

Name: _____

Total: 100 points

Unless stated otherwise, only use definitions from `Basics.v` through `Poly.v` and any lemmas proved there. Rocq proofs must be written so that they would be accepted by Rocq; informal proofs must describe the proof not simply mimic a Rocq script. Both Rocq proofs and informal proofs must be documented. Specifically, identify any induction variables, proof cases, where an induction hypothesis is used, and transforms performed by rewriting and lemma application. This is far more important than perfect syntax.

1. Functional programming in Rocq (Basics)

- (a) (6 points) Without using `leb` or `eqb`, write a structurally recursive function

```
Fixpoint ltb (n m : nat) : bool := ...
```

that returns `true` exactly when $n < m$.

- (b) (8 points) Give an informal proof by case analysis of the following. Do not use any lemmas.

```
Theorem andb_true_elim2 : forall b c : bool,  
  andb b c = true -> c = true.
```

- (c) (6 points) The proof `intros n. reflexivity. succeeds` for `forall n : nat, 0 + n = n` but fails for `forall n : nat, n + 0 = n`. Explain why, referring to the definition of `plus`.

- (d) (5 points) Rocq rejects the following definition. State precisely why, and explain why Rocq imposes this restriction.

```
Fixpoint loop (n : nat) : nat :=  
  match n with  
  | 0 => 0  
  | S n' => loop (n' + 1)  
end.
```

2. Proof by induction (Induction)

- (a) (6 points) Give an informal proof of

$$\forall n : \mathbb{N}, \quad n \times 0 = 0.$$

Then explain in one sentence why case analysis on n alone, without induction, is not enough.

- (b) (7 points) Give a careful *informal* proof of

$$\forall n\ m : \mathbb{N}, \quad S(n + m) = n + S(m).$$

- (c) (7 points) Give an informal proof of

$$\forall n\ m : \mathbb{N}, \quad n + m = m + n.$$

You may use the lemmas $n + 0 = n$ (`add_0_r`) and $S(n + m) = n + S(m)$ (`plus_n_Sm`).

- (d) (5 points) Give an informal proof of

$$\forall n\ m\ p : \mathbb{N}, \quad (n + m) + p = n + (m + p).$$

3. Lists and options (Lists)

Use the `natlist` and `natoption` types from `Lists.v`, with the notations `[]`, `x :: l`, `[x; y]`, and `l1 ++ l2`.

- (a) (6 points) Define `alternate : natlist -> natlist -> natlist`, which interleaves two lists, starting with the first. When one list runs out, the rest of the other is appended. For example, `alternate [1;2;3] [4] = [1;4;2;3]`.

- (b) (5 points) Give an informal proof of

$$\text{Theorem app_nil_r : forall l : natlist, l ++ [] = l.}$$

- (c) (8 points) A naive proof by induction of `rev_involutive : forall l, rev (rev l) = l` gets stuck.

- i. State what must be shown in the `cons` case after unfolding the definition of `rev`, and explain why the induction hypothesis cannot be used directly.
- ii. State the auxiliary lemma about `rev` and `++` that is needed (you need not prove it).
- iii. Give an informal proof of `rev_involutive` using that lemma.

- (d) (6 points) `Lists.v` first defines `nth_bad`, which returns the default value 42 when the index is out of range, and then replaces it with `nth_error : natlist -> nat -> natoption`. Define `nth_error` and explain the problem it solves.

4. Polymorphism and higher-order functions (Poly)

Use the polymorphic `list`, `map`, `filter`, `rev`, `length`, and

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

(a) (6 points) Give the type Rocq reports for each expression:

- i. `@cons`
- ii. `@fold`
- iii. `map (fun x => x + 1)`
- iv. `fold andb`

(b) (5 points) Assume `Arguments nil {X}.` is in effect. Then the following command fails:

```
Fail Definition mynil := nil.
```

Explain why, and give two different ways to define an empty list of naturals.

(c) (7 points) Define `fold_length` and `fold_map` using `fold`, then give an informal proof of

```
Theorem fold_length_correct : forall X (l : list X),
  fold_length l = length l.
```

(d) (7 points) Give an informal proof of

```
Theorem map_rev : forall (X Y : Type) (f : X -> Y) (l : list X),
  map f (rev l) = rev (map f l).
```

You may state and use a lemma relating `map` and `++`. You need not prove this lemma.