CIS 500 — Software Foundations
Midterm II

April 1, 2009

Name: ______________________________

Email: ______________________________

	Score
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
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. (5 points) Recall the definition of equivalence for `while` 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)
```
X ::= A4
```
and
```
Y ::= A2 +++ A2;
X ::= Y
```

(b)
```
X ::= a;
Y ::= a
```
and
```
Y ::= a;
X ::= a
```

(c)
```
while BTrue do (X := !X +++ 1)
```
and
```
X := !X +++ 1
```

(d)
```
while BTrue do (X := !X +++ 1)
```
and
```
while BTrue do (X := !X --- 1)
```

(e)
```
while BFalse do (X := !X +++ 1)
```
and
```
skip
```

2. (5 points) Is this claim...

   Claim: Suppose the command `c` is equivalent to `c;c`. Then, for any `b`, the command

   ```
   while b do c
   ```

   is equivalent to

   ```
   testif b then c else skip.
   ```

   ... true or false? Briefly explain.

3. (5 points) Recall that a *program transformation* is a function from commands to commands. What does it mean to say that a program transformation is "sound"? (Answer either informally or with a Coq definition.)

4. (7 points) Recall the definition of a valid Hoare triple:

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

Indicate whether or not each of the following Hoare triples is valid by writing either "valid" or "invalid." 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 = 1}}
testif (!X === a) then (while BTrue do Y ::= !X) else (Y ::= A0)
{{Y = 0}}
```

(c)
```
{{True}}
Y ::= A0; Y ::= A1
{{Y = 1}}
```

(d)
```
{{False}}
X ::= A3
{{X = 0}}
```

(e)
```
{{True}}
skip
{{False}}
```

(f)
```
{{X = 5 ∧ Y = X}}
Z ::= 0; while BNot (!X === A0) do (Z ::= !Z +++ !Y; X ::= !X --- 1)
{{Z = 25}}
```

(g)
```
{{X = 1}}
while BNot (!X === A0) do X ::= !X +++ 1
{{X = 42}}
```

5. (9 points) Give the weakest precondition for each of the following commands. (Please use the informal notation for assertions rather than Coq notation—i.e., write `X = 5`, not `fun st => st X = 5`.)

(a)

```
{{ ? }}  X ::= A5  {{ X = 5 }}
```

(b)

```
{{ ? }}  X ::= A0  {{ X = 5 }}
```

(c)

```
{{ ? }}  X ::= !X +++ !Y  {{ X = 5 }}
```

(d)

```
{{ ? }}  while A1 <<= !X do (X ::= !X---A1; Y ::= !Y---A1)  {{ Y = 5 }}
```

(e)

```
{{ ? }} while !X === A0 do Y ::= A1 {{ Y = 1 }}
```

(f)

```
{{ ? }}
  testif !X === A0
    then Y ::= !Z
    else Y ::= !W
{{ Y = 5 }}
```

6. (5 points) The notion of weakest precondition has a natural dual : given a precondition and a command, we can ask what is the *strongest postcondition* of the command with respect to the precondition. Formally, we can define it like this:

   `Q` is the strongest postcondition of `c` for `P` if:

   (a) `{{P}} c {{Q}}`, and

   (b) if `Q′` is an assertion such that `{{P}}c{{Q′}}`, then `Q st` implies `Q′ st`, for all states `st`.

   `Q` is the strongest (most difficult to satisfy) assertion that is guaranteed to hold after `c` if `P` holds before.

   For example, the strongest postcondition of the command `skip` with respect to the precondition `Y = 1` is `Y = 1`. Similarly, the postcondition in...

```
{{ Y = y }}
  if !Y === A0 then X ::= A0 else Y ::= !Y *** A2
{{ (Y = y = X = 0) ∨ (Y = 2*y ∧ y <> 0) }}
```

   ...is the strongest one.

   Complete each of the following Hoare triples with the strongest postcondition for the given command and precondition.

   (a) `{{ Y = 1 }}  X ::= !Y +++ A1  {{ ? }}`

   (b) `{{ True }}  X ::= A5  {{ ? }}`

   (c) `{{ True }}  skip  {{ ? }}`

   (d) `{{ True }}  while BTrue do skip  {{ ? }}`

   (e)

```
{{ X = x ∧ Y = y }}
  while BNot (!X === A0) do (
    Y ::= !Y +++ A2;
    X ::= !X --- A1
  )
{{ ? }}
```

7. (12 points) The following program performs integer division:

```
div =
     Q ::= A0;
     R ::= ANum x;
     while (ANum y) <<= !R do (
        R ::= !R --- (ANum y);
        Q ::= !Q +++ A1
     )
```

If x and y are numbers, running this program will yield a state where Q is the quotient of x by y and R is the remainder. (We assume that program variables Q and R are defined.)

Fill in the blanks in the following to obtain a correct decorated version of the program:

```
                                { 0<y } =>
                                { 0=0 ∧ x=x ∧ 0<y }
Q ::= A0;
                                { Q=0 ∧ x=x ∧ 0<y } =>
R ::= ANum x;
                                { Q=0 ∧ R=x ∧ 0<y } =>

                                { __________________________________________ }

while (ANum y <<= !R) do (
                                { __________________________________________ } =>


                                { __________________________________________ }
   R ::= !R --- (ANum y);

                                { __________________________________________ }
   Q ::= !Q +++ A1

                                { __________________________________________ }
)

                                { __________________________________________ } =>

                                { x=Q*y+R ∧ R<y }
```

8. (4 points) Suppose we change the initial pre-condition in problem 7 from 0<y to True (i.e., we allow y to be zero). Does the specification now make an incorrect claim — i.e., is the Hoare triple

```
{{ True }}  div  {{ x=Q*y+R ∧ R<y }}
```

invalid, or is it valid? Briefly explain your answer.

9. (6 points) Recall the syntax...

```
Inductive com : Set :=
  ...
  | CWhile : bexp → com → com
```

...and operational semantics of the `while...do...` construct:

```
Inductive ceval : state → com → state → Prop :=
  ...
  | CEWhileEnd : forall b1 st c1,
      beval st b1 = false →
      ceval st (CWhile b1 c1) st
  | CEWhileLoop : forall st st' st'' b1 c1,
      beval st b1 = true →
      ceval st c1 st' →
      ceval st' (CWhile b1 c1) st'' →
      ceval st (CWhile b1 c1) st''
```

Suppose we extend the syntax with one more constructor...

```
  | CLoopWhile : com → bexp → com
```

...written `loop c while b`:

```
 Notation "'loop c 'while' b" := (CLoopWhile c b).
```

The intended behavior of this construct is almost like that of `while...do...` except that the condition is checked at the *end* of the loop body instead of the beginning (so the body always executes at least once). For example,

```
X ::= A1;
loop
  X ::= !X +++ A1
while
  !X <<= A1
```

will leave `X` with the value `2`.

To define the operational semantics of `loop...while...` formally, we need to add two more rules to the `Inductive` declaration of `ceval`. Write these rules in the space below.

10. (6 points) Having extended the language of commands with `loop...while...`, the next thing we want is a Hoare rule for reasoning about programs that use this construct. Recall the rule for `while...do...`:

$$\frac{\{\{P \wedge b\}\}\ \ c\ \ \{\{P\}\}}{\{\{P\}\}\ \ \texttt{while b do c}\ \ \{\{P \wedge \sim b\}\}}$$

Write an analogous rule for `loop...while...`.

11. (4 points) Recall (from the review session on Monday) the small-step variant of the operational semantics of IMP. The `astep` and `bstep` relations (not shown here) are small-step reduction relations for `aexp`s and `bexp`s. The small-step relation for commands is defined as follows:

```
Inductive cstep : state → com → com → state → Prop :=
  | CSAssStep : forall st i a a',
    astep st a a' →
    cstep st (CAss i a) (CAss i a') st
  | CSAss : forall st i n,
    cstep st (CAss i (ANum n)) CSkip (extend st i n)
  | CSSeqStep : forall st c1 c1' st' c2,
    cstep st c1 c1' st' →
    cstep st (CSeq c1 c2) (CSeq c1' c2) st'
  | CSSeqFinish : forall st c2,
    cstep st (CSeq CSkip c2) c2 st
  | CSIfTrue : forall st c1 c2,
    cstep st (CIf BTrue c1 c2) c1 st
  | CSIfFalse : forall st c1 c2,
    cstep st (CIf BFalse c1 c2) c2 st
  | CSIfStep : forall st b b' c1 c2,
    bstep st b b' →
    cstep st (CIf b c1 c2) (CIf b' c1 c2) st
  | CSWhile : forall st b c1,
    cstep st (CWhile b c1) (CIf b (CSeq c1 (CWhile b c1)) CSkip) st.
```

Suppose we extend the syntax of commands with `loop...while...`, as in the previous two problems. What needs to be added to the definition of `cstep`?

12. (12 points) Recall the following definitions from `Smallstep.v`:

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

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

Inductive step : tm → tm → Prop :=
  | ES_PlusConstConst : forall n1 n2,
        step (tm_plus (tm_const n1) (tm_const n2))
             (tm_const (plus n1 n2))
  | ES_Plus1 : forall t1 t1' t2,
        (step t1 t1')
      → step (tm_plus t1 t2)
             (tm_plus t1' t2)
  | ES_Plus2 : forall v1 t2 t2',
        (value v1)
      → (step t2 t2')
      → step (tm_plus v1 t2)
             (tm_plus v1 t2').
```

In class, we discussed the Progress Theorem:

> *Theorem:* If `t` is a term, then either `t` is a value or else there exists some term `t'` such that `t` steps to `t'`.

Write a careful informal proof of this theorem.