CIS 500: Software Foundations **Midterm II**

April 7, 2016

(Standard and Advanced versions together)

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 do the questions intended for the other track.*

Mark the box of the track you wish to follow.

☐ Standard

1	/9
2	/12
3	/12
4	/12
5	/19
6	/16
7	ADVANCED ONLY/–
Total	/80

☐ Advanced

1	/9
2	/12
3	/12
4	/12
5	/19
6	STANDARD ONLY/–
7	/16
Total	/80

NOTE: Throughout this exam, we use slightly informal notation for Imp boolean expressions. For example, we write `X <> Y` to mean "X is not equal to Y". instead of the much more verbose `BNot (BEq X Y)`.

1. **Loop Invariants** (9 points)

   Recall that the assertion P appearing in the `hoare_while` rule is called the *loop invariant*. For each loop shown below, indicate which assertions are loop invariants. (There may be zero or more than one such assertion.)

   (a) `WHILE X<100 DO X ::= X+1 END`

   i. $\boxed{X > 10}$
   ii. $X < 100$
   iii. $\boxed{X \leq 100}$

   (b) `WHILE X>10 DO X ::= X+1 END`

   i. $\boxed{X > 10}$
   ii. $X < 100$
   iii. $X \leq 100$

   (c) `WHILE Y>0 DO Y := Y-1; Z ::= Z+1 END`

   i. $\boxed{X > 10}$
   ii. $\boxed{X = Y + Z}$
   iii. $\boxed{Y < Z}$

2. **Program Equivalances** (12 points)

For each pair of (standard) Imp commands below, write "equivalent" if the two programs are behaviorally equivalent (as defined by `cequiv` in the Appendix), or give a counterexample consisting of a single start state `st` that leads to different behaviors. You can indicate values for the variables in `st` by writing, for instance, `st X =` n. We have done the first one for you.

(a) $c_1 =$ `IFB X > 10 THEN X := 0 ELSE SKIP FI` $c_2 =$ `SKIP`

Answer: Counterexample: `st X = 500` (or `st X` $= n$ for any n larger than 10).

(b) $c_1 =$
```
WHILE 1 <= X DO
  X ::= X + 1
END
```
$c_2 =$
```
WHILE 2 <= X DO
  X ::= X + 1
END
```

Answer: Counterexample `st` such that `st X = 1`

(c) $c_1 =$ `X := Y;; Y := X` $c_2 =$ `Y := X;; X := Y`

Answer: Counterexample `st` such that `st X <> st Y`

(d) $c_1 =$
```
X := 0 ;;
WHILE X <> Y DO
  X ::= X + 1
END
```
$c_2 =$ `X := Y`

Answer: Equivalent

(e) $c_1 =$
```
IFB X <> Y THEN
  WHILE BTrue DO SKIP
ELSE
  SKIP
FI
```
$c_2 =$
```
WHILE X <> Y DO
  X := X + 1;;
  Y := Y + 1;;
END
```

Answer: Equivalent

3. **Hoare triples** (12 points)

Which of the the Hoare triples below are valid? If a triple is valid, circle the rules of Hoare logic that are *necessary* to justify the validity of that triple. You may need to circle more than one rule for a given triple, but do not circle a particular rule if the triple can be justified without it. Otherwise, if the triple is invalid, circle the last bullet.

For reference, the rules of Hoare logic are given in the Appendix, starting on page 12.

(a) $\{\{ X + 1 > 3 \}\}$ `X ::= X + 1` $\{\{ X > 3 \}\}$

- [✓ `hoare_asgn`]
- `hoare_skip`
- `hoare_while`
- `hoare_consequence`
- *Not a valid Hoare Triple*

(b) $\{\{ X > (Y + Y) \}\}$ `X ::= X - Y` $\{\{ X > Y \}\}$

- [✓ `hoare_asgn`]
- `hoare_skip`
- `hoare_while`
- [✓ `hoare_consequence`]
- *Not a valid Hoare Triple*

(c) $\{\{ X = X + 1 \}\}$ `SKIP` $\{\{ True \}\}$

- `hoare_asgn`
- [✓ `hoare_skip`]
- `hoare_while`
- [✓ `hoare_consequence`]
- *Not a valid Hoare Triple*

(d) $\{\{ True \}\}$ `WHILE BTrue DO X ::= X + 1 END` $\{\{ X = 2 \}\}$

- [✓ `hoare_asgn`]
- `hoare_skip`
- [✓ `hoare_while`]
- [✓ `hoare_consequence`]
- *Not a valid Hoare Triple*

Grading scheme: 2 points for each bullet. 1 point for getting valid/invalid correct and an additional point for marking the correct rules. No points were awarded for answers that were left blank: this problem asked how to prove these triples with the Hoare rules.

4. **Decorated Programs** (12 points)

The following Imp program computes the square of X and places the answer into Z.

```
Y ::= X ;;
Z ::= 0 ;;
WHILE Y <> 0 DO
  Z ::= Z + X ;;
  Y ::= Y - 1 ;;
END
```

On the next page, add appropriate annotations to the program in the provided spaces to show that the Hoare triple given by the outermost pre- and post-conditions is valid. Please be completely precise and pedantic in the way you apply the Hoare rules — i.e., write out assertions in *exactly* the form given by the rules (rather than logically equivalent ones). The provided blanks have been constructed so that, if you work backwards from the end of the program, you should only need to use the rule of consequence in the places indicated with `->>`.

The implication steps in your decoration may rely (silently) on usual rules of natural-number arithmetic, including:

- `mult_dist_sub : forall m n, m * (n - 1) = (m * n) - m`

The Hoare rules and the rules for well-formed decorated programs are provided on pages 13 and 14, for reference.

```
{{ X = m }} ->>
{{ X = m /\ 0 + X * X = m * m  }}
Y ::= X ;;
{{ X = m /\ 0 + X * Y = m * m  }};;
Z ::= 0
{{ X = m /\ Z + X * Y = m * m }}
WHILE Y <> 0 DO
  {{ X = m /\ Z + X * Y = m * m /\ Y <> 0 }} ->>
  {{ X = m /\ (Z + X) + X * (Y - 1) = m * m }}
  Z ::= Z + X ;;
  {{ X = m /\ Z + X * (Y - 1) = m * m }};;
  Y ::= Y - 1 ;;
  {{ X = m /\ Z + X * Y = m * m }}
END
{{ X = m  /\  Z + X * Y = m * m /\  ~(Y <> 0) }} ->>
{{ Z = m * m }}
```

Grading scheme:

- *1 point per implication*
- *3 points for correct "back propagation" of the mechanical parts of the annotation process*
- *4 points for the loop invariant*

5. **Operational Semantics** (19 points)

Some programming languages like Java and C have *effectful* expressions. For instance, `X++` evaluates to a number, but has the side effect of updating the state associated with variable `X` to increment its value. In this problem we consider adding such expressions to Imp.

This "post-increment" operator returns the *old* value of `X` before updating the state. That is, if in state `st` we have `st X = 0` then the behavior of `X++` is to return `0` and modify the state to `st'` such that `st' X = 1`.

The datatype below is a variant of `aexp` that includes the new post-increment operator (we omit minus and times for simplicity).

```
Inductive aexp : Type :=
  | ANum : nat -> aexp
  | AId : id -> aexp
  | APlus : aexp -> aexp -> aexp
  | AIncr : id -> aexp.  (* <----- NEW *)
```

The notation `X++` stands for `AIncr X`, where `X` is `AId 0`.

The old operational semantics given by the `aeval` function won't work because it does not allow us to return the modified state. To fix that problem, we modify `aeval` as shown below so that it returns both a natural number and the potentially modified state:

```
Fixpoint aeval (st : state) (a : aexp) : (nat * state) :=
  match a with
  | ANum n => (n, st)
  | AId x => (st x, st)
  | APlus a1 a2 =>
    let (n1, st1) := aeval st a1 in
    let (n2, st2) := aeval st1 a2 in
    (n1 + n2, st2)
  | AIncr x => (st x, t_update st x (1 + st x))
  end.
```

(Problem continues on the next page, nothing to do here.)

(a) (13 points) As with the original version of `aeval` we can also give *relational small-step semantics* for these expressions. This is defined by a relation on pairs of expressions and states, written $\texttt{a / st} \Rightarrow_a \texttt{a}' \texttt{ / st}'$, with the intuitive reading "Arithmetic expression `a` takes a small step starting from state `st` to `a'`, updating the state to be `st'`".

The "informal" small step rules given below have holes marked by boxes. Fill in the boxes so that these rules agree with the `aeval` function above. We have given you the rule for identifiers. There is no rule for `ANum n` terms.

$$\texttt{AId x / st} \Rightarrow_a \texttt{ANum(st x) / st} \quad \texttt{(id)}$$

$$\frac{\texttt{a1 / st} \Rightarrow_a \texttt{a1}' \texttt{ / st}'}{\texttt{APlus a1 a2 / st} \Rightarrow_a \texttt{APlus a1}' \texttt{ a2 / st}'} \quad \texttt{(plus\_left)}$$

$$\frac{\texttt{a2 / st} \Rightarrow_a \texttt{a2}' \texttt{ / st}'}{\texttt{APlus (ANum n) a2 / st} \Rightarrow_a \texttt{APlus (ANum n) a2}' \texttt{ / st}'} \quad \texttt{(plus\_right)}$$

$$\frac{}{\texttt{APlus (ANum n1) (ANum n2) / st} \Rightarrow_a \texttt{(ANum (n1} + \texttt{n2)) / st}} \quad \texttt{(plus)}$$

$$\frac{}{\texttt{AIncr (AId x) / st} \Rightarrow_a \texttt{ANum(st x) / t\_update st x (1} + \texttt{(st x))}} \quad \texttt{(incr)}$$

(b) (2 points) Which of the following is the best explanation for why there is no rule for stepping an `ANum n` expression? (Choose one.)

- `ANum n` is a *normal form* of the $\Rightarrow_a$ relation.
- We want to treat `ANum n` as a *value*, so for any `st`, the pair `ANum n / st` should be a *normal form* of the $\Rightarrow_a$ relation.
- We want to treat `ANum n` as a *value*, so for any expression `a` there should exist an `n` and `st'` such that $\texttt{a / st} \Rightarrow_a \texttt{ANum n / st'}$.
- `AIncr (ANum n)` should be considered a *stuck* state, and adding a step for `ANum n` would allow it to progress.

(c) (4 points) Adding side effects to the expression language can makes the order of evaluation import. The definition of `aeval` uses left-to-right evaluation order for `APlus` terms. Write down a term `a` that would evaluate to one answer using `aeval` as shown but give a *different* answer if we implemented `aeval` using right-to-left evaluation order.

Hint: this is equivalent to showing that the `APlus` operator of Imp's arithmetic language is not commutative. That is: `APlus a1 a2` will not necessarily yield the same result as `APlus a2 a1`

`(X + X) + (X++)`

6. **(Standard Only) Language Theory Concepts** (16 points)

(a) (4 points) Formulate the appropriate correctness theorem that shows that the `Fixpoint` definition of the version of `aeval` from problem 5 is equivalent to the multistep closure of the relational definition of $\Rightarrow_a$. There is no need to prove the theorem, just state it. (Use the informal notation for $\Rightarrow_a$.)

`Theorem aeval_equiv_asteps:`

$$\forall\ \texttt{a,n,st,st'}.\ (\texttt{aeval st a = (n, st')})\ \Leftrightarrow\ (\texttt{a} / \texttt{st} \Rightarrow_a^* \texttt{ANum n} / \texttt{st}')$$

(b) (3 points) Briefly describe one advantage of formalizing a language's operational semantics by using a Coq relation rather than Coq's `Fixpoint`.

Relations can be used to describe partial semantics (i.e. those that might diverge or are otherwise undefined).

(c) (3 points) Briefly describe one advantage of formalizing a language's semantics by using a small-step semantics rather than a big-step semantics.

Big-step semantics cannot distinguish stuck states from divergence. Also, small-step semantics can also be used to express concurrency, and other fine grained evaluation-order specific features.

(d) (6 points) Suppose we were to add to Imp a new command `print a` whose intended semantics is to output the value of the arithmetic expression `a` to the user via the terminal. Describe how you would modify Imp's large-step evaluation relation to formalize this new behavior. *Hint: It might be helpful to think about what the type of the modified version of* `ceval` *should be.*

7. **(Advanced Only) Informal Proof** (16 points)

   Write a careful informal proof showing that if the boolean expression `b` is equivalent to `BTrue`, then the command `IFB b THEN c1 ELSE c2 FI` is equivalent to `c1`—i.e., formally:

   $$\forall \texttt{ b c1 c2, (bequiv b BTrue)} \rightarrow \texttt{ cequiv (IFB b THEN c1 ELSE c2 FI) c1}$$

   The definitions of bequiv and cequiv are given in the Appendix, for reference.

   - $\rightarrow$ We must show, for all `st` and `st'`, that if `IFB b THEN c1 ELSE c2 F / st`$\Downarrow$ `st'` then `c1 / st` $\Downarrow$ `st'`. We proceed by cases on the rules that could possibly have been used to show `IFB b THEN c1 ELSE c2 F / st` $\Downarrow$ `st'` namely `EIfTrue` and `EIfFalse`.
     - Suppose the final rule rule in the derivation of `IFB b THEN c1 ELSE c2 F / st`$\Downarrow$ `st'` was `EIfTrue`. By the premises of `EIfTrue` we have that `c1 / st` $\Downarrow$ `st'`, which is exactly what we set out to prove.
     - On the other hand, suppose the final rule in the derivation of `IFB b THEN c1 ELSE c2 F / st` $\Downarrow$ `st'` was `EIfFalse`. We then know that `beval st b = false` and `c2/st` $\Downarrow$ `st'`. Recall that `b` is equivalent to `BTrue`, i.e. `forall st, beval st b = beval st BTrue`. In particular, this means that `beval st b = true`, since `beval st BTrue = true`. But this is a contradiction, since `EIfFalse` requires that `beval st b = false`. Thus, the final rule in the derivation could not have been `EIfFalse`.
   - $\leftarrow$ We must show, for all `st` and `st'`, that if `c1 / st` $\Downarrow$ `st'` then `IFB b THEN c1 ELSE c2 F / st` $\Downarrow$ `st'`. Since `b` is equivalent to `BTrue`, we know that `beval st b = beval st BTrue = true`. Together with the assumption that `c1 / st` $\Downarrow$ `st'`, we can apply `EIfTrue` to derive `IFB b THEN c1 ELSE c2 F / st` $\Downarrow$ `st'` as desired.

For Reference

Formal definitions for Imp

Syntax

```
Inductive aexp : Type := | ANum : nat -> aexp | AId : id -> aexp |
APlus : aexp -> aexp -> aexp | AMinus : aexp -> aexp -> aexp | AMult :
aexp -> aexp -> aexp.

Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp -> aexp -> bexp
  | BLe : aexp -> aexp -> bexp
  | BNot : bexp -> bexp
  | BAnd : bexp -> bexp -> bexp.

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.
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).
```

Evaluation relation

```
Inductive ceval : com -> state -> state -> Prop :=
  | E_Skip : forall st,
      SKIP / st || st
  | E_Ass  : forall st a1 n X,
      aeval st a1 = n ->
      (X ::= a1) / st || (update st X 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 c1 st st').
```

Program equivalence

```
Definition bequiv (b1 b2 : bexp) : Prop :=
  forall (st:state), beval st b1 = beval st b2.

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

Hoare triples

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  forall st st', c / st || st' -> P st  -> Q st'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q).
```

Implication on assertions

```
Definition assert_implies (P Q : Assertion) : Prop :=
  forall st, P st -> Q st.

Notation "P ->> Q" := (assert_implies P Q) (at level 80).
```

(ASCII `->>` is typeset as a hollow arrow in the rules below.)

Hoare logic rules

$$\frac{}{\{\!\{\, \texttt{assn\_sub X a } Q \,\}\!\}\ \texttt{X := a}\ \{\!\{\, Q \,\}\!\}}\ (\texttt{hoare\_asgn})$$

$$\frac{}{\{\!\{\, P \,\}\!\}\ \texttt{SKIP}\ \{\!\{\, P \,\}\!\}}\ (\texttt{hoare\_skip})$$

$$\frac{\begin{array}{c}\{\!\{\, P \,\}\!\}\ \texttt{c1}\ \{\!\{\, Q \,\}\!\} \\ \{\!\{\, Q \,\}\!\}\ \texttt{c2}\ \{\!\{\, R \,\}\!\}\end{array}}{\{\!\{\, P \,\}\!\}\ \texttt{c1;; c2}\ \{\!\{\, R \,\}\!\}}\ (\texttt{hoare\_seq})$$

$$\frac{\begin{array}{c}\{\!\{\, P \wedge b \,\}\!\}\ \texttt{c1}\ \{\!\{\, Q \,\}\!\} \\ \{\!\{\, P \wedge \sim b \,\}\!\}\ \texttt{c2}\ \{\!\{\, Q \,\}\!\}\end{array}}{\{\!\{\, P \,\}\!\}\ \texttt{IFB b THEN c1 ELSE c2 FI}\ \{\!\{\, Q \,\}\!\}}\ (\texttt{hoare\_if})$$

$$\frac{\{\!\{\, P \wedge b \,\}\!\}\ \texttt{c}\ \{\!\{\, P \,\}\!\}}{\{\!\{\, P \,\}\!\}\ \texttt{WHILE b DO c END}\ \{\!\{\, P \wedge \sim b \,\}\!\}}\ (\texttt{hoare\_while})$$

$$\frac{\begin{array}{c}\{\!\{\, P' \,\}\!\}\ \texttt{c}\ \{\!\{\, Q' \,\}\!\} \\ P \twoheadrightarrow P' \\ Q' \twoheadrightarrow Q\end{array}}{\{\!\{\, P \,\}\!\}\ \texttt{c}\ \{\!\{\, Q \,\}\!\}}\ (\texttt{hoare\_consequence})$$

$$\frac{\begin{array}{c}\{\!\{\, P' \,\}\!\}\ \texttt{c}\ \{\!\{\, Q \,\}\!\} \\ P \twoheadrightarrow P'\end{array}}{\{\!\{\, P \,\}\!\}\ \texttt{c}\ \{\!\{\, Q \,\}\!\}}\ (\texttt{hoare\_consequence\_pre})$$

$$\frac{\begin{array}{c}\{\!\{\, P \,\}\!\}\ \texttt{c}\ \{\!\{\, Q' \,\}\!\} \\ Q' \twoheadrightarrow Q\end{array}}{\{\!\{\, P \,\}\!\}\ \texttt{c}\ \{\!\{\, Q \,\}\!\}}\ (\texttt{hoare\_consequence\_post})$$

Decorated programs

1. `SKIP` is locally consistent if its precondition and postcondition are the same:

```
{{ P }}
SKIP
{{ P }}
```

2. The sequential composition of `c1` and `c2` is locally consistent (with respect to assertions `P` and `R`) if `c1` is locally consistent (with respect to `P` and `Q`) and `c2` is locally consistent (with respect to `Q` and `R`):

```
{{ P }}
c1;;
{{ Q }}
c2
{{ R }}
```

3. An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

```
{{ P [X |-> a] }}
X ::= a
{{ P }}
```

4. A conditional is locally consistent (with respect to assertions `P` and `Q`) if the assertions at the top of its "then" and "else" branches are exactly `P /\ b` and `P /\ ~b` and if its "then" branch is locally consistent (with respect to `P /\ b` and `Q`) and its "else" branch is locally consistent (with respect to `P /\ ~b` and `Q`):

```
{{ P }}
IFB b THEN
  {{ P /\ b }}
  c1
  {{ Q }}
ELSE
  {{ P /\ ~b }}
  c2
  {{ Q }}
FI
{{ Q }}
```

5. A while loop with precondition `P` is locally consistent if its postcondition is `P /\ ~b` and if the pre- and postconditions of its body are exactly `P /\ b` and `P`:

   ```
   {{ P }}
   WHILE b DO
     {{ P /\ b }}
     c1
     {{ P }}
   END
   {{ P /\ ~b }}
   ```

6. A pair of assertions separated by `->>` is locally consistent if the first implies the second (in all states):

   ```
   {{ P }} ->>
   {{ P' }}
   ```

Relations

```
Definition relation (X: Type) := X->X->Prop.

Inductive multi {X:Type} (R: relation X) : relation X :=
  | multi_refl  : forall (x : X), multi R x x
  | multi_step : forall (x y z : X),
                    R x y ->
                    multi R y z ->
                    multi R x z.

Notation " t '==>*' t' " := (multi step t t') (at level 40).
```