CIS 500 — Software Foundations
Midterm I

October 10, 2007

Name: ______________________________

Email: ______________________________

	Score
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
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.
- 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!

Functional Programming

1. (2 points) Consider the following `Fixpoint` definition in Coq.

```
Fixpoint zip (X Y : Set) (lx : list X) (ly : list Y) {struct lx}
           : list (X*Y) :=
  match lx with
  | nil =>
      nil _
  | x::tx =>
      match ly with
      | nil => nil _
      | y::ty => (x,y) :: (zip _ _ tx ty)
      end
  end.
```

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

2. (2 points) What does

```
Eval simpl in (zip _ _ [one,two] [no,no,yes,yes].
```

   print?

3. (6 points) Intuitively, the `zip` function transforms a pair of lists into a list of pairs. The inverse transformation, `unzip`, takes a list of pairs and returns a pair of lists. For example, evaluating

```
Eval simpl in (unzip _ _ [(one,no),(two,no)]).
```

prints:

```
  = ([one, two], [no, no])
  : list nat * list yesno
```

Fill in the blanks in the following definition of `unzip`. Write out all type parameters explicitly — do not use the "wildcard type" `_` anywhere (i.e., please be *completely explicit* about type parameters here, rather than leaving them implicit as we did in the definition of `zip` above).

```
Fixpoint unzip (X Y : Set) (l : list (X*Y)) {struct l}

          : _____________________________ :=

  match l with

  | nil =>

      _____________________________________________________

  | cons (x,y) t =>

      match unzip X Y t with

        ______________________________________________________

      end

  end.
```

4. (3 points) Recall that the `filter` function takes a function `test` of type `X->yesno` and a list `l` with elements of `X` and returns a list with elements of type `X` containing just the elements of `l` for which `test` yields `yes`.

   Which of the following `Fixpoint` definitions correctly implements the `filter` function? Circle the correct answer.

   (a)
```
Fixpoint filter (X:Set) (test: X->yesno) (l:list X) {struct l} : (list X) :=
  match l with
  | nil => nil X
  | h :: t => match (test h) with
              | no => h :: (filter X test t)
              | yes => filter X test t
              end
  end.
```

   (b)
```
Fixpoint filter (X:Set) (test: X->yesno) (l:list X) {struct l} : (list X) :=
  match l with
  | nil => nil X
  | h :: t => match (test l) with
              | no => filter X test t
              | yes => h :: (filter X test t)
              end
  end.
```

   (c)
```
Fixpoint filter (X:Set) (test: X->yesno) (l:list X) {struct l} : (list X) :=
  match l with
  | nil => nil X
  | h :: t => h :: (filter X test t)
  end.
```

   (d)
```
Fixpoint filter (X:Set) (test: X->yesno) (l:list X) {struct l} : (list X) :=
  match l with
  | nil => nil X
  | h :: t => match (test h) with
              | no => filter X test t
              | yes => h :: (filter X test t)
              end
  end.
```

   (e)
```
Fixpoint filter (X:Set) (test: X->yesno) (l:list X) {struct l} : (list X) :=
  match l with
  | nil => nil X
  | h :: t => (test h) :: (filter X test t)
  end.
```

   (f) None of the above.

Coq Basics

5. (4 points) Briefly explain what the `assert` tactic does.

6. (5 points) Recall the definition of the inductive predicate `evenI`:

```
Inductive evenI : nat -> Prop :=
  | even_zero : evenI O
  | even_SS : forall n:nat, evenI n -> evenI (S (S n)).
```

The following proof attempt is not going to succeed. Briefly explain why.

```
Lemma l : forall n,
  evenI n.
Proof.
  intros n. induction n.
    Case "O". simpl. apply even_zero.
    Case "S".
      ...
```

7. (6 points) Recall the definition of `plus`:

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

(a) What will Coq print in response to this query?

```
Eval simpl in (forall n, plus n 0 = n).
```

(b) What will Coq print in response to this query?

```
Eval simpl in (forall n, plus 0 n = n).
```

(c) Briefly (1 or two sentences) explain the difference.

Inductively Defined Sets

8. (8 points) Consider the following inductive definition:

```
Inductive foo (X:Set) : Set :=
  | c1 : list X -> foo X -> foo X
  | c2 : foo X.
```

What induction principle will Coq generate for `foo`? (Fill in the blanks.)

```
foo_ind :

  forall (X : Set) (P : foo X -> Prop),

       (forall (l : list X) (f : foo X),

              ______________________ -> _______________________ )

    -> _________________________________________________

    -> forall f : foo X, _______________________________
```

9. (6 points) Here is an induction principle for an inductively defined set `s`.

```
myset_ind :
  forall P : myset -> Prop,
      (forall y : yesno, P (con1 y))
   -> (forall (n : nat) (m : myset), P m -> P (con2 n m))
   -> forall m : myset, P m
```

What is the definition of `myset`?

Inductively Defined Propositions

10. (6 points) Recall the definition of the set `tree` from the review exercises:

```
Inductive tree : Set :=
  | leaf : tree
  | node : tree -> tree -> tree.
```

Now consider the following inductively defined proposition:

```
Inductive p : tree -> nat -> Prop :=
  | c1 : p leaf one
  | c2 : forall t1 t2 n1 n2,
           p t1 n1 -> p t2 n2 -> p (node t1 t2) (plus n1 n2)
  | c3 : forall t n, p t n -> p t (S n).
```

Describe, in English, the conditions under which the proposition `p t n` is provable.

11. (4 points) Suppose we give Coq the following definition:

```
Inductive R : nat -> nat -> nat -> Prop :=
  | c1 : R zero zero zero
  | c2 : forall m n o, R m n o -> R (S m) n (S o)
  | c3 : forall m n o, R m n o -> R m (S n) (S o)
  | c4 : forall m n o, R (S m) (S n) (S (S o)) -> R m n o
  | c5 : forall m n o, R m n o -> R n m o.
```

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

(a) `R one one two`

(b) `R two two six`

12. (2 points) If we dropped constructor `c5` from the definition of `R`, would the set of provable propositions change? Write *yes* or *no* and briefly (1 sentence) explain your answer. Write *yes* or *no*.

13. (2 points) If we dropped constructor `c4` from the definition of `R`, would the set of provable propositions change? Write *yes* or *no*.

“Programming with Propositions”

14. (5 points) Complete the following definition of existential quantification as an inductive proposition in Coq.

```
Inductive ex (X : Type) (P : X -> Prop) : Prop :=
```

Operational Semantics

15. (6 points) In the lectures, we have been working with a very simple programming language whose terms consist of just `plus` and constants. Here is an equally simple language whose terms are just the boolean constants `true` and `false` and a conditional expression:

```
Inductive tm : Set :=
  | tm_true : tm
  | tm_false : tm
  | tm_if : tm -> tm -> tm -> tm.

Inductive value : tm -> Prop :=
  | v_true : value tm_true
  | v_false : value tm_false.

Inductive eval : tm -> tm -> Prop :=
  | E_IfTrue : forall t1 t2,
        eval (tm_if tm_true t1 t2)
             t1
  | E_IfFalse : forall t1 t2,
        eval (tm_if tm_false t1 t2)
             t2
  | E_If : forall t1 t1' t2 t3,
        eval t1 t1'
     -> eval (tm_if t1 t2 t3)
             (tm_if t1' t2 t3).
```

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

(a)
```
eval tm_false tm_false
```

(b)
```
eval
  (tm_if
    tm_true
    (tm_if tm_true tm_true tm_true)
    (tm_if tm_false tm_false tm_false))
  tm_true
```

(c)
```
eval
  (tm_if
    (tm_if tm_true tm_true tm_true)
    (tm_if tm_true tm_true tm_true)
    tm_false)
  (tm_if
    tm_true
    (tm_if tm_true tm_true tm_true)
    tm_false)
```

16. (4 points) Suppose we want to add a "short circuit" to the evaluation relation so that it can recognize when the *then* and *else* branches of a conditional are the same value (either `tm_true` or `tm_false`) and reduce the whole conditional to this value in a single step, even if the guard has not yet been reduced to a value. For example, we would like this proposition to be provable:

```
eval
  (tm_if
    (tm_if tm_true tm_true tm_true)
    tm_false
    tm_false)
  tm_false
```

Write an extra clause for the `eval` relation that achieves this effect. (Fill in the blanks.)

```
| E_ShortCircuit : forall ______________________________________,

        ________________________________________________________
```

17. (9 points) It can be shown that the determinism and progress theorems for the `eval` relation in the lecture notes...

```
Theorem eval_deterministic :
  partial_function _ eval.

Theorem eval_progress : forall t,
  value t \/ (exists t', eval t t').
```

...also hold for the definition of `eval` given in question 15.

After we add the clause `E_ShortCircuit` from question 16...

(a) Does `eval_deterministic` still hold? Write *yes* or *no* and briefly (1 sentence) explain your answer.

(b) Does `eval_progress` still hold? Write *yes* or *no* and briefly (1 sentence) explain your answer.

(c) In general, is there any way we could cause `eval_progress` to fail if we *took away* one or more constructors from the original `eval` relation given in question 15? Write *yes* or *no* and briefly (1 sentence) explain your answer.