CMSC 631: Midterm Exam (Spring 2019)

1 Question 1 (15 points)

For this question, you will be asked to prove that $A \leftrightarrow B$ implies $B \leftrightarrow A$ in three different ways.
(a) Give a mathematical proof of this statement.

Our assumption $A \leftrightarrow B$ is shorthand for $A \rightarrow B$ (1) and $B \rightarrow A$ (2).
Let's first show the forward direction: $B \rightarrow A$.
  This follows from (2).
Now let's show the other direction: $A \rightarrow B$.
  This follows from (1).
Hence $B \leftrightarrow A$.

(b) Prove the statement above in Coq.

```
Lemma iff_sym : ∀ A B, (A ↔ B) → (B ↔ A).
Proof.
(* Solution *)
  intros A B H.
  destruct H as [Hab Hba].
  split; assumption.
Qed.
```

(c) Now prove it as a definition.

```
Definition iff_sym_def {A B} (H : A ↔ B) : B ↔ A :=
(* Solution *)
  match H with
  | conj Hab Hba ⇒ conj Hba Hab
  end.
```

(Bonus) Do a one line (one period) proof of (b).

```
Lemma iff_sym_one_line : ∀ A B, A ↔ B → B ↔ A.
Proof.
(* Solution *)
  intros A B []; split; assumption.
(* or *)
  intros; symmetry; assumption.
Qed.
```

2 Question 2 (15 points)

Consider the following two possible definitions of `In`, the first of which we used in class.

```
Fixpoint In {A} (a : A) (l : list A) : Prop :=
  match l with
  | []      ⇒ False
  | x :: l' ⇒ (a = x) ∨ In a l'
  end.

Inductive In' {A} : A → list A → Prop :=
  | Here : ∀ a l, In' a (a :: l)
  | Later : ∀ a x l, In' a l → In' a (x :: l).
```

Here's a proof (as a fixpoint) that `In' a l` implies `In a l`:

```
Fixpoint In'_then_In {A} (a : A) (l : list A) (P : In' a l) : In a l :=
  match P with
  | Here a l        ⇒ or_introl (eq_refl)
  | Later a x l' P' ⇒ or_intror (In'_then_In a l' P')
  end.
```

(a) Based on the proof above, fill in the next line of the equivalent Coq proof.

```
Lemma In'_then_In_start : ∀ A (a : A) (l : list A), In' a l → In a l.
Proof.
  intros A a l P.
  (* Solution *)
  induction P.
```

(b) Sketch the rest of the proof (or, if you prefer, do it in Coq).

For the "Here" case, we need to show that In a (a :: l).
This is trivially true (it's the left hand side of the disjunction).
For the "Later" case, we need to show that In a (x :: l) for some x.
However, we have an inductive hypothesis that a is in l, which is the right hand of our disjunction.

In Coq:

```
 - left. reflexivity.
 - right. assumption.
Qed.
```

(c) Is the opposite direction ([In a l] implies [In' a l]) true? Why or why not?

Yes. Both are true precisely when a is a member of the list.
Moreover, when you unwrap the definition of ∨ , they are almost identical: "Here" is equivalent to the left side of the disjunction and "Later" is equivalent to the right case.

3 Question 3 (20 points)

Write the type of each of the following Coq expressions (write "ill typed" if an expression does not have a type).

(Answers are on the right of the colon.)

(a) `@nil bool` : `list bool`

(b) `filter (fun x ⇒ eqb_string x "foo")` : `list string → list string`

(c) `2 + 2 = 5` : `Prop`

(d) `if eqb 3 4 then false else 0` is ill typed

(e) `fun (m : nat) ⇒ m :: m :: m` is ill typed

(f) `∀ (m n : nat), m ≤ n ∨ n ≤ m` : `Prop`

(g) `or False` : `Prop → Prop`

(h) `∀ (n : nat), 2 * n` is ill typed

(i) `CAss` : `string → aexp → com`

(j) `fun (n : nat) ⇒ le_n (S n)` : `∀ (n:nat), S n ≤ S n`

4 Question 4 (20 points)

For each of the types below, write a Coq expression that has that type or write "empty" if there are no such expressions.

(Answers are on the left of the colon, where applicable. The underscores for type variables were not required for full credit.)

(a) `t_empty false : total_map bool`

(b) `ANum 4 : aexp`

(c) $4 \leq 3$ is empty

(d) `or_intror eq_refl : 5 = 6 ∨ 6 = 6`

(e) `∀ b, b = true` is empty

(f) `(True, False) : Prop * Prop`

(g) `fun _ a ⇒ 5 : ∀ (A : Type), A → nat`

(h) `∀ (A : Type), nat → A` is empty

(i) `fun _ a ⇒ a : ∀ (A : Type), A → A`

(j) `@map bool : ∀ (A : Type), (bool → A) → list bool → list A`

For the last item, `fun _ _ _ ⇒ nil` is also correct but pretty uninspired.

5 Question 5 (10 points)

We often try to prove a lemma by using [destruct] on some hypothesis, only to find ourselves stuck at some stage of the proof. Give two "failure modes" for destruct, and the tactics that handle that failure mode.

(a) Failure mode and tactic #1:

Often we have to do case analysis on an inductively defined datatype with infinitely many constructors, like nat. If we just try repeatedly calling destruct, we'll find ourselves in an infinite regress.

Instead we can call `induction`, which will provide us with an inductive hypothesis for dealing with the inductive case (the S case for nat)

(b) Failure mode and tactic #2:

Destruct often throws away useful information. For instance if we call `destruct H` on `H` : `le x` 0, Coq will forget that le's second argument was zero, which allows us to discharge the `le_S` case.

For this we can use `inversion`, which keeps around information about the arguments to the hypothesis we are destructing on, and discharges cases that don't match those arguments.

6 Question 6 (10 points)

(a) Is the following lemma true? Why or why not?

```
Lemma if_test : ∀ b b' c1 c2,
    cequiv c1 c2 →
    cequiv (IFB b THEN c1 ELSE c2 FI) (IFB b' THEN c1 ELSE c2 FI).
```

True.
Since both branches are the same, both of these commands are equivalent to just `c1`.

(b) How about this lemma? Again, why or why not?

```
Lemma while_test : ∀ b b' c1 c2,
    cequiv c1 c2 →
    cequiv (WHILE b DO c1 END) (WHILE b' DO c2 END).
```

False.
The two loops may run a different number of times based on the guard. For instance, if `b = BFalse` and `b' = BTrue`, the first loop always terminates and the second one never does.

Library Reference

A Logic

```
Inductive and (X Y : Prop) : Prop :=
  conj : X → Y → and X Y.

Inductive or (X Y : Prop) : Prop :=
 | or_introl : X → or X Y
 | or_intror : Y → or X Y.

Arguments conj {X Y}.
Arguments or_introl {X Y}.
Arguments or_intror {X Y}.

Notation "A ∧ B" := (and A B).
Notation "A ∨  B" := (or A B).

Definition iff (A B : Prop) := (A → B) ∧ (B → A).
Notation "A ↔  B" := (iff A B) (at level 95).
```

B Booleans

```
Inductive bool : Type :=
  | true
  | false.

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

Definition andb (b1 b2: bool) : bool :=
  match b1 with
  | true ⇒ b2
  | false ⇒ false
  end.

Definition orb (b1 b2:bool) : bool :=
  match b1 with
  | true ⇒ true
  | false ⇒ b2
  end.
```

C Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).

Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

```
Fixpoint minus (n m:nat) : nat :=
  match n, m with
  | O   , _    ⇒ O
  | S _ , O    ⇒ n
  | S n', S m' ⇒ minus n' m'
  end.

Fixpoint mult (n m : nat) : nat :=
  match n with
    | O ⇒ O
    | S n' ⇒ plus m (mult n' m)
  end.

Notation "x + y" := (plus x y) (at level 50, left associativity).
Notation "x - y" := (minus x y) (at level 50, left associativity).
Notation "x * y" := (mult x y) (at level 40, left associativity).

Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O ⇒ match m with
         | O ⇒ true
         | S m' ⇒ false
         end
  | S n' ⇒ match m with
            | O ⇒ false
            | S m' ⇒ eqb n' m'
            end
  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.

Notation "x =? y" := (eqb x y) (at level 70).
Notation "x ≤? y" := (leb x y) (at level 70).

Inductive le : nat → nat → Prop :=
  | le_n n : le n n
  | le_S n m : le n m → le n (S m).

Notation "m ≤ n" := (le m n).
```

D Lists

```
Inductive list (X:Type) : Type :=
  | nil
  | cons (x : X) (l : list X).

Arguments nil {X}.
Arguments cons {X} _ _.

Notation "x :: y" := (cons x y) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y []) ..).
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.

Fixpoint fold {X Y} (f : X → Y → Y) (l : list X) (b : Y) : Y :=
  match l with
  | nil ⇒ b
  | h :: t ⇒ f h (fold f t b)
  end.
```

E Strings

We won't define strings from scratch here. Assume `eqb_string` has the type given below, and anything within quotes is a string.

```
Parameter eqb_string : string → string → bool.
```

F Maps

```
Definition total_map (A:Type) := string → A.

Definition t_empty {A:Type} (v : A) : total_map A :=
  (fun _ ⇒ v).

Definition t_update {A:Type} (m : total_map A) (x : string) (v : A) :=
  fun x' ⇒ if eqb_string x x' then v else m x'.

Notation "{ --> d }" := (t_empty d) (at level 0).
Notation "m '&' { a --> x }" := (t_update m a x) (at level 20).
```

G Imp

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | AId (x :  string)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).

Inductive bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BNot (b : bexp)
  | BAnd (b1 b2 : bexp).

Inductive com : Type :=
  | CSkip
  | CAss (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).

Notation "'SKIP'" := CSkip.
Notation "x '::=' a" := (CAss x a) (at level 60).
Notation "c1 ;; c2" := (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" := (CWhile b c) (at level 80, right associativity).
Notation "'IFB' c1 'THEN' c2 'ELSE' c3 'FI'" := (CIf c1 c2 c3) (at level 80, right associativity).

Definition state := total_map nat.

Fixpoint aeval (st : state) (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | AId x ⇒ st x
  | APlus a1 a2 ⇒ (aeval st a1) + (aeval st a2)
  | AMinus a1 a2  ⇒ minus (aeval st a1) (aeval st a2)
  | AMult a1 a2 ⇒ (aeval st a1) * (aeval st a2)
  end.

Fixpoint beval (st : state) (b : bexp) : bool :=
  match b with
  | BTrue       ⇒ true
  | BFalse      ⇒ false
  | BEq a1 a2   ⇒ (aeval st a1) =? (aeval st a2)
  | BLe a1 a2   ⇒ (aeval st a1) ≤? (aeval st a2)
  | BNot b1     ⇒ negb (beval st b1)
  | BAnd b1 b2  ⇒ andb (beval st b1) (beval st b2)
  end.
```

```
Reserved Notation "c1 '/' st '\\' st'"
                  (at level 40, st at level 39).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      SKIP / st \\ st
  | E_Ass  : ∀ st a1 n x,
      aeval st a1 = n →
      (x ::= a1) / st \\ st & { x --→ n }
  | E_Seq : ∀ c1 c2 st st' st'',
      c1 / st  \\ st' →
      c2 / st' \\ st'' →
      (c1 ;; c2) / st \\ st''
  | E_IfTrue : ∀ st st' b c1 c2,
      beval st b = true →
      c1 / st \\ st' →
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_IfFalse : ∀ st st' b c1 c2,
      beval st b = false →
      c2 / st \\ st' →
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_WhileFalse : ∀ b st c,
      beval st b = false →
      (WHILE b DO c END) / st \\ st
  | E_WhileTrue : ∀ st st' st'' b c,
      beval st b = true →
      c / st \\ st' →
      (WHILE b DO c END) / st' \\ st'' →
      (WHILE b DO c END) / st \\ st''

  where "c1 '/' st '\\' st'" := (ceval c1 st st').

Definition cequiv (c1 c2 : com) : Prop :=
  ∀ (st st' : state),
  (c1 / st \\ st') ↔ (c2 / st \\ st').
```