CIS 500 — Software Foundations
Midterm II

March 31, 2010

Name: _____

PennKey: _____

	Score
1	
2	
3	
4	
5	
6	
7	
8	
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. (4 points) The syntax and big-step semantics of IMP programs (over numbers and lists) are provided for your reference in the Appendix. Throughout the exam, we use informal notations in programs to make them easier to read; for example, we write `3 + 3` instead of `APlus (ANum 3) (ANum 3)` and `h :: t` instead of `ACons h t`.

   Recall the definition of equivalence for IMP programs:

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

   Which of the following pairs of programs are equivalent? Write "yes" or "no" for each one. (Where it appears, `a` is an arbitrary `aexp` — i.e., you should write "yes" only if the two programs are equivalent for every `a`.)

   (a)
```
Y ::= a :: X
```
   and
```
Y ::= a :: (head X) :: (tail X)
```

   (b)
```
WHILE IsCons Z DO
  Z ::= tail Z
END
```
   and
```
Z ::= nil
```

   (c)
```
WHILE X <> 0 DO
  X ::= X + 1
END
```
   and
```
WHILE X <> 0 DO
  Y ::= Y + 1
END
```

   (d)
```
Y ::= 0;
WHILE 0 <= X DO
  Y ::= Y + 3;
  X ::= X - 1
END
```
   and
```
Y ::= 3 * X;
X ::= 0
```

2. (10 points) Indicate whether or not each of the following Hoare triples is valid by writing either "valid" or "invalid." Also, for those that are invalid, give a counter-example. Where it appears, a is an arbitrary **aexp**—i.e., you should write "valid" only if the triple is valid for every a.

(a)
```
{ True } X ::= a { X = a }
```

(b)
```
{ X <= Y }
  IF Y == 0 THEN Y ::= 3 ELSE X ::= 0 FI
{ X = 0 }
```

(c)
```
{ X = h :: t }
  Y ::= 0;
  WHILE IsCons X DO
    IF Y == 0 THEN
      Y ::= head X
    ELSE
      SKIP
    FI;
    X ::= tail X
  END
{ Y = h }
```

(d)
```
{ 1 <= Y ∧ Y = a }
  Z ::= 0;
  WHILE Y <> 0 DO
    Z ::= Z + Y;
    X ::= X - 1
  END
{ Z = a * a }
```

(e)
```
{ True }
  WHILE Z <> 2 DO
    Z ::= Z + 2
  END
{ False }
```

3. (16 points) Recall the simple expression language with plus and numeric constants, which we introduced in Smallstep.v. (It is reproduced in the Appendix with both big-step and small-step semantics.)

   The following theorem captures the intuition that "big-step reduction implies small-step."

   Theorem: For all `t` and `v`, if `t ==> v` then `t -->* v`.

   In the space below, write a careful proof of this theorem in English. You may use the following lemmas without proving them:

```
Lemma stepmany_congr_1 : forall t1 t1' t2,
    t1 -->* t1' →
    tm_plus t1 t2 -->* tm_plus t1' t2.

Lemma stepmany_congr_2 : forall t1 t2 t2',
    value t1 →
    t2 -->* t2' →
    tm_plus t1 t2 -->* tm_plus t1 t2'.

Lemma rsc_trans : forall (X:Type) (R: relation X) (x y z : X),
    refl_step_closure R x y  →
    refl_step_closure R y z →
    refl_step_closure R x z.
```

4. (8 points) Conversely, we also saw in `Smallstep.v` that "small-step implies big-step." State the theorem (in formal Coq notation) that expresses this precisely. (Do not give a proof — just the statement of the theorem.)

5. (10 points) Recall the definition of stack machine programs and their small-step semantics (omitting the `SMinus` and `SMult` instructions):

```
Inductive sinstr : Type :=
| SPush : nat → sinstr
| SLoad : id → sinstr
| SPlus : sinstr.

Definition stack := list nat.
Definition prog  := list sinstr.

Inductive stack_step : state → prog * stack → prog * stack → Prop :=
  | SS_Push : forall st stk n p',
      stack_step st (SPush n :: p', stk)      (p', n :: stk)
  | SS_Load : forall st stk i p',
      stack_step st (SLoad i :: p', stk)      (p', st i :: stk)
  | SS_Plus : forall st stk n m p',
      stack_step st (SPlus :: p', n::m::stk)  (p', (m+n)::stk).
```

Suppose we want stack machine programs to also support a *store* instruction:

```
Inductive sinstr : Type :=
  ...      (* same instructions as before... *)
| SStore : id → sinstr.
```

Write down a new version of `stack_step` that handles the new instruction. Executing `SStore X` should have the effect of popping a value from the top of the stack and assigning it to `X`.

6. (9 points) Recall the `cstep` relation, which defines the small-step variant of the operational semantics of IMP (the full definition is given in the Appendix).

   Consider extending the syntax of IMP programs with a new primitive command called `ANYTHING`:

```
Inductive com : Type :=
...
| CAnything : com.

Notation "'ANYTHING'" := CAnything.
```

   The behavior of `ANYTHING` is completely nondeterministic: after it executes, the memory can be left in any state whatsoever. However, `ANYTHING` is guaranteed to always terminate.

   (a) What needs to be added to the definition of `cstep` to give `ANYTHING` this behavior?

   (b) Write down a Hoare rule for reasoning about `ANYTHING`, and explain why it is the right rule.

7. (15 points) Recall the definition of the `length` function from Basics.v:

```
Fixpoint length (xs:natlist) : nat :=
  match xs with
  | nil => O
  | h :: t => S (length t)
  end.
```

Here is a small Imp program that implements the same calculation:

```
                          {X = xs}
Y ::= 0;
WHILE X <> [] DO
  Y := Y + 1;
  X ::= tail(X)
END
                          {Y = length xs}
```

In the space below, write a fully decorated version of this program demonstrating that it validates the stated pre- and post-conditions.

8. (8 points) Here is a Coq function that returns `true` iff its argument is a list containing only zeros.

```
Fixpoint allzero (xs:natlist) : bool :=
  match xs with
  | nil => true
  | h :: t => andb (beq_nat h 0) (allzero t)
  end.
```

Here is an Imp program that performs the same test: when it terminates, the variable `Y` will be set to `0` iff the variable `X` at the beginning contains only zeros.

```
                          {X = xs}
Y ::= 0;
Z ::= [];
WHILE X <> [] DO
  IF head(X) <> 0 THEN
    Y ::= 1
  ELSE
    SKIP
  END;
  Z ::= head(X)::Z;
  X ::= tail(X)
                          {???}
END
                          {Y = 0 ↔ allzero xs}
```

If we want to prove that the program behaves as specified — *i.e.*, that the program together with the indicated pre- and post-conditions forms a valid Hoare triple — what invariant would we need to use for the loop; (*i.e.*, what assertion should we write in place of `???`)?

Appendix

IMP programs

The syntax and big-step semantics of IMP programs with lists is as follows:

```
Inductive val : Type :=
| VNat : nat → val
| VList : list nat → val.

Definition state := id → val.

Definition empty_state : state := fun _ => VNat 0.
Definition update (st : state) (V:id) (n : val) : state :=
  fun V' => if beq_id V V' then n else st V'.

Definition asnat (v : val) : nat :=
  match v with
  | VNat n => n
  | VList _ => 0
  end.

Definition aslist (v : val) : list nat :=
  match v with
  | VNat n => []
  | VList xs => xs
  end.

Inductive aexp : Type :=
  | ANum : nat → aexp
  | AId : id → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp
  | AHead : aexp → aexp
  | ATail : aexp → aexp
  | ACons : aexp → aexp → aexp
  | ANil  : aexp.

Definition tail (l : list nat) :=
  match l with
  | x::xs => xs
  | [] => []
  end.

Definition head (l : list nat) :=
  match l with
  | x::xs => x
  | [] => 0
  end.
```

```
Fixpoint aeval (st : state) (e : aexp) : val :=
  match e with
  | ANum n => VNat n
  | AId i => st i
  | APlus a1 a2   => VNat (asnat (aeval st a1) + asnat (aeval st a2))
  | AMinus a1 a2  => VNat (asnat (aeval st a1) - asnat (aeval st a2))
  | AMult a1 a2   => VNat (asnat (aeval st a1) * asnat (aeval st a2))
  | ATail a => VList (tail (aslist (aeval st a)))
  | AHead a => VNat (head (aslist (aeval st a)))
  | ACons a1 a2 => VList (asnat (aeval st a1) :: aslist (aeval st a2))
  | ANil => VList []
  end.

Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp → aexp → bexp
  | BLe : aexp → aexp → bexp
  | BNot : bexp → bexp
  | BAnd : bexp → bexp → bexp
  | BIsCons : aexp → bexp.

Fixpoint beval (st : state) (e : bexp) : bool :=
  match e with
  | BTrue       => true
  | BFalse      => false
  | BEq a1 a2   => beq_nat (asnat (aeval st a1)) (asnat (aeval st a2))
  | BLe a1 a2   => ble_nat (asnat (aeval st a1)) (asnat (aeval st a2))
  | BNot b1     => negb (beval st b1)
  | BAnd b1 b2  => andb (beval st b1) (beval st b2)
  | BIsCons a    => match aslist (aeval st a) with
                      | _::_ => true
                      | [] => false
                    end
  end.

Inductive com : Type :=
  | CSkip : com
  | CAss : id → aexp → com
  | CSeq : com → com → com
  | CIf : bexp → com → com → com
  | CWhile : bexp → com → com.

Notation "'SKIP'" :=
  CSkip  (at level 10).
Notation "l '::=' a" :=
  (CAss l 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' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
```

```
Inductive ceval : state → com → state → Prop :=
  | E_Skip : forall st,
      SKIP / st ==> st
  | E_Ass  : forall st a1 n l,
      aeval st a1 = n →
      (l ::= a1) / st ==> (update st l n)
  | E_Seq : forall c1 c2 st st' st'',
      c1 / st  ==> st' →
      c2 / st' ==> st'' →
      (c1 ; c2) / st ==> st''
  | E_IfTrue : forall st st' b1 c1 c2,
      beval st b1 = true →
      c1 / st ==> st' →
      (IFB b1 THEN c1 ELSE c2 FI) / st ==> st'
  | E_IfFalse : forall st st' b1 c1 c2,
      beval st b1 = false →
      c2 / st ==> st' →
      (IFB b1 THEN c1 ELSE c2 FI) / st ==> st'
  | E_WhileEnd : forall b1 st c1,
      beval st b1 = false →
      (WHILE b1 DO c1 END) / st ==> st
  | E_WhileLoop : forall st st' st'' b1 c1,
      beval st b1 = true →
      c1 / st ==> st' →
      (WHILE b1 DO c1 END) / st' ==> st'' →
      (WHILE b1 DO c1 END) / st ==> st''

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

Here is a small-step variant of the semantics for IMP as well. The `->a` and `->b` relations (not shown here) are small-step reduction relations for `aexp`s and `bexp`s.

```
Inductive cstep : (state * com) → (state * com) → Prop :=
  | CS_AssStep : forall st i a a',
    a / st -->a a' →
    (i ::= a) / st --> (i ::= a') / st
  | CS_Ass : forall st i n,
    (i ::= (ANum n)) / st --> SKIP / (update st i n)
  | CS_SeqStep : forall st c1 c1' st' c2,
    c1 / st --> c1' / st' →
    (c1 ; c2) / st --> (c1' ; c2) / st'
  | CS_SeqFinish : forall st c2,
    (SKIP ; c2) / st --> c2 / st
  | CS_IfTrue : forall st c1 c2,
    IFB BTrue THEN c1 ELSE c2 FI / st --> c1 / st
  | CS_IfFalse : forall st c1 c2,
    IFB BFalse THEN c1 ELSE c2 FI / st --> c2 / st
  | CS_IfStep : forall st b b' c1 c2,
    b / st -->b b' →
    IFB b THEN c1 ELSE c2 FI / st --> (IFB b' THEN c1 ELSE c2 FI) / st
  | CS_While : forall st b c1,
      (WHILE b DO c1 END) / st
    --> (IFB b THEN (c1; (WHILE b DO c1 END)) ELSE SKIP FI) / st

  where " t '/' st '-->' t' '/' st' " := (cstep (st,t) (st',t')).
```

Addition expression language

```
Inductive tm : Type :=
  | tm_const : nat → tm
  | tm_plus : tm → tm → tm.

Inductive value : tm → Prop :=
  v_const : forall n, value (tm_const n).

Inductive eval : tm → tm → Prop :=
  | E_Const : forall n1,
        tm_const n1 ==> tm_const n1
  | E_Plus : forall t1 n1 t2 n2,
        t1 ==> tm_const n1 →
        t2 ==> tm_const n2 →
        tm_plus t1 t2 ==> tm_const (plus n1 n2)

  where " t '==>' t' " := (eval t t').

Inductive step : tm → tm → Prop :=
  | ST_PlusConstConst : forall n1 n2,
      tm_plus (tm_const n1) (tm_const n2) --> tm_const (plus n1 n2)
  | ST_Plus1 : forall t1 t1' t2,
      t1 --> t1' →
      tm_plus t1 t2 --> tm_plus t1' t2
  | ST_Plus2 : forall n1 t2 t2',
      t2 --> t2' →
      tm_plus (tm_const n1) t2 --> tm_plus (tm_const n1) t2'

  where " t '-->' t' " := (step t t').

Inductive refl_step_closure {X:Type} (R: relation X)
                            : X → X → Prop :=
  | rsc_refl  : forall (x : X),
                  refl_step_closure R x x
  | rsc_step : forall (x y z : X),
                    R x y →
                    refl_step_closure R y z →
                    refl_step_closure R x z.

Definition stepmany := refl_step_closure step.

Notation " t '-->*' t' " := (stepmany t t') (at level 40).

Definition normal_form {X:Type} (R:relation X) (t:X) : Prop :=
  ~ exists t', R t t'.

Definition step_normal_form := normal_form step.

Definition normal_form_of (t t' : tm) :=
  (t -->* t' ∧ step_normal_form t').
```