CIS 500 — Software Foundations

Midterm II

(Standard version)

November 7, 2013

Name: ______________________________

Pennkey (e.g. `bcpierce`): ______________________________

Scores:

1		16
2		12
3		6
4		6
5		20
6		10
7		10
Total:		80

1. (16 points)

Multiple choice. Mark *all* correct answers—there may be zero or more than one. The definition of Hoare triples is given on page 13, for reference.

(a) Which instances of assertion `P` make the following Hoare triple valid?

```
{{ P }}
X ::= 3;;
Y ::= X + Y
{{ X=3 /\ Y=5 }}
```

`P` is ...

- `X = 3 /\ Y = 2`
- `X = 2 /\ Y = 3`
- `X = 2 /\ Y = 2`
- `X = 3 /\ Y = 3`

(b) Which instances of assertion `P` make the following Hoare triple valid?

```
{{ P }}
X ::= X + Y;;
Y ::= X - Y
{{ X=m /\ Y=n }}
```

`P` is ...

- `True`
- `X = n /\ Y = m`
- `X = n /\ X + Y = m`
- `False`

(c) Which instances of assertion `P` make the following Hoare triple valid?

```
{{ P }}
WHILE X <= Y DO
  X = X + 1;;
  Y = Y + 1
END
{{ False }}
```

`P` is ...

- `X = 0 /\ Y = 1`
- `2*X <= 2*Y`
- `X+1 <= Y+2`
- `Y = X+3`

(d) Which instances of assertion `Q` make the following Hoare triple valid?

```
{{ X = Y }}
X ::= X + Y;;
Y ::= X - Y
{{ Q }}
```

`Q` is ...

- `True`
- `X = Y + Y`
- `Y = X - Y`
- `False`

2. (12 points) Given the following programs, group together those that are equivalent in Imp by drawing boxes around their names. For example, if you think programs a through h are all equivalent to each other, but not to i, your answer should look like this: $\boxed{a, b, c, d, e, f, g, h}$ $\boxed{i}$.

The definition of program equivalence is repeated on page 13, for reference.

(a)
```
SKIP
```

(b)
```
X ::= X + 1;;
Y ::= 0
```

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

(d)
```
WHILE Y = Y DO
  X ::= X + 1
END;;
Y ::= 0
```

(e)
```
WHILE Y <> 0 DO
  Y ::= Y - 1
END;;
X ::= X + 1
```

(f)
```
X ::= X + Y;;
WHILE Y <> 0 DO
  Y ::= Y - 2
END
```

(g)
```
IFB X <> 0 \/ Y <> 0 THEN
  SKIP
ELSE
  Y ::= X
FI
```

(h)
```
X := X;;
Y := Y;;
Z := Z
```

(i)
```
WHILE True DO
  SKIP
END
```

3. (6 points) Recall that the notion of program equivalence (`cequiv`) is also a *congruence* relation. Briefly explain what a congruence relation is (in the context of Imp programs) why it is relevant to program optimization that `cequiv` is a congruence relation.

(a) A congruence relation is ...

(b) The fact that `cequiv` is a congruence is useful for program optimizations because ...

4. (6 points) Are the following Hoare logic rules of inference valid? If not, give a counterexample.

(a)

$$\frac{\{\!\{ P \}\!\}\ \texttt{IF b THEN c1 ELSE c2 FI}\ \{\!\{ Q \}\!\}}{\{\!\{ P \}\!\}\ \texttt{c1}\ \{\!\{ Q \}\!\}} \quad (\texttt{hoare\_revif})$$

(b)

$$\frac{\{\!\{ P \}\!\}\ \texttt{c}\ \{\!\{ P \}\!\}}{\{\!\{ P \}\!\}\ \texttt{WHILE b DO c END}\ \{\!\{ P \}\!\}} \quad (\texttt{hoare\_whilealt})$$

5. (20 points) In this exercise we consider extending Imp with "nondeterministic choice of commands" of the form

```
DO c1 OR c2 OD
```

where `c1` and `c2` are commands. The idea is that evaluating such a command results in nondeterministically running *exactly one* of `c1` or `c2` but not both.

To formalize the extended language, we first add a clause to the definition of commands:

```
Inductive com : Type :=
  ...
  | COr : com -> com -> com.

Notation "'DO' c1 'OR' c2 'OD'" := (COr c1 c2).
```

(a) Refer to the definition of `ceval` (page 13) for the evaluation relation of Imp. What rule(s) must be added to this definition to formalize the behavior of `IF1`? Write out the additional rule(s) in formal Coq notation.

(b) Can the operational semantics for this version of Imp be implemented using Coq's `Fixpoint` functions with the following signature?

```
    Fixpoint ceval (st : state) (c : com) : state := ...
```

Briefly explain why or why not.

(c) For each purported theorem about Imp with `OR` commands below, write either "provable" if the claim is provable, or give a brief (one sentence) explanation, with a counterexample if possible, of why the claim is not provable. For your reference, the definition of `cequiv`, which remains unchanged from standard Imp, is found on page 13.

i.
```
Theorem thm1 : forall (c:com),
  cequiv   c    (DO c OR c OD).
```

ii.
```
Theorem thm2 :
  cequiv  (WHILE BTrue DO SKIP END)     (X ::= 0;;
                                         WHILE X = 0 DO
                                           DO X ::= 0 OR X ::= 1 OD
                                         END).
```

iii.
```
Theorem thm3 :
  cequiv  (X ::= 1)     (X ::= 0;;
                         WHILE X = 0 DO
                           DO X ::= 0 OR X ::= 1 OD
                         END).
```

iv.
```
Theorem thm4 : forall (c:com),
  cequiv   c    (DO c OR SKIP OD).
```

(d) Write a Hoare proof rule for the `OR` command. (For reference, the standard Hoare rules for Imp are provided on page 14.)

Try to come up with a rule that is both sound and as precise as possible.

6. (10 points)
The following Imp program computes the minimum of `a` and `b`, placing the answer into `Z`.

```
{{ True }}
X ::= a;;
Y ::= b;;
Z ::= 0;;
WHILE (X <> 0 /\ Y <> 0) DO
  X := X - 1;;
  Y := Y - 1;;
  Z := Z + 1
END
{{ Z = min a b }}
```

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. Use informal notations for mathematical formulae and assertions, but 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 the following facts, as well as the usual rules of arithmetic:

- `(X = 0 \/ Y = 0) ->  min X Y = 0`
- `min (X-1) (Y-1) = (min X Y) - 1`

The Hoare rules and the rules for well-formed decorated programs are provided on pages 14 and 15, for reference.

```
  {{ True }}
  {{                                                               }} ->>
X ::= a;;
  {{                                                               }}
Y ::= b;;
  {{                                                               }}
Z ::= 0;;
  {{                                                               }}
WHILE (X <> 0 /\ Y <> 0) DO
    {{                                                               }} ->>
    {{                                                               }}
  X := X - 1;;
    {{                                                               }}
  Y := Y - 1;;
    {{                                                               }}
Z := Z + 1
    {{                                                               }}
END
  {{                                                               }} ->>
  {{ Z = min a b }}
```

7. (10 points)
Recall that

$$\sum_{X=1}^{X=n} X \quad = \quad \frac{n * (n+1)}{2}$$

The following Imp program calculates the sum above into the variable `Y`.

```
  {{ True }}
X ::= 0;;
Y ::= 0;;
WHILE X <> n DO
  X ::= X + 1;;
  Y ::= Y + X
END
  {{ Y = (n * (n+1))/2 }}
```

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. Use informal notations for mathematical formulae and assertions, but 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 the following fact about natural number division, which follows from the usual rules of arithmetic:

- `(n + 2*m)/2 = n/2 + m`

The Hoare rules and the rules for well-formed decorated programs are provided on pages 14 and 15, for reference.

```
  {{ True }} ->>
  {{                                                               }}
X ::= 0;;
  {{                                                               }}
Y ::= 0 ;;
  {{                                                               }}
WHILE X <> n DO
    {{                                                               }} ->>
    {{                                                               }}
  X ::= X + 1;;
    {{                                                               }}
  Y ::= Y + X
    {{                                                             }}
END
  {{                                                               }} ->>
  {{ Y = (n * (n + 1))/2 }}
```

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

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

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

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

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

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

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

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

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

Decorated programs

(a) `SKIP` is locally consistent if its precondition and postcondition are the same:

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

(b) 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 }}
```

(c) An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

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

(d) 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 }}
```

(e) 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 }}
```

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

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