SOLUTIONS

1. **[Standard Track Only] Behavioral Equivalence** (14 points)

Recall the notion of *program equivalence*, `cequiv`, from the Equiv chapter. Two Imp commands c_1 and c_2 are equivalent if for every starting state `st`, either c_1 and c_2 both diverge or both terminate in the same final state `st'`. For the following pairs of Imp programs mark *True* if the pair of Imp programs are equivalent. If they're not equivalent, mark *False*.

(a)

```
X := Y; Y := X
```

```
Y := X; X := Y
```

☐ *True* ☒ *False*

(b)

```
skip
```

```
while false do skip end
```

☒ *True* ☐ *False*

(c)

```
X := 10;
Y := 0;
while X <> Y do
   Y := Y + 1
end
```

```
Y := X
```

☐ *True* ☒ *False*

(d)

```
X := 10;
Y := 0;
while X <> Y do
   skip
end
```

```
skip
```

☐ *True* ☒ *False*

(e) Suppose that `c1` is equivalent to `c2`, then it follows that the following two programs are equivalent for any `b`:

```
if b then
  c1
else
  c2
```

```
if ~b then
  c1
else
  c2
```

☒ *True* ☐ *False*

(f) Suppose that `c1` is equivalent to `c2`, then for every definition of `c3`, the following programs are *equivalent*:

```
c1 ; c3
```

```
c2 ; c3
```

☒ *True* ☐ *False*

(g) Suppose that `c1` is *not* equivalent to `c2`, then it is not possible to define `c3` such that the following programs are *equivalent*:

```
c1 ; c3
```

```
c2 ; c3
```

☐ *True* ☒ *False*

2. **[Advanced Track Only] Behavioral Equivalence** (14 points total)

Suppose we wanted to add a command `print a` to Imp, which is intended to model printing the (arithmetic) expression `a` to the terminal. This problem explores the ramifications of that choice on behavioral equivalence. The state of the Imp semantics will now include a *log* of the natural numbers output on the console, represented as a list.

For the large step semantics, the rules are the same as in Appendix A except that they also propagate the log unchanged. For example, the modified rule for `skip` and the new rule for `print` are shown below:

```
  -------------------------------                (E_Skip)
   st / log =[ skip ]⇒ st / log

        aeval st a = n
-------------------------------------            (E_Print)
st / log =[ print a ]⇒ st / (n::log)
```

(a) (4 points) If we straightforwardly modify the statement of behavioral equivalence to use this definition, we end up with the following:

```
Definition cequiv (c1 c2 : com) : Prop :=
  ∀ (st st' : state) (log log' : list nat),
    (st / log =[ c1 ]⇒ st' / log') ↔ (st / log =[ c2 ]⇒ st' / log').
```

Suppose that the terminal output becomes visible to a user *immediately* after the `print` command executes. Briefly describe why this definition of program equivalence is *not* very satisfactory if we intend it to model what would be seen by a user who is watching the terminal. Give a specific example of two programs whose behavior would be mischaracterized if we use this definition.

This definition makes it so that two programs that diverge but produce different outputs are considered to be the same. For example, the following two programs would be considered the same, even though they produce different outputs:

```
while true to            while true do
  print 1                  print 2
end                      end
```

[Advanced Track Only]

It is harder to fix the problems with `cequiv` hinted at above than you might expect. One idea is to move to a small-step semantics for modeling this situation. As with the usual rules for small-step Imp, there is no rule for stepping `skip`, but we would add these rules for `print`:

```
---------------------------------------------        (C_Print)
 print n / st / log  ⟶ skip / st / (n::log)


                   a ⟶a a'
-------------------------------------------          (C_PrintStep)
 print a / st / log ⟶ print a' / st / log
```

The other rules propagate the log unchanged. For example, we have these rules for `While` and `IfTrue`:

```
                                                                                  (C_While)
------------------------------------------------------------------------------------------
while b do c end / st / log  ⟶ if b then c ; while b do c end else skip end / st / log


------------------------------------------------------  (C_IfTrue)
if true then c1 else c2 end / st / log ⟶ c1 / st / log
```

Recall that $\longrightarrow *$ is the reflexive, transitive closure of $\longrightarrow$, given by `multi step` (see Appendix B). Consider the following first try at a definition for behavioral equivalence for this situation.

```
Definition cequiv_first_try (c1 c2 : com) : Prop :=
  ∀ st log st' log',
  ∀ c1', (c1 / st / log ⟶* c1' / st' / log') →
       ∃ c2', (c2 / st / log ⟶* c2' / st' / log')
```

Informally, the definition above says that if `c1` can be run from starting state `st` with `log` to reach a state `st'` and output `log'`, then so can `c2`. Unfortunately, this *still* isn't quite right, for several reasons.

(b) (2 points) For one thing, one of the desired properties of an behavioral equivalence fails to hold. Which property fails for the above definition? (Mark one)

- ☐ Reflexivity
- ☒ Symmetry
- ☐ Transitivity
- ☐ Congruence

In an attempt to fix the problem above, we might try this definition instead:

```
Definition cequiv_second_try (c1 c2 : com) : Prop :=
  ∀ st log st' log',
  ∀ c1', (c1 / st / log ⟶* c1' / st' / log') →
       ∃ c2', (c2 / st / log ⟶* c2' / st' / log')
      ∧
  ∀ c2', (c2 / st / log ⟶* c2' / st' / log') →
       ∃ c1', (c1 / st / log ⟶* c1' / st' / log')
```

This definition *does* satisfy all of the properties from part (b), but it is also *too coarse*—it equates too many commands. Ironically, unlike for the large-step version of `cequiv` in part (a), the presence of `print` isn't the problem—this definition does correctly distinguish programs that print different outputs. However, sometimes this definition claims that two commands without `print` are equal when `cequiv` correctly distinguishes them.

[Advanced Track Only]

In particular, the commands `c1 = skip` and `c2 = while true do skip end`, which are *inequivalent* according to `cequiv` are incorrectly equated by the proposed `cequiv_second_try`.

(c) (4 points) One direction of the proof of equivalence using `cequiv_second_try` is pretty easy. Below, we give an informal proof of that claim with **three choices** left for you to determine. Mark the boxes to complete the proof.

Lemma Undesired Equivalence Left-to-Right: `∀ st log st' log',`

`∀ c1', (skip / st / log ⟶* c1' / st' / log') →`

`∃ c2', (while true do skip end / st / log ⟶* c2' / st' / log')`

Proof. Let `st`, `log`, `st'`, and `log'` be given. Suppose we have `c1'` such that `H: (skip / st / log) ⟶* c1' / st' / log'`. We must show that `∃ c2', (while true do skip end) / st / log ⟶* c2' / st' / log'`.

We proceed by induction on the derivation of `H` via the `⟶*` relation, and there are two cases:

- In the base case of `multi_refl`, we have: `c1'` = (☒ `skip` *or* ☐ `while true do skip end`), and `st = st'` and `log = log'`. We can instantiate the existential by choosing `c2'` = (☐ `skip` *or* ☒ `while true do skip end`) and we conclude by observing that in zero steps: `c2 / st / log ⟶* c2' / st / log`
- In the second case, we have that there exists `c1''`, `st1''` and `log''` such that `H1 : skip / st / log ⟶c1'' / st'' / log''` and that `H2 : c1'' / st'' / log '' ⟶* c1' / st' / log'` but we can conclude by observing (choose one):
  ☐ that `H2` directly yields the result thanks to the induction hypothesis
  ☒ that `H1` contradicts the definition of `step`, since `skip` does not step, so this case is vacuously true.

(d) (4 points) Proving the other direction needed to show that `skip` and `while true do skip end` are equated by `cequiv_second_try` is quite challenging. The statement of the needed property is:

Lemma Undesired Equivalence Right-to-Left: `∀ st log st' log',`

`∀ c2', (while true do skip end / st / log ⟶* c2' / st' / log') →`

`∃ c1', (skip / st / log ⟶* c1' / st' / log')`

To prove this, a very helpful lemma is:

Lemma No Outputs `∀ st st' c`
`while true do skip end / st / log ⟶* c / st' / log' → st = st' ∧ log = log'.`

In the space below, give a *short* informal proof of the **Right-to-Left** lemma, you should explicitly mention how you use the **No Outputs** lemma.

Proof. From the assumptions and the No Outputs lemma, we have that `st = st'` and `log = log'`. We need to choose `c1'` such that `skip / st / log ⟶* c1' / st / log`, but that follows by picking `c1' = skip` and using `multi_refl`.

3. **Hoare Logic** (18 points total)

Appendix A contains a summary of the standard Hoare Logic rules for Imp.

Adding assertions for proving properties about programs

(a) (8 points) Prove that, beginning in a state where if `X` is `m`, `Y` is `n`, and some suitable condition `P` holds, the following program *swaps* the value of two variables. Choose the (correct) condition `P` from the list below and then fill in correct assertions to complete the decorated Hoare proof. (Hint: recall that Imp semantics implement arithmetic over natural numbers, so, e.g., 2 - 3 = 0.)

☐ `P` is `True`

☒ `P` is `m <= n`

☐ `P` is `n <= m`

```
    {{ X = m ∧ Y = n ∧   P                                       }} ->>
    {{ (Y - X) + (Y - (Y - X)) = n  ∧ (Y - (Y - X)) = m
                                                                  }}
  X := Y - X
    {{X + (Y - X) = n ∧ (Y - X) = m
                                                                  }}
  Y := Y - X
    {{ X + Y = n ∧ Y = m
                                                                  }}
  X := X + Y
    {{ X = n ∧ Y = m }}
```

(b) (10 points) Complete the Hoare-logic decorations to prove that the following program will always assign 1 to X.

```
    {{ True }}
  X := 0
    {{ X = 0                          }}
  if X = 0 then
    {{ X = 0 ∧ X = 0                 }} ->>
    {{ 1 = 1                          }}
    X := 1
    {{ X = 1                          }}
  else
    {{ X = 0 ∧ ~(X = 0 )             }} ->>
    {{ 2 = 1                          }}
    X := 2
    {{ X = 1                          }}
  end
    {{ X = 1                          }} ->>
    {{ X = 1 }}
```

4. **Loop Invariants** (15 points total)

Recall that an assertion `P` is a valid loop invariant for `while b do c end` if `{{ P ∧ b }} c {{ P }}` holds.

(a) (5 points) Mark which of the following are valid loop invariants for the following program. (Choose all that apply)

```
while X > 0 do
    X := X - 1
end
```

☒ `True`

☒ `False`

☐ `X = 1`

☐ `X > 0`

☒ `X ≥ 0`

(b) (5 points) Mark which of the following are valid loop invariants AND can be used to prove the following Hoare triple. (Choose all that apply)

```
{{ even n ∧ even m ∧ n > m}}
X := n
Y := m
while X > Y do
    X := X - 1
    Y := Y + 1
end
{{ X * 2 = n + m }}
```

☐ `True`

☐ `X + Y = n + m`

☐ `X * 2 = X + Y`

☐ `(X ≥ Y) ∧ X * 2 = X + Y`

☒ `(X ≥ Y) ∧ X + Y = n + m`

(c) (5 points) Mark which of the following are loop invariants (for the outer loop) AND can be used to prove the following Hoare triple. Recall that `exp 2 n` calculates 2^n. (Choose all that apply)

```
{{ exp 2 k = n }}
X := n
Z := 0
while X > 1 do
    Y := 0
    while Y < X do
        X := X - 1
        Y := Y + 1
    end
    Z := Z + 1
end
{{ exp 2 Z = n }}
```

☐ `True`

☐ `False`

☐ `(exp 2 Z) * X = n`

☒ `((X ≥ 1) ∧ (exp 2 Z) * X = n)`

☐ `((n ≥ 1) ∧ (exp 2 Z) * X = n)`

5. **Imp Semantics** (11 points)

Suppose that we extend Imp with a new command, `swap X Y`, that *atomically swaps* the contents of the two variables `X` and `Y`. For example, after running the following program, the value of `X` will be `5000` and `Y` will be `17`

```
X := 17;
Y := 5000;
swap X Y;
```

(a) (3 points) To define the semantics for this new command, we need to add a new rule to the `ceval` predicate.

```
Inductive ceval : com → state → state → Prop :=
  | E_Skip : ∀ st,
      st =[ skip ]⇒ st

  | E_Asgn  : ∀ st a n x,
      aeval st a = n →
      st =[ x := a ]⇒ (x !→ n ; st)

  | (* other cases are standard and omitted *)

  ???.
```

Recall that `x !→ v ; m` updates the map `m` so that identifier `x` maps to `v`. Which of the following clauses should be placed in the hole `???` above to properly define the semantics for swap? (Choose one)

☐
```
  | E_Swap : ∀ st x y,
      (x !→ st y; st) =[ swap x y ]⇒ (y !→ st x ; st)
```

☒
```
  | E_Swap : ∀ st x y,
      st =[ swap x y ]⇒ (x !→ st y ; y !→ st x ; st)
```

☐
```
  | E_Swap : ∀ st x y,
      (x !→ st y ; y !→ st x ; st) =[ swap x y ]⇒ st
```

☐
```
  | E_Swap : ∀ st x y n m,
      st =[ swap x y ]⇒ (x !→ n ; y !→ m ; st)
```

(b) (8 points) Assuming that the `swap X Y` instruction's semantics are implemented correctly, which of the following are valid statements about Hoare triples? (Mark all that are correct.)

☒ `∀m,n, {{ X = m /\ Y = n }} swap X Y {{ X = n /\ Y = m }}`

☒ `{{ fun st => Q [X |-> st Y] [Y |-> st X] st }} swap X Y {{ Q }}`

☒ `{{ P }} swap X Y {{ fun st => P [X |-> st Y][Y |-> st X] st }}`

☒ `{{ X = Y }} swap X Y {{ Y = X }}`

6. **Small Step Semantics** (20 points total)

In this problem, we will work with a simple language that only has `Push` and `Pop` instructions. These instructions will be used to manipulate a `stack`, which is just a list of natural numbers.

```
Inductive instr : Type :=
| Push : nat → instr
| Pop : nat → instr.

Definition stack := list nat.
```

The evaluation function `eval` below gives a precise specification of the behavior.

```
Fixpoint eval (prog : list instr) (st : stack) : option stack :=
    match prog with
    | [] ⇒ Some st
    | i :: rest ⇒
      match i with
      | Push n ⇒ eval rest (n :: st)
      | Pop n ⇒
        match st with
        | [] ⇒ None
        | n' :: st' ⇒ if n =? n' then eval rest st' else None
        end
      end
    end.
```

For example,

```
eval [Push 1; Pop 1; Push 2] [] = Some [2].
```

The `eval` function returns an `option` because evaluation can fail. Specifically, this happens when we try to `Pop` a value from the top of the stack that is not there. For example,

```
eval [Pop 2] [1] = None.
```

(a) (8 points) Complete this inductively defined relation for the small-step semantics so that it captures the same behavior as `eval`. For example, the following should be provable using your relation:

```
step ([Push 1; Pop 1; Push 2] , []) ([Pop 1; Push 2] , [1]).
```

On the other hand, these should not be provable for any `(st : stack)` or `(next : list instr * stack)`:

```
step ([] , st) next

step ([Pop 2] , [1]) next
```

```
Inductive step : (list instr * stack) → (list instr * stack) → Prop :=
| SPush : ∀ (n : nat) (l : list instr) (st : stack),
          step (Push n :: l, st) (l, n :: st)
| SPop : ∀ (n : nat) (l : list instr) (st : stack),
         step (Pop n :: l, n :: st) (l, st).
```

(b) (12 points) *The answers to this problem will be graded relative to a **correct definition** of the* `step` *relation (based on the provided specification) and **not** relative to your answer to part (a). As a result, you should be able to reason about this problem regardless of whether you solved part (a).*

Consider these two new `step` rules. First,

```
| SIgnore : ∀ (i : instr) (l : list instr) (st : stack),
            step (i :: l, st) (l, st)
```

This allows us to non-deterministically ignore an instruction, so we could prove things like

```
step ([Push 1] , []) ([] , []).
```

Second,

```
| SRepeat : ∀ (i : instr) (l : list instr) (st st' : stack),
            step (i :: l, st) (l, st') →
            step (i :: l, st) (i :: l, st').
```

This allows us to non-deterministically repeat an instruction, so we could prove things like

```
step ([Push 1] , []) ([Push 1] , [1]).
```

We define our values for this language as follows:

```
Inductive value : (list instr * stack) → Prop :=
| v_empty : ∀ st, value ([], st).
```

Which completions make true statements? Please consult Appendix B for the definitions, which are taken from the `Smallstep.v` file. (Mark all that apply, 1 point for each box.)

i. `normalizing` is provable...
   - ☒ for the original `step`
   - ☒ for `step + SIgnore`
   - ☒ for `step + SRepeat`
   - ☒ for `step + SIgnore + SRepeat`

ii. `value_is_nf` is provable...
   - ☒ for the original `step`
   - ☒ for `step + SIgnore`
   - ☒ for `step + SRepeat`
   - ☒ for `step + SIgnore + SRepeat`

iii. `nf_is_value` is provable...
   - ☐ for the original `step`
   - ☒ for `step + SIgnore`
   - ☐ for `step + SRepeat`
   - ☒ for `step + SIgnore + SRepeat`

7. **Types, Preservation, Progress** (12 points total)

This problem refers to the combined language of `Bools` and `Nats` from the `Types.v` file. For your reference, this language defined in Appendix C, along with the statements of the various lemmas used below.

(a) (6 points) Suppose we add this new typing rule to the system (keeping all the others unchanged):

```
-----------------              (TrueNat_added)
⊢  true ∈ Nat
```

Which of the following properties no longer hold for this language? (Mark all that apply; if they all remain true, mark that box instead.)

☐ `bool_canonical`

☒ `nat_canonical`

☒ `progress`

☐ `preservation`

☐ `step_deterministic`

☐ *They all remain true*

(b) (6 points) Suppose we instead add this new stepping rule to the system (keeping all the others unchanged):

```
------------------                    (ST_Pred0_added)
⊢  pred 0 ⟶ true
```

Which of the following properties no longer hold for this language? (Mark all that apply; if they all remain true, mark that box instead.)

☐ `bool_canonical`

☐ `nat_canonical`

☐ `progress`

☒ `preservation`

☒ `step_deterministic`

☐ *They all remain true*

CIS 5000 2022 Midterm 2 Appendices

(Do not write answers in the appendices. They will not be graded)

Appendix A: Imp Semantics and Hoare Logic Rules

Imp Large Step Semantics

```
                -----------------                            (E_Skip)
                st =[ skip ]⇒ st

                aeval st a = n
        ------------------------------                       (E_Asgn)
        st =[ x := a ]⇒ (x !→ n ; st)

                st  =[ c1 ]⇒ st'
                st' =[ c2 ]⇒ st''
              ---------------------                          (E_Seq)
              st =[ c1;c2 ]⇒ st''

               beval st b = true
                st =[ c1 ]⇒ st'
     --------------------------------------                 (E_IfTrue)
     st =[ if b then c1 else c2 end ]⇒ st'

              beval st b = false
                st =[ c2 ]⇒ st'
     --------------------------------------                 (E_IfFalse)
     st =[ if b then c1 else c2 end ]⇒ st'

              beval st b = false
         -----------------------------                      (E_WhileFalse)
         st =[ while b do c end ]⇒ st

               beval st b = true
                st =[ c ]⇒ st'
       st' =[ while b do c end ]⇒ st''
       --------------------------------                     (E_WhileTrue)
       st  =[ while b do c end ]⇒ st''
```

Imp Hoare Logic Rules

```
---------------------------  (hoare_asgn)
{{Q [X ⊢> a]}} X:=a {{Q}}

--------------------  (hoare_skip)
{{ P }} skip {{ P }}

  {{ P }} c1 {{ Q }}
  {{ Q }} c2 {{ R }}
 ----------------------  (hoare_seq)
 {{ P }} c1;c2 {{ R }}

        {{P ∧   b}} c1 {{Q}}
        {{P ∧ ∼ b}} c2 {{Q}}
------------------------------------  (hoare_if)
{{P}} if b then c1 else c2 end {{Q}}

        {{P ∧ b}} c {{P}}
  ----------------------------------  (hoare_while)
  {{P}} while b do c end {{P ∧ ∼ b}}

        {{P'}} c {{Q'}}
           P →> P'
           Q' →> Q
   -----------------------------   (hoare_consequence)
         {{P}} c {{Q}}
```

Appendix B: Normal Form and Value Definitions

Given `value` and `step` relations defined elsewhere, we have the following additional definitions.

```
Definition normal_form (t : list instr * stack) : Prop :=
    ~ ∃ t', step t t'.

Lemma normalizing : ∀ t,
    ∃ t', (multi step) t t' ∧ normal_form t'.

Lemma value_is_nf : ∀ v,
    value v → normal_form v.

Lemma nf_is_value : ∀ nf,
    normal_form nf → value nf.

Inductive multi {X : Type} (R : relation X) : relation X :=
  | multi_refl : ∀ (x : X), multi R x x
  | multi_step : ∀ (x y z : X),
                    R x y →
                    multi R y z →
                    multi R x z.
```

Appendix C: Combined Bool and Nat terms

This Appendix has the definitions of the terms, small-step semantics, and typing rules for the language from `Types.v`.

Syntax and Values

```
t ::= true                     bvalue v ↔ (v = true ∨ v = false)
    | false
    | if t then t else t       nvalue n ↔ (n = 0 ∨ (∃ m, n = succ m ∧ nvalue m)
    | 0
    | succ t
    | pred t
    | iszero t
```

Small-step operational semantics

```
                              ------------------------------                  (ST_IfTrue)
                              if true then t1 else t2 ⟶ t1

                              ------------------------------                  (ST_IfFalse)
                              if false then t1 else t2 ⟶ t2

                                         t1 ⟶ t1'
                       ---------------------------------------------------    (ST_If)
                       if t1 then t2 else t3 ⟶ if t1' then t2 else t3

                                       t1 ⟶ t1'
                                    --------------------                      (ST_Succ)
                                    succ t1 ⟶ succ t1'

                                      ------------                            (ST_Pred0)
                                      pred 0 ⟶ 0

                                    numeric value v
                                   -------------------                        (ST_PredSucc)
                                   pred (succ v) ⟶ v

                                        t1 ⟶ t1'
                                    --------------------                      (ST_Pred)
                                    pred t1 ⟶ pred t1'

                                     -----------------                        (ST_IsZero0)
                                     iszero 0 ⟶ true

                                    numeric value v
                                 -------------------------                    (ST_IszeroSucc)
                                 iszero (succ v) ⟶ false

                                      t1 ⟶ t1'
                                  ------------------------                    (ST_Iszero)
                                  iszero t1 ⟶ iszero t1'
```

Type System

```
                ---------------                         (T_True)
                ⊢ true ∈ Bool

               ---------------                          (T_False)
               ⊢ false ∈ Bool

⊢ t1 ∈ Bool     ⊢ t2 ∈ T     ⊢ t3 ∈ T
--------------------------------------------           (T_If)
       ⊢ if t1 then t2 else t3 ∈ T

                  --------------                        (T_0)
                  ⊢ 0 ∈ Nat

                 ⊢ t1 ∈ Nat
               -----------------                        (T_Succ)
               ⊢ succ t1 ∈ Nat

                 ⊢ t1 ∈ Nat
               -----------------                        (T_Pred)
               ⊢ pred t1 ∈ Nat

                 ⊢ t1 ∈ Nat
             --------------------                     (T_Iszero)
               ⊢  iszero t1 ∈ Bool
```

Key Lemmas

```
Lemma bool_canonical : ∀ t,
  ⊢ t ∈ Bool → value t → bvalue t.

Lemma nat_canonical : ∀ t,
  ⊢ t ∈ Nat → value t → nvalue t.

Theorem progress : ∀ t T,
  ⊢ t ∈ T →
  value t ∨ ∃ t', t ⟶ t'.

Theorem preservation : ∀ t t' T,
  ⊢ t ∈ T →
  t ⟶ t' →
  ⊢ t' ∈ T.

Definition deterministic {X : Type} (R : relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.

Theorem step_deterministic:
  deterministic step.
```