CMSC 631: Midterm Exam (Fall 2019)

1 Question 1 (20 points)

Write the type of each of the following Coq expressions (write "ill typed" if an expression does not have a type).

(a) `42 + 0`
[redacted]

(b) `42 + 0 = 17`
[redacted]

(c) `fun (x : nat) ⇒ x :: []`
[redacted]

(d) `fun (x : list nat) ⇒ x :: []`
[redacted]

(e) `fun (x : list nat) ⇒ x :: x`
[redacted]

(f) `and True`
[redacted]

(g) `forall (n : nat), n`
[redacted]

(h) `forall (n : nat), n ≤ n`
[redacted]

(i) `map (eqb 42)`
[redacted]

(j) `map or [True]`
[redacted]

2 Question 2 (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) `nat → list nat`

[redacted]

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

[redacted]

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

[redacted]

(d) `forall (X : Type), X → X`

[redacted]

(e) `list Prop`

[redacted]

(f) `list (Prop -> Prop)`

[redacted]

(g) `forall (X : Type), X → Prop`

[redacted]

(h) `forall (X : Type), Prop → X`

[redacted]

(i) `forall (X : Type), Prop → Prop`

[redacted]

(j) `forall (X Y: Type), X → Y → X * Y`

[redacted]

3 Question 3 (20 points)

(a) Suppose Coq's current goal state looks like this:

```
x, y : nat
b  : bool
H1 : (x,y) = (42, 17)
H2 : negb b = negb true
=======================
true = false
```

(i) If we give the command `discriminate`, what will happen?

Error
Nothing (no error, but no change to the state)
No more subgoals

(ii) If we give the command `injection H1`, what will happen?

Error
Nothing (no error, but no change to the state)
No more subgoals
Goal changes to `x = 42 -> y = 17 -> true = false`

(iii) If we give the command `injection H2`, what will happen?

Error
Nothing (no error, but no change to the state)
No more subgoals
Goal changes to `b = true -> true = false`

(b) Suppose Coq's current goal state looks like this:

```
m, n : nat
H1 : forall x, x * m = n * x
H2 : x = n \/ x = m
=======================
n * n = n * m
```

(i) If we give the command `destruct H2`, what will happen?

Error
Nothing (no error, but no change to the state)
No more subgoals
`H2` is replaced by two hypotheses `H : x = n` and `H0 : x = m`
Goal is replaced by two subgoals, one with hypothesis `H : x = n` and one with hypothesis `H : x = m`

(ii) If we give the command `rewrite H1`, what will happen?

Error
Nothing (no error, but no change to the state)
No more subgoals
Goal changes to `n * n = n * n`
Goal changes to `n * n = m * m`
Goal changes to `n * m = n * m`
Goal changes to `m * m = n * m`

4 Question 4 (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:

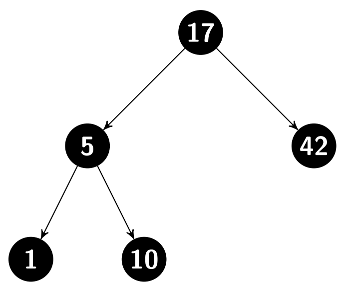
```
Inductive tree :=
| Empty : tree
| Node : nat -> tree -> tree -> tree.
```

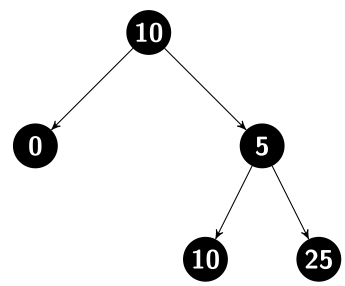
For example, the following definitions:

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

```
Definition ex_tree_2 : tree :=
  Node 10
       (Node 0 Empty Empty)
       (Node 5
             (Node 10 Empty Empty)
             (Node 25 Empty Empty)).
```

represent the trees:





(a) Fill in the following function `element_in` which takes a natural number `x` and a tree `t`, traverses the entire tree, and returns `true` if the value `x` appears in any of the nodes of `t`.

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

```
Example in_5_1  : element_in  5 ex_tree_1 = true.
Example in_42_1 : element_in 42 ex_tree_1 = true.
Example in_0_2  : element_in  0 ex_tree_2 = true.
Example not_in_0_1  : element_in  0 ex_tree_1 = false.
Example not_in_42_1 : element_in 42 ex_tree_2 = false.

Fixpoint element_in (x : nat) (t : tree) : bool :=
  [redacted]
```

A tree is called a *binary search tree* with elements between some lower bound `min` and some upper bound `max`, if for all nodes `(Node x l r)` in the tree, `x` is strictly greater than `min` and strictly smaller than `max`, and all values stored in the left subtree `l` are strictly less than the node's value `x`, and all values stored in the right subtree `r` are strictly greater than `x`.

For example, `ex_tree_1` is a binary search tree with elements between 0 and 50, but it wouldn't be one if we changed the label 10 to 1 or to 5. Similarly, `ex_tree_2` is *not* a binary search tree, not matter what choices we make for `min` and `max`.

(b) Fill in the following inductive definition that captures what it means for a tree to be such a binary search tree. It should be the case that the following `Example`s are provable:

```
(* Satisfies definition: *)
Example bst1 : bst 0 50  ex_tree_1.
(* Trivially satisfies definition: *)
Example bst2 : bst 0 50  Empty.
(* ex_tree_1 contains 42 that is not strictly smaller than 42: *)
Example not_bst_1 : ~ (bst 0 42 ex_tree_1).
(* The right child contains label 5 which not strictly greater
   than the value 10 of the top node. *)
Example not_bst_2 : ~ (bst 0 42 ex_tree_2).

Inductive bst : nat -> nat -> tree -> Prop :=
```

[redacted]

(c) Fill in the following function `element_in_bst` which takes a natural number `x` and a tree `t` *that is assumed to be a binary search tree* and returns `true` if the value `x` appears in any of the nodes of `t`. For example, for `ex_tree_1`, `element_in 5 ex_tree` and `element_in 42 ex_tree_1` should both return `true`, while `element_in 0 ex_tree_1` should not. You do *not* have to check whether `t` is a binary search tree. You should *not* traverse the entire tree.

```
Fixpoint element_in_bst (x : nat) (t : tree) : bool :=
```

[redacted]

5 Question 5 (20 points)

(a) Explain (briefly) why we can't write an evaluation function for the IMP language. (The IMP definition can be found in the reference appendix).

[redacted]

In the rest of this problem, we will replace the `WHILE` command with a different command: a for-loop with fixed bounds. The language now contains a new construct `FOR n c ENDFOR`, where `n` are natural numbers and `c` is an IMP command. Such a command simply evaluates `c` (the inner command) `n` times in a row. For example, if we run the following program, it will terminate in a state where `X` is mapped to 42.

```
X := 0;
FOR 42
  X := X + 1
ENDFOR
```

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)
  | CFor (n : nat) (c : com). (* ← new *)
```

(b) Can we write an evaluation function for commands in this version of IMP? Explain briefly. *You do not have to write this evaluation function.*

[redacted]

(c) 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'
  [redacted]t''.

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

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').
```