# CIS 500 — Software Foundations

**Midterm II**

**March 30, 2011**

**Answer key**

The core definitions of the Imp language are repeated, for easy reference, in the handout (pages 7 to 8). Now consider extending Imp with commands of the form

```
FLIP X
```

where `X` is an identifier. The effect of executing `FLIP X` is to assign either `0` or `1` to `X`, nondeterministically. For example, after executing the program

```
FLIP Y;
Z := Y + 2
```

the value of `Z` might be `2` or it might be `3`, and those are the only two possibilities. (Note that we are not saying anything about the *probabilities* of the two outcomes—just that both can happen.)

Let's call this new language *Flimp* ("Imp extended with `FLIP`"). Questions 1–4 all refer to Flimp.

1. (6 points) To formalize the extended language, we first add a clause to the definition of commands:

```
Inductive com : Type :=
  ...
  | CFlip : id -> com.

Notation "'FLIP' l" := (CFlip l) (at level 60).
```

Next, we must extend the operational semantics. The `cstep` relation (shown on page 8) defines a small-step semantics for Imp. What rule(s) must be added to the definition of `cstep` to formalize the behavior of the `FLIP` command in Flimp? Write out the additional rule(s) in formal Coq notation.

*Answer:*

```
  | CS_Flip0 : forall st i,
      (FLIP i) / st ==> SKIP / update st i 0
  | CS_Flip1 : forall st i,
      (FLIP i) / st ==> SKIP / update st i 1
```

*or*

```
  | CS_Flip : forall st i n,
      (n = 0 \/ n = 1) ->
      (FLIP i) / st ==> SKIP / update st i n
```

2. (6 points) Write down a Hoare logic rule for `FLIP` commands, and briefly explain why it is the right rule. (For reference, the standard Hoare rules for Imp are provided on page 9 of the handout.)

*Answer:*

$$\frac{}{\{\texttt{assn\_sub X 0 } P \land \texttt{assn\_sub X 1 } P\}\ \texttt{FLIP X}\ \{P\}}\ \textsc{Hoare\_Flip}$$

In order to prove that $P$ will hold after executing `FLIP X`, we need to know that $P$ holds *both* when substituting 0 and 1 for `X`, since we don't know ahead of time which value `X` will take on.

3. (10 points) Which of the following pairs of programs are equivalent? Write "yes" or "no" for each one.

(a)
```
WHILE X > 0 DO          and        WHILE X > 0 DO
  X ::= X + 1                        Y ::= Y - 1
END                                END
```
*Answer: Yes*

(b)
```
IFB X < 10 THEN                 and        WHILE X < 10 DO
  X ::= X + Y - Z;                           X ::= X + Y - Z;
  Y ::= X * 3 - 4                            Y ::= X * 3 - 4
  WHILE X < 10 DO                          END
    X ::= X + Y - Z;
    Y ::= X * 3 - 4
  END
ELSE
  SKIP
FI
```
*Answer: Yes*

(c)
```
WHILE X <> 0 DO          and        WHILE X <> 0 DO
  FLIP Y;                             SKIP
  X ::= X + 1                       END
END
```
*Answer: Yes*

(d)
```
Z ::= 1;                 and        X ::= 0;
WHILE X <> 0 DO                     Z ::= 1
  FLIP X;
  FLIP Z
END
```
*Answer: No*

(e)
```
WHILE X <> 1 DO          and        X ::= 1
  FLIP X
END
```
*Answer: Yes*

*Grading scheme: 1 point each for parts a, b, c, and f. 2 points for the others.*

4. (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. (Note that, in part d, the variable `a` represents an arbitrary `aexp` – i.e., you should write "valid" only if the triple is valid for *every* `a`. If you give a counter-example, specify which `a` it applies to.)

(a)
```
{ X = 1 } X ::= 1 { X = 1 }
```

*Answer: Valid.*

(b)
```
{ X = 0 }
WHILE X > 0 DO
  X ::= X + 1
END
{ X > 0 }
```

*Answer: Invalid*

(c)
```
{ X = 0 } FLIP Y { X = 0 }
```

*Answer: Valid.*

(d)
```
{ X = a } FLIP Y { X = a }
```

*Answer: Invalid: consider* `a = Y`

(e)
```
{ True }
FLIP X;
IFB X = 0 THEN Y ::= 2 ELSE Y ::= 1 FI
{ Y > X }
```

*Answer: Invalid: consider the case where* `X` *gets set to 1, in which case* `Y` *also becomes 1.*

(f)
```
{ False }
FLIP X;
{ X = 0 }
```

*Answer: Valid*

(g)
```
{ True }
FLIP X;
WHILE X <> 0 DO
  Y ::= X
END;
{ Y = 1 }
```

*Answer: Invalid: consider the case where* `X` *gets set to 0, in which case* `Y`*'s value remains unchanged.*

5. (24 points) We can define the mathematical *min* function in Coq as follows:

```
Definition min (x:nat) (y:nat) : nat :=
  if beq_nat (x - y) 0 then x else y.
```

The following Imp program calculates the minimum of two numbers `a` and `b`, in the sense that, when it terminates, the program variable `Z` will be set to `min a b`.

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

Note that, as usual when dealing with decorated programs, we're using informal notations, for example writing

```
WHILE (X <> 0 /\ Y <> 0)
```

instead of:

```
WHILE (BAnd (BNot (BEq (AId X) (ANum 0)))
            (BNot (BEq (AId Y) (ANum 0))))
```

On the next page, add appropriate annotations to the program in the provided spaces to demonstrate this fact. 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). Note that 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 `=>`. (Again, remember that the Hoare rules are provided on page 9 of the handout.)

For the `=>` steps in your annotations, you may rely (silently) on the following facts about `min`

```
Lemma lemma1 : forall x y,
  (x=0 \/ y=0) -> min x y = 0.

Lemma lemma2 : forall x y,
  min (x-1) (y-1) = (min x y) - 1.
```

plus, as usual, standard high-school algebra.

*Solution:*

```
  {{ True }}
  =>
  {{ 0 + min a b = min a b }}
X ::= a;
  {{ 0 + min X b = min a b }}
Y ::= b;
  {{ 0 + min X Y = min a b }}
Z ::= 0;
```

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

6. (24 points) Suppose we define a simple language of numbers and constants, similar to the toy language used in the `Smallstep.v` chapter. Terms $t$ are either of the form $\mathsf{const}\ n$ for some natural number constant $n$, or of the form $\mathsf{add}\ t_1\ t_2$ for some terms $t_1$ and $t_2$:

$$t ::= \mathsf{const}\ n \mid \mathsf{add}\ t\ t$$

We defined a big-step evaluation relation $t \Downarrow n$ for this language as follows:

$$\frac{}{\mathsf{const}\ n \Downarrow n}\ \textsc{E\_Const}$$

$$\frac{t_1 \Downarrow n_1 \qquad t_2 \Downarrow n_2}{\mathsf{add}\ t_1\ t_2 \Downarrow n_1 + n_2}\ \textsc{E\_Plus}$$

We also defined a small-step evaluation relation $t \Longrightarrow t'$:

$$\frac{}{\mathsf{add}\ (\mathsf{const}\ n_1)\ (\mathsf{const}\ n_2) \Longrightarrow \mathsf{const}\ (n_1 + n_2)}\ \textsc{ST\_PlusConstConst}$$

$$\frac{t_1 \Longrightarrow t_1'}{\mathsf{add}\ t_1\ t_2 \Longrightarrow \mathsf{add}\ t_1'\ t_2}\ \textsc{ST\_Plus1}$$

$$\frac{t_2 \Longrightarrow t_2'}{\mathsf{add}\ (\mathsf{const}\ n_1)\ t_2 \Longrightarrow \mathsf{add}\ (\mathsf{const}\ n_1)\ t_2'}\ \textsc{ST\_Plus2}$$

In `Smallstep.v`, we proved the equivalence of these two ways of presenting the semantics. One piece of that proof was the lemma shown below. Write out a careful informal proof of this lemma in English.

*Lemma*: For all terms $t$ and $t'$ and numbers $n$, if $t \Longrightarrow t'$ and $t' \Downarrow n$, then $t \Downarrow n$.

*Answer:*

*Proof:* Let $t$ and $t'$ be terms. We prove by induction on a derivation of $t \Longrightarrow t'$ that, for all natural numbers $n$, if $t' \Downarrow n$ then $t \Downarrow n$.

- Suppose the last rule in the derivation of $t \Longrightarrow t'$ was ST_PlusConstConst. Then $t = \mathsf{add}\ (\mathsf{const}\ n_1)\ (\mathsf{const}\ n_2)$ and $t' = \mathsf{const}\ (n_1 + n_2)$ for some $n_1$ and $n_2$. Let $n$ be a natural number and suppose $\mathsf{const}\ (n_1 + n_2) \Downarrow n$. We must show that $\mathsf{add}\ (\mathsf{const}\ n_1)\ (\mathsf{const}\ n_2) \Downarrow n$. But we know by inversion on $\mathsf{const}\ (n_1 + n_2) \Downarrow n$ that $n_1 + n_2 = n$, and hence this follows by E_Plus and two applications of E_Const.

- Suppose the last rule in the derivation of $t \Longrightarrow t'$ was ST_Plus1. Then $t = \mathsf{add}\ t_1\ t_2$ and $t' = \mathsf{add}\ t_1'\ t_2$ where $t_1 \Longrightarrow t_1'$. The induction hypothesis tells us that for all natural numbers $n'$, if $t_1' \Downarrow n'$ then $t_1 \Downarrow n'$.

  Now let $n$ be a natural number and suppose $\mathsf{add}\ t_1'\ t_2 \Downarrow n$; we must show that $\mathsf{add}\ t_1\ t_2 \Downarrow n$. By inversion, there are natural numbers $n_1$ and $n_2$ such that $t_1' \Downarrow n_1$, $t_2 \Downarrow n_2$, and $n = n_1 + n_2$. By the induction hypothesis (with $n' = n_1$), we know $t_1 \Downarrow n_1$ as well, and hence the desired result follows by E_Plus.

- The case where the last rule is ST_Plus2 is similar.

## IMP programs

Here are the key definitions for the syntax and small-step semantics of IMP programs:

```
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).
```

(Remember that the `==>a` and `==>b` relations — not shown here — are small-step reduction relations for `aexps` and `bexps`.)

```
Inductive cstep : (com * state) -> (com * state) -> 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 (t,st) (t',st')).
```

**Hoare logic rules**

$$\frac{}{\{\texttt{assn\_sub V a } Q\}\ \texttt{V := a}\ \{Q\}}\ \textsc{Hoare\_Asgn}$$

$$\frac{\{P'\}\ \texttt{c}\ \{Q'\} \qquad P \longrightarrow P' \qquad Q' \longrightarrow Q}{\{P\}\ \texttt{c}\ \{Q\}}\ \textsc{Hoare\_Consequence}$$

$$\frac{\{P'\}\ \texttt{c}\ \{Q\} \qquad P \longrightarrow P'}{\{P\}\ \texttt{c}\ \{Q\}}\ \textsc{Hoare\_Pre} \qquad \frac{\{P\}\ \texttt{c}\ \{Q'\} \qquad Q' \longrightarrow Q}{\{P\}\ \texttt{c}\ \{Q\}}\ \textsc{Hoare\_Post}$$

$$\frac{}{\{P\}\ \texttt{SKIP}\ \{P\}}\ \textsc{Hoare\_Skip} \qquad \frac{\{P\}\ \texttt{c1}\ \{Q\} \qquad \{Q\}\ \texttt{c2}\ \{R\}}{\{P\}\ \texttt{c1 ; c2}\ \{R\}}\ \textsc{Hoare\_Seq}$$

$$\frac{\{P \wedge b\}\ \texttt{c1}\ \{Q\} \qquad \{P \wedge \sim b\}\ \texttt{c2}\ \{Q\}}{\{P\}\ \texttt{IFB b THEN c1 ELSE c2 FI}\ \{Q\}}\ \textsc{Hoare\_If}$$

$$\frac{\{P \wedge b\}\ \texttt{c}\ \{P\}}{\{P\}\ \texttt{WHILE b DO c END}\ \{P \wedge \sim b\}}\ \textsc{Hoare\_While}$$