CMSC 631: Midterm Exam (Fall 2023)

1 Question 1 (10 points)

For each of the terms below, write their type or "ill-typed" if an expression doesn't have a type.

(a) `[bool]`

(b) `[nat; true]`

(c) `fun (x:X) ⇒ conj x x`

(d) `fun (x : nat → nat) y ⇒ x y`

(e) `Some (fun y ⇒ y + y)`

2 Question 2 (20 points)

For each of the types below, write a (nontrivial if possible) Coq expression that has that type or write "empty" if there are no such expressions.

(a) `bool → list nat`

(b) `forall (X Y : Prop), unit`

(c) `forall (X Y : Type), X → Y`

(d) `forall (X Y : Type), Y → Y`

(e) `nat * Prop`

(f) `list (bool * bool -> bool)`

(g) `forall (X : Prop), option X`

(h) `forall (x y : nat), x = y`

(i) `forall (x y : nat), x = x ∧ y = y`

3 Question 3 (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:

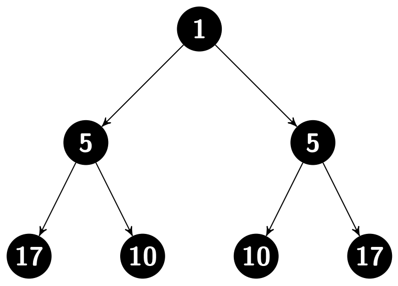
```
Inductive tree :=
| Empty : tree
| Node : nat -> tree -> tree -> tree.
```

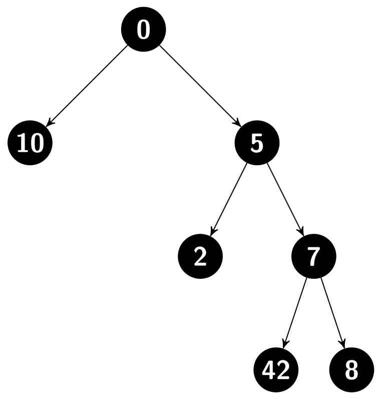
For example, the following definitions:

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

```
Definition ex_tree_2 : tree :=
  Node 0
       (Node 10 Empty Empty)
       (Node 5
             (Node 2 Empty Empty)
             (Node 7
                   (Node 42 Empty Empty)
                   (Node 8  Empty Empty))).
```

represent the trees:





(a) A binary tree is *symmetric* if its left and right subtrees are mirror images of each other. For example, ex_tree_1 is a symmetric tree, while ex_tree_2 is not.

Fill in the following function that calculates whether a given binary tree is symmetric. You are free to write any helper function you want.

```
(* Trivial tree: *)
Example sym_Empty : is_sym Empty = true.
(* Examples *)
Example sym_1 : is_sym ex_tree_1 = true.
Example sym_2 : is_sym ex_tree_2 = false.
```

```
Fixpoint is_sym (t: tree) : bool :=
```

(b) Write an *inductive relation* that holds when a tree is symmetric: do *not* use any functions you defined in the previous problem, but you are again allowed to write any additional inductive relations you want. The following `Example`s should be provable.

```
(* Trivial tree: *)
Example Sym_Empty : sym Empty.
(* Examples *)
Example Sym_1 : sym ex_tree_1.
Example Sym_2 : sym ex_tree_2.

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

4 Question 4 (25 points)

1. Fill in the following function that computes the nth power of 2 of a given natural number:

```
Fixpoint power_of_2 (n : nat) :=
```

```
Notation "'2^' x" := (power_of_2 x).
```

2. Now consider the following IMP program:

```
  X := m
  Y := 0;
  Z := 1;
  WHILE X > 0 DO
    X := X - 1;
    Y := Z + Y;
    Z := Z + Z;
  DONE
  Y := Y + 1
```

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.

```
{{ True }} ->>
{{                                                     }}
  X := m
{{                                                     }}
  Y := 0;
{{                                                     }}
  Z := 1;
{{                                                     }}
  WHILE X > 0 DO
{{                                                     }} ->>
{{                                                     }}
    X := X - 1;
{{                                                     }}
    Y := Z + Y;
{{                                                     }}
    Z := Z + Z;
{{                                                     }}
DONE
{{                                                     }} ->>
{{                                                     }}
Y := Y + 1
{{ Y = 2^m }}
```

5 Question 5 (15 points)

In the rest of this problem, we will add a `fixpoint` command to IMP: it takes an arithmetic variable and a command as an argument; it then executes the command. If the value of the variable changed, then the command executes again; otherwise, execution of the fixpoint ends there. 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)
  | CFixpoint (x : string) (c : com) (* ← new *).
```

As notation, instead of `(CFixpoin x c)` we will write

```
Fix x { c }
```

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

```
Y := 42                                     FIX X { Y := 42 }
```

```
WHILE TRUE DO
  X := X + 1                                FIX X { X := X + 1 }
DONE
```

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

```
Reserved Notation
         "st '=[' c ']=>' st'"
         (at level 40, c custom com at level 99,
          st constr, st' constr at next level).

Inductive ceval : com -> state -> state -> Prop :=
  | E_Skip : forall st,
      st =[ skip ]=> st
  | E_Asgn  : forall st a n x,
      aeval st a = n ->
      st =[ x := a ]=> (x !-> n ; st)
  | E_Seq : forall c1 c2 st st' st'',
      st  =[ c1 ]=> st'  ->
      st' =[ c2 ]=> st'' ->
      st  =[ c1 ; c2 ]=> st''
  | E_IfTrue : forall st st' b c1 c2,
      beval st b = true ->
      st =[ c1 ]=> st' ->
      st =[ if b then c1 else c2 end]=> st'
  | E_IfFalse : forall st st' b c1 c2,
      beval st b = false ->
      st =[ c2 ]=> st' ->
      st =[ if b then c1 else c2 end]=> st'
  | E_WhileFalse : forall b st c,
      beval st b = false ->
      st =[ while b do c end ]=> st
  | E_WhileTrue : forall st st' st'' b c,
      beval st b = true ->
      st  =[ c ]=> st' ->
      st' =[ while b do c end ]=> st'' ->
      st  =[ while b do c end ]=> st''

  (* FILL IN HERE *)









  where "st =[ c ]=> st'" := (ceval c st st').
```

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).

Inductive eq (A : Type) : (x : A) → Prop :=
| eq_refl : eq x x.
```

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 "'_' '!→' v" := (t_empty v).
Notation "x '!→' v ';' m" := (t_update m x v).
```

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).

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
         "st '=[' c ']⇒' st'"
         (at level 40, c custom com at level 99,
          st constr, st' constr at next level).

Inductive ceval : com → state → state → Prop :=
  | E_Skip : forall st,
      st =[ skip ]⇒ st
  | E_Asgn  : forall st a n x,
      aeval st a = n →
      st =[ x := a ]⇒ (x !→ n ; st)
  | E_Seq : forall c1 c2 st st' st'',
      st  =[ c1 ]⇒ st'  →
      st' =[ c2 ]⇒ st'' →
      st  =[ c1 ; c2 ]⇒ st''
  | E_IfTrue : forall st st' b c1 c2,
      beval st b = true →
      st =[ c1 ]⇒ st' →
      st =[ if b then c1 else c2 end]⇒ st'
  | E_IfFalse : forall st st' b c1 c2,
      beval st b = false →
      st =[ c2 ]⇒ st' →
      st =[ if b then c1 else c2 end]⇒ st'
  | E_WhileFalse : forall b st c,
      beval st b = false →
      st =[ while b do c end ]⇒ st
  | E_WhileTrue : forall st st' st'' b c,
      beval st b = true →
      st  =[ c ]⇒ st' →
      st' =[ while b do c end ]⇒ st'' →
      st  =[ while b do c end ]⇒ st''

  where "st =[ c ]⇒ st'" := (ceval c st st').

Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st' : state),
  (st =[ c1 ]⇒ st') ↔ (st =[ c2 ]⇒ st').
```

H Hoare Rules

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

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


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


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

         {{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}}
```