CIS 500: Software Foundations **Midterm I**

October 4, 2016

Name (printed): ____

Username (PennKey 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 waste time answering questions intended for the other track.*

Mark the box of the track you are following. (If you are following one track but want to move to the other now, please mark the box for the track you *want* to be on and write a note on this page telling us that we should switch you to this track.)

☐ Standard ☐ Advanced

1. (9 points) Circle *True* or *False* for each statement.

(a) 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*

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

True *False*

(c) The proposition `False` cannot be proved in Coq, no matter what axioms we add.

True *False*

(d) if `H : S (S x) = S y` is a current assumption, then `inversion H` will solve any goal.

True *False*

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

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

True *False*

(f) `[] =~ Star re` is provable for every `re`. (The definition of `=~` is given in the appendix.)

True *False*

(g) If we assume `s =~ EmptySet` in Coq, then we can prove `s =~ EmptyStr`.

True *False*

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

True *False*

(i) For every property of numbers `P : nat -> Prop`, we can construct a boolean function `testP : nat -> bool` such that `testP` reflects `P`.

True *False*

2. (10 points) Write the type of each of the following Coq expressions, or write "ill-typed" if it does not have one.

(a) `3 = 4`

(b) `beq_nat 3 4`

(c) `forall (x:nat), beq_nat x x`

(d) `fun (n : nat) => ev n`

(e) `fun n => forall m, leb m n = true`

(f) `if beq_nat 0 1 then (fun n => plus n 5) else plus 6`

(g) `fun (X:Type) (x:X) => x :: nil`

(h) `(fun n => plus n) 3`

(i) `fun P => (P \/ False)`

(j) `ev_SS`

3. [**Standard 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) `forall (X Y : Type), option X -> option Y`

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

(c) `bool -> Prop`

(d) `forall (X : Type), (X -> X) -> X`

(e) `3 <= 2`

(f) `1 <= 2`

(g) `[2] =~ (Char 2)`

(h) `[20, 10] =~ (App (Char 20) (Char 10))`

4. (11 points) For each of the following propositions, write "**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]`

(b) `forall s, In 3 ([1;2;3] ++ s)`

(c) `forall s, In 3 (s ++ [1;2;3])`

(d) `exists s, In 3 (s ++ [1;2;3])`

(e) `exists (x y : list nat), x ++ y = y ++ x`

(f) `forall n, n+5 <= n+6`

(g) `forall f g, (forall x, f x = g x) -> f = g`

(h) `forall x y, x * y = y * x`

(i) `forall P : Prop, P \/ ~P`

(j) `forall P : Prop, P -> ~~P`

(k) `forall P : Prop, P`

5. This problem asks you to translate mathematical ideas from English into Coq.

   (a) (3 points) In class, we saw how to define the relation `In` using a `Fixpoint` (the definition is repeated in the appendix).

   Give an alternative definition of `In` as an `Inductive` relation `IndIn`, such that `IndIn x l` holds exactly when `In x l` holds. Do not use `In` in your solution.

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

   (b) (4 points) Define an inductive relation that holds exactly when every element of a list appears at most once—that is, there are no duplicate elements in the list. This time, you *may* use the definition of `In` in your solution.

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

(c) (5 points) Consider the following inductively defined relation `Doubles`, which holds exactly when each element in the list is immediately repeated (e.g., `Doubles` holds for the lists `[]`, `[1;1]`, `[1;1;2;2;1;1]`, `[1;1;1;1]`, but not for the lists `[1]` or `[1;1;1]`).

```
Inductive Doubles {X:Type} : list X -> Prop :=
| DoublesNil:  Doubles []
| DoublesCons: forall x l, Doubles l -> Doubles (x :: x :: l).
```

Give an alternative definition of `Doubles` as a `Fixpoint DoublesP`, such that `DoublesP l` holds exactly when `Doubles l` holds. Do not use `Doubles` in your solution.

```
Fixpoint DoublesP {X:Type} (l: list X) : Prop :=
```

(d) (8 points) Give an inductively defined property that specifies whether a regular expression `re` is *nullable*—that is, when it can match the empty string. For example, the regular expressions

```
EmptyStr
Star (Char 10)
Union (Union (Char 20) EmptyStr) (Char 10)
```

are all nullable, while

```
Char 10
App (Char 10) (Star (Char 20))
Union (Char 10) (Char 20)
```

are not nullable.

```
Inductive Nullable {X:Type} : reg_exp X -> Prop :=
```

6. An alternate way to encode lists in Coq is the `dlist` ("doubly-ended list") type, which has a third constructor corresponding to a "`cons` at the end" (`snoc`) operation on regular lists, as shown below:

```
Inductive dlist (X:Type) : Type :=
| d_nil : dlist X
| d_cons : X -> dlist X -> dlist X
| d_snoc : dlist X -> X -> dlist X.

(* Make the type parameter implicit. *)
Arguments d_nil {X}.
Arguments d_cons {X} _ _.
Arguments d_snoc {X} _ _.
```

We can convert any `dlist` to a regular list using the following function (the definition of `snoc` is given in the references).

```
Fixpoint to_list {X} (dl: dlist X) : list X :=
match dl with
| d_nil => []
| d_cons x l => x::(to_list l)
| d_snoc l x => snoc (to_list l) x
end.
```

(a) (2 points) As we saw in the homework with the alternate "binary" encoding of natural numbers, there may be multiple `dlist`s that represent the same list. Demonstrate this by giving definitions of `example1` and `example2` such that the lemma `distinct_dlists_but_same_list` below is provable (there is no need to prove it).

```
Definition example1 : dlist nat :=
```

```
Definition example2 : dlist nat :=
```

```
Lemma distinct_dlists_but_same_list :
  example1 <> example2 /\ (to_list example1) = (to_list example2).
```

(b) (6 points) We can define list operations directly on the `dlist` representation. Complete the following function for appending two `dlists`. (Your function should work by recursion on `l1`.)

```
Fixpoint dapp {X} (l1 l2: dlist X) : dlist X :=
```

(c) (6 points) The `dapp` function from part (b) should satisfy the following correctness lemma stating that it agrees with the list append operation `++` (whose definition is given in the appendix).

```
Lemma dapp_correct : forall (X:Type) (l1 l2:dlist X),
  to_list (dapp l1 l2) = (to_list l1) ++ (to_list l2).
Proof.
  intros X l1.
  induction l1 as [| x l| l x].
  Case "d_nil".   ...
  Case "d_cons".  ...
  Case "d_snoc".  ...
Qed.
```

- What induction hypothesis is available in the `d_cons` case of the proof? (Circle one.)
  - i. `to_list (dapp (d_cons x l) l2) = (to_list (d_cons x l)) ++ (to_list l2)`
  - ii. `to_list (dapp l l2) = (to_list l) ++ (to_list l2)`
  - iii. `forall l2 : dlist X, to_list (dapp l l2) = to_list l ++ to_list l2`
  - iv. `forall l2 : dlist X,`
    `to_list (dapp (d_cons x l) l2) = to_list (d_cons x l) ++ to_list l2`
- What induction hypothesis is available in the `d_snoc` case of the proof? (Circle one.)
  - i. `to_list (dapp (d_snoc l x) l2) = (to_list (d_snoc l x)) ++ (to_list l2)`
  - ii. `to_list (dapp l l2) = (to_list l) ++ (to_list l2)`
  - iii. `forall l2 : dlist X, to_list (dapp l l2) = to_list l ++ to_list l2`
  - iv. `forall l2 : dlist X,`
    `to_list (dapp (d_snoc l x) l2) = to_list (d_snoc l x) ++ to_list l2`

7. [**Advanced Only**] (16 points) Fill in the indicated cases of a careful *informal* proof of the following theorem from *Software Foundations*:

> **Theorem:** For all strings `s`, regular expressions `re`, and characters `x`, if `s =~ re` and `In x s`, then `In x (re_chars re)`.

Your proof may use the following lemma:

> **Lemma** `in_app_iff`: `In a (l++l')` if and only if `In a l` or `In a l'`, for all lists `l` and `l'` and elements `a`.

Proof: Proceed by induction on the evidence that `s =~ re`.

Cases `MEmpty`, `MApp`, `MUnionL`, `MUnionR`: *(Omitted. Do not worry about these cases.)*

Case `MChar`: If the rule used to prove `s =~ re` is `MChar`, we know that `re` is `Char x'` and `s` is the singleton list `[x']`. *(Fill in the rest of this case and the two cases below...)*

Case `MStar0`:

Case `MStarApp`:

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

Inductive ev : nat -> Prop :=
| ev_0 : ev O
| ev_SS : forall n : nat, ev n -> ev (S (S n)).
```

Options

```
Inductive option (X:Type) : Type :=
  | Some : X -> option X
  | None : option X.

Arguments Some {X} _.
Arguments None {X}.
```

Lists

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

Definition snoc (X:Type) (l:list X) (x:X) := app l (cons x nil).

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.
```

Regular Expressions

```
Inductive reg_exp (T : Type) : Type :=
| EmptySet : reg_exp T
| EmptyStr : reg_exp T
| Char : T -> reg_exp T
| App : reg_exp T -> reg_exp T -> reg_exp T
| Union : reg_exp T -> reg_exp T -> reg_exp T
| Star : reg_exp T -> reg_exp T.

Inductive exp_match {X: Type} : list X -> reg_exp X -> 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 ->
              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).

Fixpoint re_chars {T} (re : reg_exp T) : list T :=
  match re with
  | EmptySet => []
  | EmptyStr => []
  | Char x => [x]
  | App re1 re2 => re_chars re1 ++ re_chars re2
  | Union re1 re2 => re_chars re1 ++ re_chars re2
  | Star re => re_chars re
  end.
```