SOFTWARE FOUNDATIONS

VOLUME 1: LOGICAL FOUNDATIONS

- thought their def of list was confusing a little: a list is either an empty list or else a pair of a number and another list. (n, xs)
- Really don't like the semicolon separator in lists!
- liked how proofs really depend on how you define your functions. this became clear in the proof of count and sum for bags. so if i do: if v==h then s(count v t) else count v t makes the proof harder than when i have: if v==h then 1 else 0 + cont v t
- not sure why my subset function isn't working if instead of removing an element at a time I remove all elements of a value all at the same time , which basically says that Cannot guess decreasing argument of fix.—> of course this is in the list part so we can go over it if time allows!
- for the app_assoc4 is there an easier way than applying rewrite of app_assoc twice? in general what if I want to apply a theorem as much as i can using just one command?
- interesting that you realize sometimes proofs are much easier than the trouble you go through! like count_member_nonzero
- really enjoyed remove_decrease_count when i figured out i can induct on list first and then destruct on the nat
-

```
Require Export Induction.
Module NatList.
```

Pairs of Numbers

In an `Inductive` type definition, each constructor can take any number of arguments — none (as with `true` and `O`), one (as with `S`), or more than one, as here:

```
Inductive natprod : Type :=
| pair : nat → nat → natprod.
```

This declaration can be read: "There is just one way to construct a pair of numbers: by applying the constructor `pair` to two arguments of type `nat`."

```
Check (pair 3 5).
```

Here are two simple functions for extracting the first and second components of a pair. The definitions also illustrate how to do pattern matching on two-argument constructors.

```
Definition fst (p : natprod) : nat :=
  match p with
  | pair x y ⇒ x
  end.

Definition snd (p : natprod) : nat :=
  match p with
  | pair x y ⇒ y
  end.

Compute (fst (pair 3 5)).
(* ===> 3 *)
```

Since pairs are used quite a bit, it is nice to be able to write them with the standard mathematical notation `(x,y)` instead of `pair x y`. We can tell Coq to allow this with a `Notation` declaration.

```
Notation "( x , y )" := (pair x y).
```

The new pair notation can be used both in expressions and in pattern matches (indeed, we've actually seen this already in the Basics chapter, in the definition of the `minus` function — this works because the pair notation is also provided as part of the standard library):

```
Compute (fst (3,5)).

Definition fst' (p : natprod) : nat :=
  match p with
  | (x,y) ⇒ x
  end.

Definition snd' (p : natprod) : nat :=
  match p with
  | (x,y) ⇒ y
  end.

Definition swap_pair (p : natprod) : natprod :=
  match p with
  | (x,y) ⇒ (y,x)
  end.
```

Let's try to prove a few simple facts about pairs.

If we state things in a particular (and slightly peculiar) way, we can complete proofs with just reflexivity (and its built-in simplification):

```
Theorem surjective_pairing' : ∀ (n m : nat),
  (n,m) = (fst (n,m), snd (n,m)).
Proof.
  reflexivity. Qed.
```

But `reflexivity` is not enough if we state the lemma in a more natural way:

```
Theorem surjective_pairing_stuck : ∀ (p : natprod),
  p = (fst p, snd p).
Proof.
  simpl. (* Doesn't reduce anything! *)
Abort.
```

We have to expose the structure of `p` so that `simpl` can perform the pattern match in `fst` and `snd`. We can do this with `destruct`.

```
Theorem surjective_pairing : ∀ (p : natprod),
  p = (fst p, snd p).
Proof.
  intros p. destruct p as [n m]. simpl. reflexivity. Qed.
```

Notice that, unlike its behavior with `nat`s, `destruct` generates just one subgoal here. That's because `natprod`s can only be constructed in one way.

Exercise: 1 star (snd_fst_is_swap)

```
Theorem snd_fst_is_swap : ∀ (p : natprod),
  (snd p, fst p) = swap_pair p.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 1 star, optional (fst_swap_is_snd)

```
Theorem fst_swap_is_snd : ∀ (p : natprod),
  fst (swap_pair p) = snd p.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Lists of Numbers

Generalizing the definition of pairs, we can describe the type of *lists* of numbers like this: "A list is either the empty list or else a pair of a number and another list."

```
Inductive natlist : Type :=
  | nil : natlist
  | cons : nat → natlist → natlist.
```

For example, here is a three-element list:

```
Definition mylist := cons 1 (cons 2 (cons 3 nil)).
```

As with pairs, it is more convenient to write lists in familiar programming notation. The following declarations allow us to use `::` as an infix `cons` operator and square brackets as an "outfix" notation for constructing lists.

don't like the semicolon separator!

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

It is not necessary to understand the details of these declarations, but in case you are interested, here is roughly what's going on. The `right associativity` annotation tells Coq how to parenthesize expressions involving several uses of `::` so that, for example, the next three declarations mean exactly the same thing:

```
Definition mylist1 := 1 :: (2 :: (3 :: nil)).
Definition mylist2 := 1 :: 2 :: 3 :: nil.
Definition mylist3 := [1;2;3].
```

The `at level 60` part tells Coq how to parenthesize expressions that involve both `::` and some other infix operator. For example, since we defined `+` as infix notation for the `plus` function at level 50,

lower the number
higher the precedecence

```
Notation "x + y" := (plus x y)
                    (at level 50, left associativity).
```

the + operator will bind tighter than `::`, so `1 + 2 :: [3]` will be parsed, as we'd expect, as `(1 + 2) :: [3]` rather than `1 + (2 :: [3])`.

(Expressions like "`1 + 2 :: [3]`" can be a little confusing when you read them in a `.v` file. The inner brackets, around 3, indicate a list, but the outer brackets, which are invisible in the HTML rendering, are there to instruct the "coqdoc" tool that the bracketed part should be displayed as Coq code rather than running text.)

The second and third `Notation` declarations above introduce the standard square-bracket notation for lists; the right-hand side of the third one illustrates Coq's syntax for declaring n-ary notations and translating them to nested sequences of binary constructors.

Repeat

A number of functions are useful for manipulating lists. For example, the `repeat` function takes a number `n` and a `count` and returns a list of length `count` where every element is `n`.

```
Fixpoint repeat (n count : nat) : natlist :=
  match count with
  | O ⇒ nil
  | S count' ⇒ n :: (repeat n count')
  end.
```

Length

The `length` function calculates the length of a list.

```
Fixpoint length (l:natlist) : nat :=
  match l with
  | nil ⇒ O
  | h :: t ⇒ S (length t)
  end.
```

Append

The `app` function concatenates (appends) two lists.

```
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil ⇒ l2
  | h :: t ⇒ h :: (app t l2)
  end.
```

Actually, `app` will be used a lot in some parts of what follows, so it is convenient to have an infix operator for it.

```
Notation "x ++ y" := (app x y)
                     (right associativity, at level 60).
```

```
Example test_app1: [1;2;3] ++ [4;5] = [1;2;3;4;5].
Proof. reflexivity. Qed.
Example test_app2: nil ++ [4;5] = [4;5].
Proof. reflexivity. Qed.
Example test_app3: [1;2;3] ++ nil = [1;2;3].
Proof. reflexivity. Qed.
```

Head (with default) and Tail

Here are two smaller examples of programming with lists. The `hd` function returns the first element (the "head") of the list, while `tl` returns everything but the first element (the "tail"). Of course, the empty list has no first element, so we must pass a default value to be returned in that case.

```
Definition hd (default:nat) (l:natlist) : nat :=
  match l with
  | nil ⇒ default
  | h :: t ⇒ h
  end.

Definition tl (l:natlist) : natlist :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ t
  end.

Example test_hd1: hd 0 [1;2;3] = 1.
Proof. reflexivity. Qed.
Example test_hd2: hd 0 [] = 0.
Proof. reflexivity. Qed.
Example test_tl: tl [1;2;3] = [2;3].
Proof. reflexivity. Qed.
```

Exercises

Exercise: 2 stars, recommended (list_funs)

Complete the definitions of `nonzeros`, `oddmembers` and `countoddmembers` below. Have a look at the tests to understand what these functions should do.

```
Fixpoint nonzeros (l:natlist) : natlist
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_nonzeros:
  nonzeros [0;1;0;2;3;0;0] = [1;2;3].
  (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: NatList.test_nonzeros *)

Fixpoint oddmembers (l:natlist) : natlist
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_oddmembers:
  oddmembers [0;1;0;2;3;0;0] = [1;3].
```

```
    (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: NatList.test_oddmembers *)

Definition countoddmembers (l:natlist) : nat
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_countoddmembers1:
  countoddmembers [1;0;3;1;4;5] = 4.
  (* FILL IN HERE *) Admitted.

Example test_countoddmembers2:
  countoddmembers [0;2;4] = 0.
  (* FILL IN HERE *) Admitted.

Example test_countoddmembers3:
  countoddmembers nil = 0.
  (* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars, advanced (alternate)

Complete the definition of `alternate`, which "zips up" two lists into one, alternating between elements taken from the first list and elements from the second. See the tests below for more specific examples.

Note: one natural and elegant way of writing `alternate` will fail to satisfy Coq's requirement that all `Fixpoint` definitions be "obviously terminating." If you find yourself in this rut, look for a slightly more verbose solution that considers elements of both lists at the same time. (One possible solution requires defining a new kind of pairs, but this is not the only way.)

```
Fixpoint alternate (l1 l2 : natlist) : natlist
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_alternate1:
  alternate [1;2;3] [4;5;6] = [1;4;2;5;3;6].
  (* FILL IN HERE *) Admitted.

Example test_alternate2:
  alternate [1] [4;5;6] = [1;4;5;6].
  (* FILL IN HERE *) Admitted.

Example test_alternate3:
  alternate [1;2;3] [4] = [1;4;2;3].
  (* FILL IN HERE *) Admitted.

Example test_alternate4:
  alternate [] [20;30] = [20;30].
  (* FILL IN HERE *) Admitted.
```

□

Bags via Lists

A `bag` (or `multiset`) is like a set, except that each element can appear multiple times rather than just once. One possible implementation is to represent a bag of numbers as a list.

```
Definition bag := natlist.
```

Exercise: 3 stars, recommended (bag_functions)

Complete the following definitions for the functions `count`, `sum`, `add`, and `member` for bags.

```
Fixpoint count (v:nat) (s:bag) : nat
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

```
Fixpoint count (v:nat) (s:bag) : nat :=
match v, s with
| v, [] => O
(** sucks that I cannot do that. why can't I do this?
| v, v :: t => S (count v t)
| v, _ :: t => count v t
*)
| v, h :: t => if beq_nat v h then S (count v t) else count v t
end.
```

BECAUSE YOU DON'T NEED TO EVEN PATTERN MATCH ON V. LOOK AT YOUR NEW DEF!!

All these proofs can be done just by `reflexivity`.

```
Example test_count1: count 1 [1;2;3;1;4;1] = 3.
 (* FILL IN HERE *) Admitted.
Example test_count2: count 6 [1;2;3;1;4;1] = 0.
 (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: NatList.test_count2 *)
```

Multiset `sum` is similar to set `union`: `sum a b` contains all the elements of `a` and of `b`. (Mathematicians usually define `union` on multisets a little bit differently — using max instead of sum — which is why we don't use that name for this operation.) For `sum` we're giving you a header that does not give explicit names to the arguments. Moreover, it uses the keyword `Definition` instead of `Fixpoint`, so even if you had names for the arguments, you wouldn't be able to process them recursively. The point of stating the question this way is to encourage you to think about whether `sum` can be implemented in another way — perhaps by using functions that have already been defined.

```
Definition sum : bag → bag → bag
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_sum1: count 1 (sum [1;2;3] [1;4;1]) = 3.
 (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: NatList.test_sum1 *)

Definition add (v:nat) (s:bag) : bag
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_add1: count 1 (add 1 [1;4;1]) = 3.
 (* FILL IN HERE *) Admitted.
Example test_add2: count 5 (add 1 [1;4;1]) = 0.
 (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: NatList.test_add1 *)
(* GRADE_THEOREM 0.5: NatList.test_add2 *)

Definition member (v:nat) (s:bag) : bool
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

```
Example test_member1: member 1 [1;4;1] = true.
 (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: NatList.test_member1 *)
(* GRADE_THEOREM 0.5: NatList.test_member2 *)

Example test_member2: member 2 [1;4;1] = false.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars, optional (bag_more_functions)

Here are some more `bag` functions for you to practice with.

When `remove_one` is applied to a bag without the number to remove, it should return the same bag unchanged.

```
Fixpoint remove_one (v:nat) (s:bag) : bag
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_remove_one1:
  count 5 (remove_one 5 [2;1;5;4;1]) = 0.
  (* FILL IN HERE *) Admitted.

Example test_remove_one2:
  count 5 (remove_one 5 [2;1;4;1]) = 0.
  (* FILL IN HERE *) Admitted.

Example test_remove_one3:
  count 4 (remove_one 5 [2;1;4;5;1;4]) = 2.
  (* FILL IN HERE *) Admitted.

Example test_remove_one4:
  count 5 (remove_one 5 [2;1;5;4;5;1;4]) = 1.
  (* FILL IN HERE *) Admitted.

Fixpoint remove_all (v:nat) (s:bag) : bag
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_remove_all1: count 5 (remove_all 5 [2;1;5;4;1]) =
0.
 (* FILL IN HERE *) Admitted.
Example test_remove_all2: count 5 (remove_all 5 [2;1;4;1]) = 0.
 (* FILL IN HERE *) Admitted.
Example test_remove_all3: count 4 (remove_all 5 [2;1;4;5;1;4]) =
2.
 (* FILL IN HERE *) Admitted.
Example test_remove_all4: count 5 (remove_all 5
[2;1;5;4;5;1;4;5;1;4]) = 0.
 (* FILL IN HERE *) Admitted.
```

why isn't this working?

```
Fixpoint subset (s:bag) (s':bag) : bool :=
match s, s' with
  | [], _ => true
| h :: t, s' => leb (count h s) (count h s')
    && subset (remove_all h s) (remove_all h s')
  end.
```

```
Fixpoint subset (s1:bag) (s2:bag) : bool
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_subset1: subset [1;2] [2;1;4;1] = true.
 (* FILL IN HERE *) Admitted.
```

```
Example test_subset2: subset [1;2;2] [2;1;4;1] = false.
 (* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars, recommended (bag_theorem)

Write down an interesting theorem `bag_theorem` about bags involving the functions `count` and `add`, and prove it. Note that, since this problem is somewhat open-ended, it's possible that you may come up with a theorem which is true, but whose proof requires techniques you haven't learned yet. Feel free to ask for help if you get stuck!

```
(*
Theorem bag_theorem : ...
Proof.
  ...
Qed.
*)
```

☐

Reasoning About Lists

As with numbers, simple facts about list-processing functions can sometimes be proved entirely by simplification. For example, the simplification performed by `reflexivity` is enough for this theorem...

```
Theorem nil_app : ∀ l:natlist,
  [] ++ l = l.
Proof. reflexivity. Qed.
```

...because the `[]` is substituted into the "scrutinee" (the expression whose value is being "scrutinized" by the match) in the definition of `app`, allowing the match itself to be simplified.

Also, as with numbers, it is sometimes helpful to perform case analysis on the possible shapes (empty or non-empty) of an unknown list.

```
Theorem tl_length_pred : ∀ l:natlist,
  pred (length l) = length (tl l).
Proof.
  intros l. destruct l as [| n l'].
  - (* l = nil *)
    reflexivity.
  - (* l = cons n l' *)
    reflexivity. Qed.
```

Here, the `nil` case works because we've chosen to define `tl nil = nil`. Notice that the `as` annotation on the `destruct` tactic here introduces two names, `n` and `l'`, corresponding to the fact that the `cons` constructor for lists takes two arguments (the head and tail of the list it is constructing).

Usually, though, interesting theorems about lists require induction for their proofs.

Micro-Sermon

Simply reading example proof scripts will not get you very far! It is important to work through the details of each one, using Coq and thinking about what each step achieves. Otherwise it is more or less guaranteed that the exercises will make no sense when you get to them. 'Nuff said.

Induction on Lists

Proofs by induction over datatypes like `natlist` are a little less familiar than standard natural number induction, but the idea is equally simple. Each `Inductive` declaration defines a set of data values that can be built up using the declared constructors: a boolean can be either `true` or `false`; a number can be either `O` or `S` applied to another number; a list can be either `nil` or `cons` applied to a number and a list.

Moreover, applications of the declared constructors to one another are the *only* possible shapes that elements of an inductively defined set can have, and this fact directly gives rise to a way of reasoning about inductively defined sets: a number is either `O` or else it is `S` applied to some *smaller* number; a list is either `nil` or else it is `cons` applied to some number and some *smaller* list; etc. So, if we have in mind some proposition `P` that mentions a list `l` and we want to argue that `P` holds for *all* lists, we can reason as follows:

- First, show that `P` is true of `l` when `l` is `nil`.
- Then show that `P` is true of `l` when `l` is `cons n l'` for some number `n` and some smaller list `l'`, assuming that `P` is true for `l'`.

Since larger lists can only be built up from smaller ones, eventually reaching `nil`, these two arguments together establish the truth of `P` for all lists `l`. Here's a concrete example:

```
Theorem app_assoc : ∀ l1 l2 l3 : natlist,
  (l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3).
Proof.
  intros l1 l2 l3. induction l1 as [| n l1' IHl1'].
  - (* l1 = nil *)
    reflexivity.
  - (* l1 = cons n l1' *)
    simpl. rewrite → IHl1'. reflexivity. Qed.
```

Notice that, as when doing induction on natural numbers, the `as...` clause provided to the `induction` tactic gives a name to the induction hypothesis corresponding to the smaller list l_1' in the `cons` case. Once again, this Coq proof is not especially illuminating as a static written document — it is easy to see what's going on if you are reading the proof in an interactive Coq session and you can see the current goal and context at each point, but this state is not visible in the written-down parts of the Coq proof. So a natural-language proof — one written for human readers — will need to include more explicit signposts; in particular, it will help the reader stay oriented if we remind them exactly what the induction hypothesis is in the second case.

For comparison, here is an informal proof of the same theorem.

Theorem: For all lists l_1, l_2, and l_3, $(l_1 ++ l_2) ++ l_3 = l_1 ++ (l_2 ++ l_3)$.

Proof: By induction on l_1.

- First, suppose l_1 = [ ]. We must show

  $$([] ++ l_2) ++ l_3 = [] ++ (l_2 ++ l_3),$$

  which follows directly from the definition of ++.

- Next, suppose $l_1 = n::l_1'$, with

  $$(l_1' ++ l_2) ++ l_3 = l_1' ++ (l_2 ++ l_3)$$

  (the induction hypothesis). We must show

  $$((n :: l_1') ++ l_2) ++ l_3 = (n :: l_1') ++ (l_2 ++ l_3).$$

  By the definition of ++, this follows from

  $$n :: ((l_1' ++ l_2) ++ l_3) = n :: (l_1' ++ (l_2 ++ l_3)),$$

  which is immediate from the induction hypothesis. □

Reversing a List

For a slightly more involved example of inductive proof over lists, suppose we use `app` to define a list-reversing function `rev`:

```
Fixpoint rev (l:natlist) : natlist :=
  match l with
  | nil ⇒ nil
  | h :: t ⇒ rev t ++ [h]
  end.

Example test_rev1: rev [1;2;3] = [3;2;1].
Proof. reflexivity. Qed.
Example test_rev2: rev nil = nil.
Proof. reflexivity. Qed.
```

Properties of `rev`

Now let's prove some theorems about our newly defined `rev`. For something a bit more challenging than what we've seen, let's prove that reversing a list does not change its length. Our first attempt gets stuck in the successor case...

```
Theorem rev_length_firsttry : ∀ l : natlist,
  length (rev l) = length l.
Proof.
  intros l. induction l as [| n l' IHl'].
  - (* l =  *)
```

```
      reflexivity.
    - (* l = n :: l' *)
      (* This is the tricky case.  Let's begin as usual
         by simplifying. *)
      simpl.
      (* Now we seem to be stuck: the goal is an equality
         involving ++, but we don't have any useful equations
         in either the immediate context or in the global
         environment!  We can make a little progress by using
         the IH to rewrite the goal... *)
      rewrite <- IHl'.
      (* ... but now we can't go any further. *)
  Abort.
```

So let's take the equation relating `++` and `length` that would have enabled us to make progress and prove it as a separate lemma.

```
Theorem app_length : ∀ l1 l2 : natlist,
  length (l1 ++ l2) = (length l1) + (length l2).
Proof.
  (* WORKED IN CLASS *)
  intros l1 l2. induction l1 as [| n l1' IHl1'].
  - (* l1 = nil *)
    reflexivity.
  - (* l1 = cons *)
    simpl. rewrite → IHl1'. reflexivity. Qed.
```

Note that, to make the lemma as general as possible, we quantify over *all* `natlist`s, not just those that result from an application of `rev`. This should seem natural, because the truth of the goal clearly doesn't depend on the list having been reversed. Moreover, it is easier to prove the more general property.

Now we can complete the original proof.

```
Theorem rev_length : ∀ l : natlist,
  length (rev l) = length l.
Proof.
  intros l. induction l as [| n l' IHl'].
  - (* l = nil *)
    reflexivity.
  - (* l = cons *)
    simpl. rewrite → app_length, plus_comm.
    simpl. rewrite → IHl'. reflexivity. Qed.
```

For comparison, here are informal proofs of these two theorems:

Theorem: For all lists l_1 and l_2, `length` (l_1 ++ l_2) = `length` l_1 + `length` l_2.

Proof: By induction on l_1.

- First, suppose l_1 = `[]`. We must show

  `length ([] ++ l2) = length [] + length l2,`

  which follows directly from the definitions of `length` and `++`.

- Next, suppose `l1 = n::l1'`, with

```
length (l1' ++ l2) = length l1' + length l2.
```

  We must show

```
length ((n::l1') ++ l2) = length (n::l1') + length l2).
```

  This follows directly from the definitions of `length` and `++` together with the induction hypothesis. □

Theorem: For all lists `l`, `length (rev l) = length l`.

Proof: By induction on `l`.

- First, suppose `l = []`. We must show

```
length (rev []) = length [],
```

  which follows directly from the definitions of `length` and `rev`.

- Next, suppose `l = n::l'`, with

```
length (rev l') = length l'.
```

  We must show

```
length (rev (n :: l')) = length (n :: l').
```

  By the definition of `rev`, this follows from

```
length ((rev l') ++ [n]) = S (length l')
```

  which, by the previous lemma, is the same as

```
length (rev l') + length [n] = S (length l').
```

  This follows directly from the induction hypothesis and the definition of `length`. □

The style of these proofs is rather longwinded and pedantic. After the first few, we might find it easier to follow proofs that give fewer details (which can easily work out in our own minds or on scratch paper if necessary) and just highlight the non-obvious steps. In this more compressed style, the above proof might look like this:

Theorem: For all lists `l`, `length (rev l) = length l`.

Proof: First, observe that `length (l ++ [n]) = S (length l)` for any `l` (this follows by a straightforward induction on `l`). The main property again follows by induction on `l`, using the observation together with the induction hypothesis in the case where `l = n'::l'`. □

Which style is preferable in a given situation depends on the sophistication of the expected audience and how similar the proof at hand is to ones that the audience will already be familiar with. The more pedantic style is a good default for our present purposes.

Search

We've seen that proofs can make use of other theorems we've already proved, e.g., using `rewrite`. But in order to refer to a theorem, we need to know its name! Indeed, it is often hard even to remember what theorems have been proven, much less what they are called.

Coq's `Search` command is quite helpful with this. Typing `Search foo` will cause Coq to display a list of all theorems involving `foo`. For example, try uncommenting the following line to see a list of theorems that we have proved about `rev`:

```
(*  Search rev. *)
```

Keep `Search` in mind as you do the following exercises and throughout the rest of the book; it can save you a lot of time!

If you are using ProofGeneral, you can run `Search` with `C-c C-a C-a`. Pasting its response into your buffer can be accomplished with `C-c C-;`.

List Exercises, Part 1

Exercise: 3 stars (list_exercises)

More practice with lists:

```
Theorem app_nil_r : ∀ l : natlist,
  l ++ [] = l.
Proof.
  (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: NatList.app_nil_r *)

Theorem rev_app_distr: ∀ l1 l2 : natlist,
  rev (l1 ++ l2) = rev l2 ++ rev l1.
Proof.
  (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: NatList.rev_app_distr *)

Theorem rev_involutive : ∀ l : natlist,
  rev (rev l) = l.
Proof.
  (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: NatList.rev_involutive *)
```

There is a short solution to the next one. If you find yourself getting tangled up, step back and try to look for a simpler way.

easier than applying assoc twice?

```
Theorem app_assoc4 : ∀ l1 l2 l3 l4 : natlist,
  l1 ++ (l2 ++ (l3 ++ l4)) = ((l1 ++ l2) ++ l3) ++ l4.
Proof.
  (* FILL IN HERE *) Admitted.
(* GRADE_THEOREM 0.5: NatList.app_assoc4 *)
```

An exercise about your implementation of `nonzeros`:

```
Lemma nonzeros_app : ∀ l1 l2 : natlist,
  nonzeros (l1 ++ l2) = (nonzeros l1) ++ (nonzeros l2).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars (beq_natlist)

Fill in the definition of `beq_natlist`, which compares lists of numbers for equality. Prove that `beq_natlist l l` yields `true` for every list `l`.

```
Fixpoint beq_natlist (l1 l2 : natlist) : bool
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.

Example test_beq_natlist1 :
  (beq_natlist nil nil = true).
 (* FILL IN HERE *) Admitted.

Example test_beq_natlist2 :
  beq_natlist [1;2;3] [1;2;3] = true.
(* FILL IN HERE *) Admitted.

Example test_beq_natlist3 :
  beq_natlist [1;2;3] [1;2;4] = false.
 (* FILL IN HERE *) Admitted.

Theorem beq_natlist_refl : ∀ l:natlist,
  true = beq_natlist l l.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

List Exercises, Part 2

Here are a couple of little theorems to prove about your definitions about bags above.

Exercise: 1 star (count_member_nonzero)

```
Theorem count_member_nonzero : ∀ (s : bag),
  leb 1 (count 1 (1 :: s)) = true.
Proof.
  (* FILL IN HERE *) Admitted.
```

sometimes catches you by surprise that a proof can be way simpler than what you thoght!!! like this one!!

☐

The following lemma about `leb` might help you in the next exercise.

```
Theorem ble_n_Sn : ∀ n,
  leb n (S n) = true.
Proof.
  intros n. induction n as [| n' IHn'].
  - (* 0 *)
    simpl. reflexivity.
  - (* S n' *)
    simpl. rewrite IHn'. reflexivity. Qed.
```

Exercise: 3 stars, advanced (remove_decreases_count)

```
Theorem remove_decreases_count: ∀ (s : bag),
  leb (count 0 (remove_one 0 s)) (count 0 s) = true.
Proof.
  (* FILL IN HERE *) Admitted.
```

really enjoyed this one. it's full of aha moments!!
nonzeros_app will be the same! interesting that
I figured it out in this one and not the other one :)

☐

Exercise: 3 stars, optional (bag_count_sum)

Write down an interesting theorem `bag_count_sum` about bags involving the functions `count` and `sum`, and prove it using Coq. (You may find that the difficulty of the proof depends on how you defined `count`!)

```
(* FILL IN HERE *)
```

☐

Exercise: 4 stars, advanced (rev_injective)

Prove that the `rev` function is injective — that is,

```
∀ (l1 l2 : natlist), rev l1 = rev l2 → l1 = l2.
```

(There is a hard way and an easy way to do this.)

```
(* FILL IN HERE *)
```

☐

Options

Suppose we want to write a function that returns the `n`th element of some list. If we give it type `nat → natlist → nat`, then we'll have to choose some number to return when the list is too short...

```
Fixpoint nth_bad (l:natlist) (n:nat) : nat :=
  match l with
  | nil ⇒ 42 (* arbitrary! *)
  | a :: l' ⇒ match beq_nat n O with
               | true ⇒ a
               | false ⇒ nth_bad l' (pred n)
               end
  end.
```

This solution is not so good: If `nth_bad` returns `42`, we can't tell whether that value actually appears on the input without further processing. A better alternative is to change the return type of `nth_bad` to include an error value as a possible outcome. We call this type `natoption`.

```
Inductive natoption : Type :=
  | Some : nat → natoption
  | None : natoption.
```

We can then change the above definition of `nth_bad` to return `None` when the list is too short and `Some a` when the list has enough members and `a` appears at position `n`.

We call this new function `nth_error` to indicate that it may result in an error.

```
Fixpoint nth_error (l:natlist) (n:nat) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ match beq_nat n O with
               | true ⇒ Some a
               | false ⇒ nth_error l' (pred n)
               end
  end.

Example test_nth_error1 : nth_error [4;5;6;7] 0 = Some 4.
+
Example test_nth_error2 : nth_error [4;5;6;7] 3 = Some 7.
+
Example test_nth_error3 : nth_error [4;5;6;7] 9 = None.
+
```

(In the HTML version, the boilerplate proofs of these examples are elided. Click on a box if you want to see one.)

This example is also an opportunity to introduce one more small feature of Coq's programming language: conditional expressions...

```
Fixpoint nth_error' (l:natlist) (n:nat) : natoption :=
  match l with
  | nil ⇒ None
  | a :: l' ⇒ if beq_nat n O then Some a
               else nth_error' l' (pred n)
  end.
```

Coq's conditionals are exactly like those found in any other language, with one small generalization. Since the boolean type is not built in, Coq actually supports conditional expressions over *any* inductively defined type with exactly two constructors. The guard is considered true if it evaluates to the first constructor in the `Inductive` definition and false if it evaluates to the second.

The function below pulls the `nat` out of a `natoption`, returning a supplied default in the `None` case.

```
Definition option_elim (d : nat) (o : natoption) : nat :=
  match o with
  | Some n' ⇒ n'
  | None ⇒ d
  end.
```

Exercise: 2 stars (hd_error)

Using the same idea, fix the `hd` function from earlier so we don't have to pass a default element for the `nil` case.

```
Definition hd_error (l : natlist) : natoption
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *).
Admitted.
```

```
Example test_hd_error1 : hd_error [] = None.
 (* FILL IN HERE *) Admitted.

Example test_hd_error2 : hd_error [1] = Some 1.
 (* FILL IN HERE *) Admitted.

Example test_hd_error3 : hd_error [5;6] = Some 5.
 (* FILL IN HERE *) Admitted.
```

☐

Exercise: 1 star, optional (option_elim_hd)

This exercise relates your new `hd_error` to the old `hd`.

```
Theorem option_elim_hd : ∀ (l:natlist) (default:nat),
  hd default l = option_elim default (hd_error l).
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

```
End NatList.
```

Partial Maps

As a final illustration of how data structures can be defined in Coq, here is a simple *partial map* data type, analogous to the map or dictionary data structures found in most programming languages.

First, we define a new inductive datatype `id` to serve as the "keys" of our partial maps.

```
Inductive id : Type :=
  | Id : nat → id.
```

Internally, an `id` is just a number. Introducing a separate type by wrapping each nat with the tag `Id` makes definitions more readable and gives us the flexibility to change representations later if we wish.

We'll also need an equality test for `id`s:

```
Definition beq_id (x1 x2 : id) :=
  match x1, x2 with
  | Id n1, Id n2 ⇒ beq_nat n1 n2
  end.
```

Exercise: 1 star (beq_id_refl)

```
Theorem beq_id_refl : ∀ x, true = beq_id x x.
Proof.
  (* FILL IN HERE *) Admitted.
```

☐

Now we define the type of partial maps:

```
Module PartialMap.
Export NatList.
```

```
Inductive partial_map : Type :=
  | empty : partial_map
  | record : id → nat → partial_map → partial_map.
```

This declaration can be read: "There are two ways to construct a `partial_map`: either using the constructor `empty` to represent an empty partial map, or by applying the constructor `record` to a key, a value, and an existing `partial_map` to construct a `partial_map` with an additional key-to-value mapping."

The `update` function overrides the entry for a given key in a partial map (or adds a new entry if the given key is not already present).

```
Definition update (d : partial_map)
                  (x : id) (value : nat)
                  : partial_map :=
  record x value d.
```

Last, the `find` function searches a `partial_map` for a given key. It returns `None` if the key was not found and `Some val` if the key was associated with `val`. If the same key is mapped to multiple values, `find` will return the first one it encounters.

```
Fixpoint find (x : id) (d : partial_map) : natoption :=
  match d with
  | empty ⇒ None
  | record y v d' ⇒ if beq_id x y
                     then Some v
                     else find x d'
  end.
```

Exercise: 1 star (update_eq)

```
Theorem update_eq :
  ∀ (d : partial_map) (x : id) (v: nat),
    find x (update d x v) = Some v.
Proof.
 (* FILL IN HERE *) Admitted.
```

☐

Exercise: 1 star (update_neq)

```
Theorem update_neq :
  ∀ (d : partial_map) (x y : id) (o: nat),
    beq_id x y = false → find x (update d y o) = find x d.
Proof.
 (* FILL IN HERE *) Admitted.
```

☐

```
End PartialMap.
```

Exercise: 2 stars (baz_num_elts)

Consider the following inductive definition:

```
Inductive baz : Type :=
  | Baz1 : baz → baz
```

```
  | Baz2 : baz → bool → baz.
```

ask someone? How *many* elements does the type `baz` have? (Explain your answer in words, preferrably English.)

```
(* FILL IN HERE *)
```

□