall (b : bool) (b0 : bar), P b0 → P (bar3 b b0)) →
       forall b : bar, P b
```

Write out the corresponding inductive set definition.

```
Inductive bar : Set :=

  | bar1 : ____________________________________________

  | bar2 : ____________________________________________

  | bar3 : ____________________________________________.
```

8. (6 points) Suppose we give Coq the following definition:

```
Inductive R : nat → list nat → Prop :=
  | c1 : R 0 []
  | c2 : forall n l, R n l → R (S n) (n :: l)
  | c3 : forall n l, R (S n) l → R n l.
```

Which of the following propositions are provable? (Write *yes* or *no* next to each one.)

(a) `R 2 [1,0]`

(b) `R 1 [1,2,1,0]`

(c) `R 6 [3,2,1,0]`

9. (6 points) The following inductively defined proposition...

```
Inductive appears_in (X:Set) (a:X) : list X → Prop :=
  | ai_here : forall l, appears_in X a (a::l)
  | ai_later : forall b l, appears_in X a l → appears_in X a (b::l).
```

...gives us a precise way of saying that a value `a` appears at least once as a member of a list `l`.

Use `appears_in` to complete the following definition of the proposition `no_repeats X l`, which should be provable exactly when `l` is a list (with elements of type `X`) where every member is different from every other. For example, `no_repeats nat [1,2,3,4]` and `no_repeats bool []` should be provable, while `no_repeats nat [1,2,1]` and `no_repeats bool [true,true]` should not be.

```
Inductive no_repeats (X:Set) : list X → Prop :=
```

10. (2 points) Complete the definition of **and**, as it is defined in `Logic.v`:

```
Inductive and (A B : Prop) : Prop :=
```

11. (2 points) Complete the definition of **or**, as it is defined in `Logic.v`:

```
Inductive or (A B : Prop) : Prop :=
```

12. (6 points) Write an informal proof (in English) of the proposition $\forall$ `P : Prop`, `~(P` $\wedge$ `~P)`.

13. (4 points) Recall the `nat`-indexed proposition `ev` from `Logic.v`:

```
Inductive ev : nat → Prop :=
  | ev_0 : ev 0
  | ev_SS : forall n:nat, ev n → ev (S (S n)).
```

Complete the definition of the following proof object:

```
Definition ev_plus2 : forall n, ev n → ev (plus 2 n) :=
```

14. (6 points) Recall the definition of `ex` (existential quantification) from `Logic.v`:

```
Inductive ex (X : Set) (P : X → Prop) : Prop :=
  ex_intro : forall witness:X, P witness → ex X P.
```

(a) In English, what does the proposition

```
ex nat (fun n => ev (S n))
```

mean?

(b) Complete the definition of the following proof object:

```
Definition p : ex nat (fun n => ev (S n)) :=
```

15. (10 points) Recall the definition of the `index` function:

```
Fixpoint index (X : Set) (n : nat) (l : list X) {struct l} : option X :=
  match l with
  | [] => None
  | a :: l' => if beq_nat n O then Some a else index _ (pred n) l'
  end.
```

Write an informal proof of the following theorem:

$$\forall\ \texttt{X n l},\ \texttt{length l = n} \rightarrow \texttt{index X (S n) l = None}.$$