CMSC 631: Midterm Exam (Fall 2020)

1 Question 1 (20 points)

For each of the types below, write a Coq expression that has that type or write “empty” if there are no such expressions.

(a) `bool → list bool`

(b) `forall (X : Type), option nat`

(c) `forall (X Y : Type), X`

(d) `forall (X Y : Type), X → Y`

(e) `Prop * Prop`

(f) `option (Prop * Prop -> Prop)`

(g) `forall (X : Type), list X`

(h) `forall (P Q : Prop), P ∧ Q`

(i) `forall (P Q : Prop), P → Q → P ∧ Q`

2 Question 2 (20 points)

A *binary tree* with natural numbers as labels is either *empty* or a *node* that contains some natural number along with two binary trees as children. Formally, binary trees can be defined as follows in Coq:

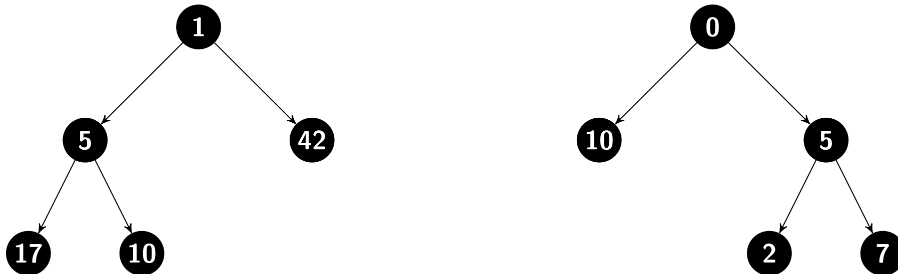
```
Inductive tree :=
| Empty : tree
| Node : nat -> tree -> tree -> tree.
```

For example, the following definitions:

```
Definition ex_tree_1 : tree :=
  Node 1
       (Node 5
             (Node 17  Empty Empty)
             (Node 10 Empty Empty))
       (Node 42 Empty Empty).
```

```
Definition ex_tree_2 : tree :=
  Node 0
       (Node 10 Empty Empty)
       (Node 5
             (Node 2 Empty Empty)
             (Node 7 Empty Empty)).
```

represent the trees:



(a) A *heap* is a binary tree that satisfies the heap property: in a heap, for any given node C, if P is a parent node of C, then the key (the value) of P is less than or equal to the key of C. The node at the "top" of the heap (with no parents) is called the root node.

For example, `ex_tree_1` satisfies the heap property, but `ex_tree_2` does not: in the right subtree, the label of the parent (5) is greater than the label of the child (2).

Fill in the following inductive definition that captures what it means for a tree to satisfy the heap property. It should be the case that the following `Example`s are provable.

Hint: You might find it easier to define another auxiliary inductive definition that takes an additional parameter - denoting the value of the parent of a tree. You're welcome to do so or not to (it's definitely possible without)!

```
(* Satisfies definition: *)
Example heap1 : heap_wf ex_tree_1.
(* Trivially satisfies definition: *)
Example heap2 : heap_wf Empty.
(* The right subtree of ex_tree_2 does not satisfy the heap property: *)
Example not_heap_1 : ~ (heap_wf (Node 5 (Node 2 Empty Empty)
                                        (Node 7 Empty Empty))).
(* Therefore ex_tree_2 is also not a valid heap: *)
Example not_heap_2 : ~ (heap_wf ex_tree_2).
```

```
Inductive heap_wf : tree -> Prop :=
```

(b) Assume we have the following `pop` function that operate on heaps. It returns the root of the heap along with a new tree that contains all the remaining elements of the heap while preserve the heap property. It satisfies the following examples:

```
pop  : tree -> option (nat * tree)

(* Popping an empty tree returns None *)
Example pop_empty : pop Empty = None.

(* Popping a singleton tree: *)
Example pop_single : pop (Node 42 Empty Empty) = Some (42, Empty).

(* Popping ex_tree_1: *)
Example pop_ex :
  pop (Node 1
         (Node 5
            (Node 17  Empty Empty)
            (Node 10 Empty Empty))
         (Node 42 Empty Empty)) =
  Some (1, Node 5
             (Node 10 (Node 17 Empty Empty) Empty)
             (Node 42 Empty Empty)).
```

Fill in the following function `heapsort` which takes a natural number `n` and a heap `t`, and returns the smallest `n` elements of the heap in ascending order.

Use `pop`!

For example, all the following `Example`s should hold:

```
(* Sorting an empty tree. *)
Example sort_empty : forall n, heapsort n Empty = [].

(* Sorting the example with enough fuel: *)
Example sort_ex_1 : heapsort 10 ex_tree_1 = [1;5;10;17;42].

(* Sorting the example with less fuel than the list length: *)
Example sort_ex_2 : heapsort 3 ex_tree_1 = [1;5;10].

Fixpoint heapsort (n : nat) (t : tree) : list nat :=
```

3 Question 3 (20 points)

(a) Consider the following IMP program:

```
Y := 0;
WHILE X < 42 DO
  X := X + 1;
  Y := Y + 1
DONE
```

Fill in the annotations in the following program to show that the Hoare triple given by the outermost pre- and post- conditions is valid. Be completely precise and pedantic in the way you apply Hoare rules—write assertions in *exactly* the form given by the rules rather than just 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 `->>`. These implication steps can (silently) rely on all the usual rules of **natural** number arithmetic.

```
{{ m < 42 }} ->>
{{                                                  }}
  X := m
{{                                                  }}
  Y := 0;
{{                                                  }}
  WHILE X < 42 DO
{{                                                  }} ->>
{{                                                  }}
    X := X + 1;
{{                                                  }}
    Y := Y + 1
{{                                                  }}
DONE
{{                                                  }} ->>
{{ X = 42 /\ Y = 42 - m }}
```

(b) Consider the following pairs of IMP programs. For each pair, write "Equivalent" if they are equivalent according to `cequiv`, or provide a counterexample (an initial state for which the resulting states are different):

i)

```
Y := 0;
WHILE X < 42 DO
  X := X + 1;
  Y := Y + 1
DONE
```

```
Y := 42 - X
```

Answer:

ii)

```
WHILE X < 42 DO
  X := X + 1;
DONE
```

```
X := 42
```

Answer:

iii)

```
X := 0;
WHILE X < 42 DO
  X := X + 1;
DONE
```

```
IF Y < 17 THEN
  X := 42
ELSE
  X := 42
ENDIF
```

Answer:

iv)

```
IF X = Y THEN
  Z := 0
ELSE
  Z := X - Y
ENDIF
```

```
Z := X - Y
```

Answer:

4 Question 4 (20 points)

In the rest of this problem, we will add a NONZERO command to IMP: a conditional that checks whether an arithmetic expression is zero or not and then only executes the nested command if it is not. Here is the syntax of this extension:

```
Inductive com : Type :=
  | CSkip
  | CAss (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com)
  | CNonZero (a : bexp) (c : com) (* ← new *).
```

We will use an WHEN-NONZERO-ENDWHEN notation where, given a command `CNonZero a c`, we will write:

```
WHEN a NONZERO
  c
ENDWHEN
```

The expected behavior of the new construct should be such that the two following programs are equivalent accord to `cequiv`:

```
IF a = 0 THEN
  c
ELSE
  skip
ENDIF
```

```
WHEN a NONZERO
  c
ENDWHEN
```

(a) Write a Hoare rule for this new construct. It should be strong enough to prove the following triple:

```
{{ X - Y = Z }}
WHEN X - Y NONZERO
  Z := 0
ENDON
{{ Z = 0 }}
```

(b) Extend the evaluation relation for IMP to account for this new construct.

```
Reserved Notation "c1 '/' st '\\' st'" (at level 40, st at level 39).

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 \\ 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' b c1 c2,
      beval st b = true ->
      c1 / st \\ st' ->
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_IfFalse : forall st st' b c1 c2,
      beval st b = false ->
      c2 / st \\ st' ->
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_WhileFalse : forall b st c,
      beval st b = false ->
      (WHILE b DO c END) / st \\ st
  | E_WhileTrue : forall st st' st'' b c,
      beval st b = true ->
      c / st \\ st' ->
      (WHILE b DO c END) / st' \\ st'' ->
      (WHILE b DO c END) / st \\ st''

  (* FILL IN HERE *)

  where "c1 '/' st '\\' st'" := (ceval c1 st st').
```

5 Bonus Questions (2 points)

(a) How long did this exam take you? (1 point, no matter what you answer)

(b) Write a haiku about this class. (0.5 point if funny, 0.5 point if actually a haiku).
A haiku is a short poem that follows a 5-7-5 syllable pattern. Example:

Zoom lectures not fun.
I'd rather be in person.
Well, maybe next year.

Library Reference

A Logic

```
Inductive and (X Y : Prop) : Prop :=
  conj : X → Y → and X Y.

Inductive or (X Y : Prop) : Prop :=
 | or_introl : X → or X Y
 | or_intror : Y → or X Y.

Arguments conj {X Y}.
Arguments or_introl {X Y}.
Arguments or_intror {X Y}.

Notation "A ∧ B" := (and A B).
Notation "A ∨ B" := (or A B).

Definition iff (A B : Prop) := (A → B) ∧ (B → A).
Notation "A ↔ B" := (iff A B) (at level 95).
```

B Booleans

```
Inductive bool : Type :=
  | true
  | false.

Definition negb (b:bool) : bool :=
  match b with
  | true ⇒ false
  | false ⇒ true
  end.

Definition andb (b1 b2: bool) : bool :=
  match b1 with
  | true ⇒ b2
  | false ⇒ false
  end.

Definition orb (b1 b2:bool) : bool :=
  match b1 with
  | true ⇒ true
  | false ⇒ b2
  end.
```

C Numbers

```
Inductive nat : Type :=
  | O
  | S (n : nat).

Fixpoint plus (n : nat) (m : nat) : nat :=
  match n with
    | O ⇒ m
    | S n' ⇒ S (plus n' m)
  end.
```

```
Fixpoint minus (n m:nat) : nat :=
  match n, m with
  | O   , _    ⇒ O
  | S _ , O    ⇒ n
  | S n', S m' ⇒ minus n' m'
  end.

Fixpoint mult (n m : nat) : nat :=
  match n with
    | O ⇒ O
    | S n' ⇒ plus m (mult n' m)
  end.

Notation "x + y" := (plus x y) (at level 50, left associativity).
Notation "x - y" := (minus x y) (at level 50, left associativity).
Notation "x * y" := (mult x y) (at level 40, left associativity).

Fixpoint eqb (n m : nat) : bool :=
  match n with
  | O ⇒ match m with
        | O ⇒ true
        | S m' ⇒ false
        end
  | S n' ⇒ match m with
           | O ⇒ false
           | S m' ⇒ eqb n' m'
           end
  end.

Fixpoint leb (n m : nat) : bool :=
  match n with
  | O ⇒ true
  | S n' ⇒
      match m with
      | O ⇒ false
      | S m' ⇒ leb n' m'
      end
  end.

Notation "x =? y" := (eqb x y) (at level 70).
Notation "x ≤? y" := (leb x y) (at level 70).

Inductive le : nat → nat → Prop :=
  | le_n n : le n n
  | le_S n m : le n m → le n (S m).

Notation "m ≤ n" := (le m n).
```

D Lists

```
Inductive list (X:Type) : Type :=
  | nil
  | cons (x : X) (l : list X).

Arguments nil {X}.
Arguments cons {X} _ _.

Notation "x :: y" := (cons x y) (at level 60, right associativity).
Notation "[ ]" := nil.
Notation "[ x ; .. ; y ]" := (cons x .. (cons y []) ..).
Notation "x ++ y" := (app x y)  (at level 60, right associativity).

Fixpoint map {X Y: Type} (f : X → Y) (l : list X) : (list Y) :=
  match l with
  | []     ⇒ []
  | h :: t ⇒ (f h) :: (map f t)
  end.

Fixpoint filter {X : Type} (test : X → bool) (l : list X)
                : (list X) :=
  match l with
  | []     ⇒ []
  | h :: t ⇒ if test h then h :: (filter test t)
                        else       filter test t
  end.

Fixpoint fold {X Y} (f : X → Y → Y) (l : list X) (b : Y) : Y :=
  match l with
  | nil ⇒ b
  | h :: t ⇒ f h (fold f t b)
  end.
```

E Strings

We won't define strings from scratch here. Assume `eqb_string` has the type given below, and anything within quotes is a string.

```
Parameter eqb_string : string → string → bool.
```

F Maps

```
Definition total_map (A:Type) := string → A.

Definition t_empty {A:Type} (v : A) : total_map A :=
  (fun _ ⇒ v).

Definition t_update {A:Type} (m : total_map A) (x : string) (v : A) :=
  fun x' ⇒ if eqb_string x x' then v else m x'.

Notation "{ --> d }" := (t_empty d) (at level 0).
Notation "m '&' { a --> x }" := (t_update m a x) (at level 20).
```

G Imp

```
Inductive aexp : Type :=
  | ANum (n : nat)
  | AId (x :  string)
  | APlus (a1 a2 : aexp)
  | AMinus (a1 a2 : aexp)
  | AMult (a1 a2 : aexp).

Inductive bexp : Type :=
  | BTrue
  | BFalse
  | BEq (a1 a2 : aexp)
  | BLe (a1 a2 : aexp)
  | BNot (b : bexp)
  | BAnd (b1 b2 : bexp).

Inductive com : Type :=
  | CSkip
  | CAss (x : string) (a : aexp)
  | CSeq (c1 c2 : com)
  | CIf (b : bexp) (c1 c2 : com)
  | CWhile (b : bexp) (c : com).

Notation "'SKIP'" := CSkip.
Notation "x '::=' a" := (CAss x 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 "'TEST' c1 'THEN' c2 'ELSE' c3 'FI'" := (CIf c1 c2 c3) (at level 80, right associativity).

Definition state := total_map nat.

Fixpoint aeval (st : state) (a : aexp) : nat :=
  match a with
  | ANum n ⇒ n
  | AId x ⇒ st x
  | APlus a1 a2 ⇒ (aeval st a1) + (aeval st a2)
  | AMinus a1 a2  ⇒ minus (aeval st a1) (aeval st a2)
  | AMult a1 a2 ⇒ (aeval st a1) * (aeval st a2)
  end.

Fixpoint beval (st : state) (b : bexp) : bool :=
  match b with
  | BTrue       ⇒ true
  | BFalse      ⇒ false
  | BEq a1 a2   ⇒ (aeval st a1) =? (aeval st a2)
  | BLe a1 a2   ⇒ (aeval st a1) ≤? (aeval st a2)
  | BNot b1     ⇒ negb (beval st b1)
  | BAnd b1 b2  ⇒ andb (beval st b1) (beval st b2)
  end.
```

```
Reserved Notation "c1 '/' st '\\' st'"
                  (at level 40, st at level 39).

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 \\ 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' b c1 c2,
      beval st b = true →
      c1 / st \\ st' →
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_IfFalse : forall st st' b c1 c2,
      beval st b = false →
      c2 / st \\ st' →
      (IFB b THEN c1 ELSE c2 FI) / st \\ st'
  | E_WhileFalse : forall b st c,
      beval st b = false →
      (WHILE b DO c END) / st \\ st
  | E_WhileTrue : forall st st' st'' b c,
      beval st b = true →
      c / st \\ st' →
      (WHILE b DO c END) / st' \\ st'' →
      (WHILE b DO c END) / st \\ st''

  where "c1 '/' st '\\' st'" := (ceval c1 st st').

Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st' : state),
  (c1 / st \\ st') ↔ (c2 / st \\ st').
```