INDPROP

INDUCTIVELY DEFINED PROPOSITIONS

```
Set Warnings "-notation-overridden,-parsing".
From LF Require Export Logic.
From Coq Require Export Lia.
```

Inductively Defined Propositions

In the Logic chapter, we looked at several ways of writing propositions, including conjunction, disjunction, and existential quantification. In this chapter, we bring yet another new tool into the mix: *inductive definitions*.

Note: For the sake of simplicity, most of this chapter uses an inductive definition of "evenness" as a running example. You may find this confusing, since we already have a perfectly good way of defining evenness as a proposition (`n` is even if it is equal to the result of doubling some number); if so, rest assured that we will see many more compelling examples of inductively defined propositions toward the end of this chapter and in future chapters.

In past chapters, we have seen two ways of stating that a number `n` is even: We can say

(1) `evenb n = true`, or

(2) `∃ k, n = double k`.

Yet another possibility is to say that `n` is even if we can establish its evenness from the following rules:

- Rule `ev_0`: The number `0` is even.
- Rule `ev_SS`: If `n` is even, then `S (S n)` is even.

To illustrate how this new definition of evenness works, let's imagine using it to show that `4` is even. By rule `ev_SS`, it suffices to show that `2` is even. This, in turn, is again guaranteed by rule `ev_SS`, as long as we can show that `0` is even. But this last fact follows directly from the `ev_0` rule.

We will see many definitions like this one during the rest of the course. For purposes of informal discussions, it is helpful to have a lightweight notation that makes them easy to read and write. *Inference rules* are one such notation. (We'll use `ev` for the name of this property, since `even` is already used.)

$$\frac{}{\texttt{ev 0}}\ \text{(ev\_0)}$$

$$\frac{\texttt{ev n}}{\texttt{ev (S (S n))}}\ \text{(ev\_SS)}$$

Each of the textual rules that we started with is reformatted here as an inference rule; the intended reading is that, if the *premises* above the line all hold, then the *conclusion* below the line follows. For example, the rule `ev_SS` says that, if `n` satisfies `ev`, then `S (S n)` also does. If a rule has no premises above the line, then its conclusion holds unconditionally.

We can represent a proof using these rules by combining rule applications into a *proof tree*. Here's how we might transcribe the above proof that `4` is even:

```
--------  (ev_0)
 ev 0
--------  (ev_SS)
 ev 2
--------  (ev_SS)
 ev 4
```

(Why call this a "tree", rather than a "stack", for example? Because, in general, inference rules can have multiple premises. We will see examples of this shortly.)

Inductive Definition of Evenness

Putting all of this together, we can translate the definition of evenness into a formal Coq definition using an `Inductive` declaration, where each constructor corresponds to an inference rule:

```
Inductive ev : nat → Prop :=
| ev_0 : ev 0
| ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

This definition is interestingly different from previous uses of `Inductive`. For one thing, we are defining not a `Type` (like `nat`) or a function yielding a `Type` (like `list`), but rather a function from `nat` to `Prop` -- that is, a property of numbers. But what is really new is that, because the `nat` argument of `ev` appears to the *right* of the colon on the first line, it is allowed to take different values in the types of different constructors: `0` in the type of `ev_0` and `S (S n)` in the type of `ev_SS`. Accordingly, the type of each constructor must be specified explicitly (after a colon), and each constructor's type must have the form `ev n` for some natural number `n`.

In contrast, recall the definition of `list`:

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

This definition introduces the `X` parameter *globally*, to the *left* of the colon, forcing the result of `nil` and `cons` to be the same (i.e., `list X`). Had we tried to bring `nat` to the left of the colon in defining `ev`, we would have seen an error:

```
Fail Inductive wrong_ev (n : nat) : Prop :=
| wrong_ev_0 : wrong_ev 0
```

```
| wrong_ev_SS (H: wrong_ev n) : wrong_ev (S (S n)).
(* ===> Error: Last occurrence of "wrong_ev" must have "n"
        as 1st argument in "wrong_ev 0". *)
```

In an `Inductive` definition, an argument to the type constructor on the left of the colon is called a "parameter", whereas an argument on the right is called an "index" or "annotation."

For example, in `Inductive list (X : Type) := ...`, the `X` is a parameter; in `Inductive ev : nat → Prop := ...`, the unnamed `nat` argument is an index.

We can think of the definition of `ev` as defining a Coq property `ev : nat → Prop`, together with "evidence constructors" `ev_0 : ev 0` and `ev_SS : ∀ n, ev n → ev (S (S n))`.

Such "evidence constructors" have the same status as proven theorems. In particular, we can use Coq's `apply` tactic with the rule names to prove `ev` for particular numbers...

```
Theorem ev_4 : ev 4.
Proof. apply ev_SS. apply ev_SS. apply ev_0. Qed.
```

... or we can use function application syntax:

```
Theorem ev_4' : ev 4.
Proof. apply (ev_SS 2 (ev_SS 0 ev_0)). Qed.
```

We can also prove theorems that have hypotheses involving `ev`.

```
Theorem ev_plus4 : ∀ n, ev n → ev (4 + n).
Proof.
  intros n. simpl. intros Hn.
  apply ev_SS. apply ev_SS. apply Hn.
Qed.
```

Exercise: 1 star, standard (ev_double)

```
Theorem ev_double : ∀ n,
  ev (double n).
Proof.
  (* FILL IN HERE *) Admitted.
□
```

Using Evidence in Proofs

Besides *constructing* evidence that numbers are even, we can also *destruct* such evidence, which amounts to reasoning about how it could have been built.

Introducing `ev` with an `Inductive` declaration tells Coq not only that the constructors `ev_0` and `ev_SS` are valid ways to build evidence that some number is `ev`, but also that these two constructors are the *only* ways to build evidence that numbers are `ev`.

In other words, if someone gives us evidence `E` for the assertion `ev n`, then we know that `E` must be one of two things:

- `E` is `ev_0` (and `n` is `O`), or
- `E` is `ev_SS n' E'` (and `n` is `S (S n')`, where `E'` is evidence for `ev n'`).

This suggests that it should be possible to analyze a hypothesis of the form `ev n` much as we do inductively defined data structures; in particular, it should be possible to argue by *induction* and *case analysis* on such evidence. Let's look at a few examples to see what this means in practice.

Inversion on Evidence

Suppose we are proving some fact involving a number `n`, and we are given `ev n` as a hypothesis. We already know how to perform case analysis on `n` using `destruct` or `induction`, generating separate subgoals for the case where `n = O` and the case where `n = S n'` for some `n'`. But for some proofs we may instead want to analyze the evidence that `ev n` *directly*. As a tool, we can prove our characterization of evidence for `ev n`, using `destruct`.

```
Theorem ev_inversion :
  ∀ (n : nat), ev n →
    (n = 0) ∨ (∃ n', n = S (S n') ∧ ev n').
Proof.
  intros n E.
  destruct E as [ | n' E'] eqn:EE.
  - (* E = ev_0 : ev 0 *)
    left. reflexivity.
  - (* E = ev_SS n' E' : ev (S (S n')) *)
    right. ∃ n'. split. reflexivity. apply E'.
Qed.
```

The following theorem can easily be proved using `destruct` on evidence.

```
Theorem ev_minus2 : ∀ n,
  ev n → ev (pred (pred n)).
Proof.
  intros n E.
  destruct E as [| n' E'] eqn:EE.
  - (* E = ev_0 *) simpl. apply ev_0.
  - (* E = ev_SS n' E' *) simpl. apply E'.
Qed.
```

However, this variation cannot easily be handled with just `destruct`.

```
Theorem evSS_ev : ∀ n,
  ev (S (S n)) → ev n.
```

Intuitively, we know that evidence for the hypothesis cannot consist just of the `ev_0` constructor, since `O` and `S` are different constructors of the type `nat`; hence, `ev_SS` is the only case that applies. Unfortunately, `destruct` is not smart enough to realize this, and it still generates two subgoals. Even worse, in doing so, it keeps the final goal unchanged, failing to provide any useful information for completing the proof.

```
Proof.
  intros n E.
  destruct E as [| n' E'] eqn:EE.
  - (* E = ev_0. *)
    (* We must prove that n is even from no assumptions! *)
Abort.
```

What happened, exactly? Calling `destruct` has the effect of replacing all occurrences of the property argument by the values that correspond to each constructor. This is enough in the case of `ev_minus2` because that argument `n` is mentioned directly in the final goal. However, it doesn't help in the case of `evSS_ev` since the term that gets replaced (`S (S n)`) is not mentioned anywhere.

If we remember that term `S (S n)`, the proof goes through. (We'll discuss `remember` in more detail below.)

```
Theorem evSS_ev_remember : ∀ n,
  ev (S (S n)) → ev n.
Proof.
  intros n E. remember (S (S n)) as k eqn:Hk. destruct E as [|n' E'] eqn:EE.
  - (* E = ev_0 *)
    (* Now we do have an assumption, in which k = S (S n) has been
       rewritten as 0 = S (S n) by destruct. That assumption
       gives us a contradiction. *)
    discriminate Hk.
  - (* E = ev_S n' E' *)
    (* This time k = S (S n) has been rewritten as S (S n') = S (S n). *)
    injection Hk as Heq. rewrite <- Heq. apply E'.
Qed.
```

Alternatively, the proof is straightforward using our inversion lemma.

```
Theorem evSS_ev : ∀ n, ev (S (S n)) → ev n.
Proof.
 intros n H. apply ev_inversion in H.
 destruct H as [H0|H1].
 - discriminate H0.
 - destruct H1 as [n' [Hnm Hev]]. injection Hnm as Heq.
   rewrite Heq. apply Hev.
Qed.
```

Note how both proofs produce two subgoals, which correspond to the two ways of proving `ev`. The first subgoal is a contradiction that is discharged with `discriminate`. The second subgoal makes use of `injection` and `rewrite`. Coq provides a handy tactic called `inversion` that factors out that common pattern.

The `inversion` tactic can detect (1) that the first case (`n = 0`) does not apply and (2) that the `n'` that appears in the `ev_SS` case must be the same as `n`. It has an "`as`" variant similar to `destruct`, allowing us to assign names rather than have Coq choose them.

```
Theorem evSS_ev' : ∀ n,
  ev (S (S n)) → ev n.
Proof.
  intros n E.
  inversion E as [| n' E' Heq].
  (* We are in the E = ev_SS n' E' case now. *)
  apply E'.
Qed.
```

The `inversion` tactic can apply the principle of explosion to "obviously contradictory" hypotheses involving inductively defined properties, something that takes a bit more work using our inversion lemma. For example:

```
Theorem one_not_even : ¬ ev 1.
Proof.
  intros H. apply ev_inversion in H.
  destruct H as [ | [m [Hm _]]].
  - discriminate H.
  - discriminate Hm.
Qed.

Theorem one_not_even' : ¬ ev 1.
  intros H. inversion H. Qed.
```

Exercise: 1 star, standard (inversion_practice)

Prove the following result using `inversion`. (For extra practice, you can also prove it using the inversion lemma.)

```
Theorem SSSSev__even : ∀ n,
  ev (S (S (S (S n)))) → ev n.
Proof.
  (* FILL IN HERE *) Admitted.
□
```

Exercise: 1 star, standard (ev5_nonsense)

Prove the following result using `inversion`.

```
Theorem ev5_nonsense :
  ev 5 → 2 + 2 = 9.
Proof.
  (* FILL IN HERE *) Admitted.
□
```

The `inversion` tactic does quite a bit of work. For example, when applied to an equality assumption, it does the work of both `discriminate` and `injection`. In addition, it carries out the `intros` and `rewrite`s that are typically necessary in the case of `injection`. It can also be applied, more generally, to analyze evidence for inductively defined propositions. As examples, we'll use it to reprove some theorems from chapter Tactics. (Here we are being a bit lazy by omitting the `as` clause from `inversion`, thereby asking Coq to choose names for the variables and hypotheses that it introduces.)

```
Theorem inversion_ex1 : ∀ (n m o : nat),
  [n; m] = [o; o] →
  [n] = [m].
Proof.
  intros n m o H. inversion H. reflexivity. Qed.

Theorem inversion_ex2 : ∀ (n : nat),
  S n = O →
  2 + 2 = 5.
Proof.
  intros n contra. inversion contra. Qed.
```

Here's how `inversion` works in general. Suppose the name `H` refers to an assumption `P` in the current context, where `P` has been defined by an `Inductive` declaration. Then, for each of the constructors of `P`, `inversion H` generates a subgoal in which `H` has been replaced by the exact, specific conditions under which this constructor could have been used to prove `P`. Some of these subgoals will be self-contradictory; `inversion` throws these away. The ones that are left represent the cases that must be proved to establish the original goal. For those, `inversion` adds all equations into the proof context that must hold of the arguments given to `P` (e.g., `S (S n') = n` in the proof of `evSS_ev`).

The `ev_double` exercise above shows that our new notion of evenness is implied by the two earlier ones (since, by `even_bool_prop` in chapter Logic, we already know that those are equivalent to each other). To show that all three coincide, we just need the following lemma.

```
Lemma ev_even_firsttry : ∀ n,
  ev n → even n.
Proof.
  (* WORKED IN CLASS *)
```

We could try to proceed by case analysis or induction on `n`. But since `ev` is mentioned in a premise, this strategy would probably lead to a dead end, because (as we've noted before) the induction hypothesis will talk about n-1 (which is *not* even!). Thus, it seems better to first try `inversion` on the evidence for `ev`. Indeed, the first case can be solved trivially. And we can seemingly make progress on the second case with a helper lemma.

```
  intros n E. inversion E as [EQ' | n' E' EQ'].
  - (* E = ev_0 *)
    ∃ 0. reflexivity.
  - (* E = ev_SS n' E' *) simpl.
```

Unfortunately, the second case is harder. We need to show `∃ k, S (S n') = double k`, but the only available assumption is `E'`, which states that `ev n'` holds. Since this isn't directly useful, it seems that we are stuck and that performing case analysis on `E` was a waste of time.

If we look more closely at our second goal, however, we can see that something interesting happened: By performing case analysis on `E`, we were able to reduce the original result to a similar one that involves a *different* piece of evidence for `ev`: namely `E'`. More formally, we can finish our proof by showing that

```
∃ k', n' = double k',
```

which is the same as the original statement, but with `n'` instead of `n`. Indeed, it is not difficult to convince Coq that this intermediate result suffices.

Unforunately, now we are stuck. To make that apparent, let's move `E'` back into the goal from the hypotheses.

```
    generalize dependent E'.
```

Now it is clear we are trying to prove another instance of the same theorem we set out to prove. This instance is with `n'`, instead of `n`, where `n'` is a smaller natural number than `n`.

```
Abort.
```

Induction on Evidence

If this looks familiar, it is no coincidence: We've encountered similar problems in the Induction chapter, when trying to use case analysis to prove results that required induction. And once again the solution is... induction!

The behavior of `induction` on evidence is the same as its behavior on data: It causes Coq to generate one subgoal for each constructor that could have used to build that evidence, while providing an induction hypothesis for each recursive occurrence of the property in question.

To prove a property of `n` holds for all numbers for which `ev n` holds, we can use induction on `ev n`. This requires us to prove two things, corresponding to the two ways in which `ev n` could have been constructed. If it was constructed by `ev_0`, then `n=0`, and the property must hold of `0`. If it was constructed by `ev_SS`, then the evidence of `ev n` is of the form `ev_SS n' E'`, where `n = S (S n')` and `E'` is evidence for `ev n'`. In this case, the inductive hypothesis says that the property we are trying to prove holds for `n'`.

Let's try our current lemma again:

```
Lemma ev_even : ∀ n,
  ev n → even n.
Proof.
  intros n E.
  induction E as [|n' E' IH].
  - (* E = ev_0 *)
    ∃ 0. reflexivity.
  - (* E = ev_SS n' E'
       with IH : even E' *)
    unfold even in IH.
    destruct IH as [k Hk].
    rewrite Hk. ∃ (S k). simpl. reflexivity.
Qed.
```

Here, we can see that Coq produced an `IH` that corresponds to `E'`, the single recursive occurrence of `ev` in its own definition. Since `E'` mentions `n'`, the induction hypothesis talks about `n'`, as opposed to `n` or some other number.

The equivalence between the second and third definitions of evenness now follows.

```
Theorem ev_even_iff : ∀ n,
  ev n ↔ even n.
Proof.
  intros n. split.
  - (* -> *) apply ev_even.
  - (* <- *) unfold even. intros [k Hk]. rewrite Hk. apply ev_double.
Qed.
```

As we will see in later chapters, induction on evidence is a recurring technique across many areas, and in particular when formalizing the semantics of programming languages, where many properties of interest are defined inductively.

The following exercises provide simple examples of this technique, to help you familiarize yourself with it.

Exercise: 2 stars, standard (ev_sum)

```
Theorem ev_sum : ∀ n m, ev n → ev m → ev (n + m).
Proof.
  (* FILL IN HERE *) Admitted.
□
```

Exercise: 4 stars, advanced, optional (ev'_ev)

In general, there may be multiple ways of defining a property inductively. For example, here's a (slightly contrived) alternative definition for `ev`:

```
Inductive ev' : nat → Prop :=
| ev'_0 : ev' 0
| ev'_2 : ev' 2
| ev'_sum n m (Hn : ev' n) (Hm : ev' m) : ev' (n + m).
```

Prove that this definition is logically equivalent to the old one. To streamline the proof, use the technique (from Logic) of applying theorems to arguments, and note that the same technique works with constructors of inductively defined propositions.

```
Theorem ev'_ev : ∀ n, ev' n ↔ ev n.
Proof.
 (* FILL IN HERE *) Admitted.
□
```

Exercise: 3 stars, advanced, especially useful (ev_ev__ev)

There are two pieces of evidence you could attempt to induct upon here. If one doesn't work, try the other.

```
Theorem ev_ev__ev : ∀ n m,
  ev (n+m) → ev n → ev m.
Proof.
  (* FILL IN HERE *) Admitted.
□
```

Exercise: 3 stars, standard, optional (ev_plus_plus)

This exercise can be completed without induction or case analysis. But, you will need a clever assertion and some tedious rewriting. Hint: is `(n+m) + (n+p)` even?

```
Theorem ev_plus_plus : ∀ n m p,
  ev (n+m) → ev (n+p) → ev (m+p).
Proof.
  (* FILL IN HERE *) Admitted.
□
```

Inductive Relations

A proposition parameterized by a number (such as `ev`) can be thought of as a *property* -- i.e., it defines a subset of `nat`, namely those numbers for which the proposition is provable. In the same way, a two-argument proposition can be thought of as a *relation* -- i.e., it defines a set of pairs for which the proposition is provable.

```
Module Playground.
```

Just like properties, relations can be defined inductively. One useful example is the "less than or equal to" relation on numbers.

The following definition should be fairly intuitive. It says that there are two ways to give evidence that one number is less than or equal to another: either observe that they are the same number, or give evidence that the first is less than or equal to the predecessor of the second.

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

Notation "n <= m" := (le n m).
```

Proofs of facts about `≤` using the constructors `le_n` and `le_S` follow the same patterns as proofs about properties, like `ev` above. We can `apply` the constructors to prove `≤` goals (e.g., to show that `3<=3` or `3<=6`), and we can use tactics like `inversion` to extract information from `≤` hypotheses in the context (e.g., to prove that `(2 ≤ 1) → 2+2=5`.)

Here are some sanity checks on the definition. (Notice that, although these are the same kind of simple "unit tests" as we gave for the testing functions we wrote in the first few lectures, we must construct their proofs explicitly -- `simpl` and `reflexivity` don't do the job, because the proofs aren't just a matter of simplifying computations.)

```
Theorem test_le1 :
  3 ≤ 3.
Proof.
  (* WORKED IN CLASS *)
  apply le_n. Qed.

Theorem test_le2 :
  3 ≤ 6.
Proof.
  (* WORKED IN CLASS *)
  apply le_S. apply le_S. apply le_S. apply le_n. Qed.

Theorem test_le3 :
  (2 ≤ 1) → 2 + 2 = 5.
Proof.
  (* WORKED IN CLASS *)
  intros H. inversion H. inversion H2. Qed.
```

The "strictly less than" relation `n < m` can now be defined in terms of `le`.

```
Definition lt (n m:nat) := le (S n) m.

Notation "m < n" := (lt m n).

End Playground.
```

Here are a few more simple relations on numbers:

```
Inductive square_of : nat → nat → Prop :=
  | sq n : square_of n (n × n).

Inductive next_nat : nat → nat → Prop :=
  | nn n : next_nat n (S n).

Inductive next_ev : nat → nat → Prop :=
  | ne_1 n (H: ev (S n)) : next_ev n (S n)
  | ne_2 n (H: ev (S (S n))) : next_ev n (S (S n)).
```

Exercise: 2 stars, standard, optional (total_relation)

Define an inductive binary relation `total_relation` that holds between every pair of natural numbers.

```
(* FILL IN HERE *)
□
```

Exercise: 2 stars, standard, optional (empty_relation)

Define an inductive binary relation `empty_relation` (on numbers) that never holds.

```
(* FILL IN HERE *)
□
```

From the definition of `le`, we can sketch the behaviors of `destruct`, `inversion`, and `induction` on a hypothesis `H` providing evidence of the form `le` e_1 e_2. Doing `destruct H` will generate two cases. In the first case, $e_1 = e_2$, and it will replace instances of e_2 with e_1 in the goal and context. In the second case, e_2 = `S n'` for some `n'` for which `le` e_1 `n'` holds, and it will replace instances of e_2 with `S n'`. Doing `inversion H` will remove impossible cases and add generated equalities to the context for further use. Doing `induction H` will, in the second case, add the induction hypothesis that the goal holds when e_2 is replaced with `n'`.

Exercise: 3 stars, standard, optional (le_exercises)

Here are a number of facts about the ≤ and < relations that we are going to need later in the course. The proofs make good practice exercises.

```
Lemma le_trans : ∀ m n o, m ≤ n → n ≤ o → m ≤ o.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem O_le_n : ∀ n,
  0 ≤ n.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem n_le_m__Sn_le_Sm : ∀ n m,
  n ≤ m → S n ≤ S m.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem Sn_le_Sm__n_le_m : ∀ n m,
  S n ≤ S m → n ≤ m.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem le_plus_l : ∀ a b,
  a ≤ a + b.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem plus_le : ∀ n1 n2 m,
  n1 + n2 ≤ m →
  n1 ≤ m ∧ n2 ≤ m.
Proof.
 (* FILL IN HERE *) Admitted.
```

Hint: the next one may be easiest to prove by induction on `n`.

```
Theorem add_le_cases : ∀ n m p q,
    n + m ≤ p + q → n ≤ p ∨ m ≤ q.
Proof.
(* FILL IN HERE *) Admitted.

Theorem lt_S : ∀ n m,
  n < m →
  n < S m.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem plus_lt : ∀ n1 n2 m,
  n1 + n2 < m →
  n1 < m ∧ n2 < m.
Proof.
(* FILL IN HERE *) Admitted.

Theorem leb_complete : ∀ n m,
  n <=? m = true → n ≤ m.
Proof.
  (* FILL IN HERE *) Admitted.
```

Hint: The next one may be easiest to prove by induction on `m`.

```
Theorem leb_correct : ∀ n m,
  n ≤ m →
  n <=? m = true.
Proof.
  (* FILL IN HERE *) Admitted.
```

Hint: The next one can easily be proved without using `induction`.

```
Theorem leb_true_trans : ∀ n m o,
  n <=? m = true → m <=? o = true → n <=? o = true.
Proof.
  (* FILL IN HERE *) Admitted.
```
☐

Exercise: 2 stars, standard, optional (leb_iff)

```
Theorem leb_iff : ∀ n m,
  n <=? m = true ↔ n ≤ m.
Proof.
  (* FILL IN HERE *) Admitted.
```
☐

```
Module R.
```

Exercise: 3 stars, standard, especially useful (R_provability)

We can define three-place relations, four-place relations, etc., in just the same way as binary relations. For example, consider the following three-place relation on numbers:

```
Inductive R : nat → nat → nat → Prop :=
   | c1 : R 0 0 0
   | c2 m n o (H : R m n o) : R (S m) n (S o)
   | c3 m n o (H : R m n o) : R m (S n) (S o)
   | c4 m n o (H : R (S m) (S n) (S (S o))) : R m n o
   | c5 m n o (H : R m n o) : R n m o.
```

- Which of the following propositions are provable?
  - `R 1 1 2`
  - `R 2 2 6`
- If we dropped constructor c_5 from the definition of `R`, would the set of provable propositions change? Briefly (1 sentence) explain your answer.
- If we dropped constructor c_4 from the definition of `R`, would the set of provable propositions change? Briefly (1 sentence) explain your answer.

```
(* FILL IN HERE *)

(* Do not modify the following line: *)
Definition manual_grade_for_R_provability : option (nat×string) := None.
```
☐

Exercise: 3 stars, standard, optional (R_fact)

The relation R above actually encodes a familiar function. Figure out which function; then state and prove this equivalence in Coq?

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

Theorem R_equiv_fR : ∀ m n o, R m n o ↔ fR m n = o.
Proof.
(* FILL IN HERE *) Admitted.
□

End R.
```

Exercise: 2 stars, advanced (subsequence)

A list is a *subsequence* of another list if all of the elements in the first list occur in the same order in the second list, possibly with some extra elements in between. For example,

```
[1;2;3]
```

is a subsequence of each of the lists

```
[1;2;3]
[1;1;1;2;2;3]
[1;2;7;3]
[5;6;1;9;9;2;7;3;8]
```

but it is *not* a subsequence of any of the lists

```
[1;2]
[1;3]
[5;6;2;1;7;3;8].
```

- Define an inductive proposition `subseq` on `list nat` that captures what it means to be a subsequence. (Hint: You'll need three cases.)
- Prove `subseq_refl` that subsequence is reflexive, that is, any list is a subsequence of itself.
- Prove `subseq_app` that for any lists l_1, l_2, and l_3, if l_1 is a subsequence of l_2, then l_1 is also a subsequence of l_2 ++ l_3.
- (Optional, harder) Prove `subseq_trans` that subsequence is transitive -- that is, if l_1 is a subsequence of l_2 and l_2 is a subsequence of l_3, then l_1 is a subsequence of l_3. Hint: choose your induction carefully!

```
Inductive subseq : list nat → list nat → Prop :=
(* FILL IN HERE *)
.

Theorem subseq_refl : ∀ (l : list nat), subseq l l.
Proof.
  (* FILL IN HERE *) Admitted.

Theorem subseq_app : ∀ (l1 l2 l3 : list nat),
  subseq l1 l2 →
  subseq l1 (l2 ++ l3).
Proof.
  (* FILL IN HERE *) Admitted.

Theorem subseq_trans : ∀ (l1 l2 l3 : list nat),
  subseq l1 l2 →
  subseq l2 l3 →
  subseq l1 l3.
Proof.
  (* FILL IN HERE *) Admitted.
□
```

Exercise: 2 stars, standard, optional (R_provability2)

Suppose we give Coq the following definition:

```
Inductive R : nat → list nat → Prop :=
  | c1 : R 0 []
  | c2 n l (H: R n l) : R (S n) (n :: l)
  | c3 n l (H: R (S n) l) : R n l.
```

Which of the following propositions are provable?

- `R 2 [1;0]`
- `R 1 [1;2;1;0]`
- `R 6 [3;2;1;0]`

```
(* FILL IN HERE *)
□
```

Case Study: Regular Expressions

The `ev` property provides a simple example for illustrating inductive definitions and the basic techniques for reasoning about them, but it is not terribly exciting -- after all, it is equivalent to the two non-inductive definitions of evenness that we had already seen, and does not seem to offer any concrete benefit over them.

To give a better sense of the power of inductive definitions, we now show how to use them to model a classic concept in computer science: *regular expressions*.

Regular expressions are a simple language for describing sets of strings. Their syntax is defined as follows:

```
Inductive reg_exp (T : Type) : Type :=
  | EmptySet
  | EmptyStr
  | Char (t : T)
  | App (r1 r2 : reg_exp T)
  | Union (r1 r2 : reg_exp T)
  | Star (r : reg_exp T).

Arguments EmptySet {T}.
Arguments EmptyStr {T}.
Arguments Char {T} _.
Arguments App {T} _ _.
Arguments Union {T} _ _.
Arguments Star {T} _.
```

Note that this definition is *polymorphic*: Regular expressions in `reg_exp T` describe strings with characters drawn from `T` -- that is, lists of elements of `T`.

(We depart slightly from standard practice in that we do not require the type `T` to be finite. This results in a somewhat different theory of regular expressions, but the difference is not significant for our purposes.)

We connect regular expressions and strings via the following rules, which define when a regular expression *matches* some string:

- The expression `EmptySet` does not match any string.
- The expression `EmptyStr` matches the empty string `[]`.
- The expression `Char x` matches the one-character string `[x]`.
- If re_1 matches s_1, and re_2 matches s_2, then `App` re_1 re_2 matches s_1 `++` s_2.
- If at least one of re_1 and re_2 matches `s`, then `Union` re_1 re_2 matches `s`.
- Finally, if we can write some string `s` as the concatenation of a sequence of strings `s = s_1 ++ ... ++ s_k`, and the expression `re` matches each one of the strings `s_i`, then `Star re` matches `s`.

  In particular, the sequence of strings may be empty, so `Star re` always matches the empty string `[]` no matter what `re` is.

We can easily translate this informal definition into an `Inductive` one as follows. We use the notation `s =~ re` in place of `exp_match s re`; by "reserving" the notation before defining the `Inductive`, we can use it in the definition!

```
Reserved Notation "s =~ re" (at level 80).

Inductive exp_match {T} : list T → reg_exp T → Prop :=
  | MEmpty : [] =~ EmptyStr
  | MChar x : [x] =~ (Char x)
  | MApp s1 re1 s2 re2
             (H1 : s1 =~ re1)
             (H2 : s2 =~ re2)
           : (s1 ++ s2) =~ (App re1 re2)
  | MUnionL s1 re1 re2
                (H1 : s1 =~ re1)
              : s1 =~ (Union re1 re2)
  | MUnionR re1 s2 re2
                (H2 : s2 =~ re2)
              : s2 =~ (Union re1 re2)
  | MStar0 re : [] =~ (Star re)
  | MStarApp s1 s2 re
                 (H1 : s1 =~ re)
                 (H2 : s2 =~ (Star re))
               : (s1 ++ s2) =~ (Star re)
  where "s =~ re" := (exp_match s re).
```

Again, for readability, we display this definition using inference-rule notation.

$$\frac{}{\texttt{[] =~ EmptyStr}}\ \text{(MEmpty)}$$

$$\frac{}{\texttt{[x] =~ Char x}}\ \text{(MChar)}$$

$$\frac{s_1 \texttt{ =~ } re_1 \qquad s_2 \texttt{ =~ } re_2}{s_1 \texttt{ ++ } s_2 \texttt{ =~ App } re_1\ re_2}\ \text{(MApp)}$$

$$\frac{s_1 \texttt{ =~ } re_1}{s_1 \texttt{ =~ Union } re_1\ re_2}\ \text{(MUnionL)}$$

$$\frac{s_2 \texttt{ =~ } re_2}{s_2 \texttt{ =~ Union } re_1\ re_2}\ \text{(MUnionR)}$$

$$\frac{}{\texttt{[] =~ Star re}}\ \text{(MStar0)}$$

$$\frac{s_1 \texttt{ =~ re} \qquad s_2 \texttt{ =~ Star re}}{s_1 \texttt{ ++ } s_2 \texttt{ =~ Star re}}\ \text{(MStarApp)}$$

Notice that these rules are not *quite* the same as the informal ones that we gave at the beginning of the section. First, we don't need to include a rule explicitly stating that no string matches `EmptySet`; we just don't happen to include any rule that would have the effect of some string matching `EmptySet`. (Indeed, the syntax of inductive definitions doesn't even *allow* us to give such a "negative rule.")

Second, the informal rules for `Union` and `Star` correspond to two constructors each: `MUnionL` / `MUnionR`, and `MStar0` / `MStarApp`. The result is logically equivalent to the original rules but more convenient to use in Coq, since the recursive occurrences of `exp_match` are given as direct arguments to the constructors, making it easier to perform induction on evidence. (The $\texttt{exp\_match\_ex}_1$ and $\texttt{exp\_match\_ex}_2$ exercises below ask you to prove that the constructors given in the inductive declaration and the ones that would arise from a more literal transcription of the informal rules are indeed equivalent.)

Let's illustrate these rules with a few examples.

```
Example reg_exp_ex1 : [1] =~ Char 1.
```

```
Example reg_exp_ex2 : [1; 2] =~ App (Char 1) (Char 2).
```

(Notice how the last example applies `MApp` to the string `[1]` directly. Since the goal mentions `[1; 2]` instead of `[1] ++ [2]`, Coq wouldn't be able to figure out how to split the string on its own.)

Using `inversion`, we can also show that certain strings do *not* match a regular expression:

```
Example reg_exp_ex3 : ¬ ([1; 2] =~ Char 1).
```

We can define helper functions for writing down regular expressions. The `reg_exp_of_list` function constructs a regular expression that matches exactly the list that it receives as an argument:

```
Fixpoint reg_exp_of_list {T} (l : list T) :=
  match l with
  | [] ⇒ EmptyStr
  | x :: l' ⇒ App (Char x) (reg_exp_of_list l')
  end.

Example reg_exp_ex4 : [1; 2; 3] =~ reg_exp_of_list [1; 2; 3].
```

We can also prove general facts about `exp_match`. For instance, the following lemma shows that every string `s` that matches `re` also matches `Star re`.

```
Lemma MStar1 :
  ∀ T s (re : reg_exp T) ,
    s =~ re →
    s =~ Star re.
```

(Note the use of `app_nil_r` to change the goal of the theorem to exactly the same shape expected by `MStarApp`.)

Exercise: 3 stars, standard (exp_match_ex1)

The following lemmas show that the informal matching rules given at the beginning of the chapter can be obtained from the formal inductive definition.

```
Lemma empty_is_empty : ∀ T (s : list T),
  ¬ (s =~ EmptySet).
Proof.
  (* FILL IN HERE *) Admitted.

Lemma MUnion' : ∀ T (s : list T) (re1 re2 : reg_exp T),
  s =~ re1 ∨ s =~ re2 →
  s =~ Union re1 re2.
Proof.
  (* FILL IN HERE *) Admitted.
```

The next lemma is stated in terms of the `fold` function from the Poly chapter: If `ss : list (list T)` represents a sequence of strings `s1, ..., sn`, then `fold app ss []` is the result of concatenating them all together.

```
Lemma MStar' : ∀ T (ss : list (list T)) (re : reg_exp T),
  (∀ s, In s ss → s =~ re) →
  fold app ss [] =~ Star re.
Proof.
  (* FILL IN HERE *) Admitted.
□
```

Exercise: 4 stars, standard, optional (reg_exp_of_list_spec)

Prove that `reg_exp_of_list` satisfies the following specification:

```
Lemma reg_exp_of_list_spec : ∀ T (s1 s2 : list T),
  s1 =~ reg_exp_of_list s2 ↔ s1 = s2.
Proof.
  (* FILL IN HERE *) Admitted.
□
```

Since the definition of `exp_match` has a recursive structure, we might expect that proofs involving regular expressions will often require induction on evidence.

For example, suppose that we wanted to prove the following intuitive result: If a regular expression `re` matches some string `s`, then all elements of `s` must occur as character literals somewhere in `re`.

To state this theorem, we first define a function `re_chars` that lists all characters that occur in a regular expression:

```
Fixpoint re_chars {T} (re : reg_exp T) : list T :=
  match re with
  | EmptySet ⇒ []
  | EmptyStr ⇒ []
  | Char x ⇒ [x]
  | App re1 re2 ⇒ re_chars re1 ++ re_chars re2
  | Union re1 re2 ⇒ re_chars re1 ++ re_chars re2
  | Star re ⇒ re_chars re
  end.
```

We can then phrase our theorem as follows:

```
Theorem in_re_match : ∀ T (s : list T) (re : reg_exp T) (x : T),
  s =~ re →
  In x s →
  In x (re_chars re).
Proof.
  intros T s re x Hmatch Hin.
  induction Hmatch
    as [| x'
        | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
        | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
        | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2].
  (* WORKED IN CLASS *)
  - (* MEmpty *)
    simpl in Hin. destruct Hin.
  - (* MChar *)
    simpl. simpl in Hin.
    apply Hin.
  - (* MApp *)
    simpl.
```

Something interesting happens in the `MApp` case. We obtain *two* induction hypotheses: One that applies when `x` occurs in `s1` (which matches `re1`), and a second one that applies when `x` occurs in `s2` (which matches `re2`).

```
    simpl. rewrite In_app_iff in *.
    destruct Hin as [Hin | Hin].
    + (* In x s1 *)
      left. apply (IH1 Hin).
    + (* In x s2 *)
      right. apply (IH2 Hin).
```

```
  - (* MUnionL *)
    simpl. rewrite In_app_iff.
    left. apply (IH Hin).
  - (* MUnionR *)
    simpl. rewrite In_app_iff.
    right. apply (IH Hin).
  - (* MStar0 *)
    destruct Hin.
  - (* MStarApp *)
    simpl.
```

Here again we get two induction hypotheses, and they illustrate why we need induction on evidence for `exp_match`, rather than induction on the regular expression `re`: The latter would only provide an induction hypothesis for strings that match `re`, which would not allow us to reason about the case `In x` s_2.

```
    rewrite In_app_iff in Hin.
    destruct Hin as [Hin | Hin].
    + (* In x s1 *)
      apply (IH1 Hin).
    + (* In x s2 *)
      apply (IH2 Hin).
Qed.
```

Exercise: 4 stars, standard (re_not_empty)

Write a recursive function `re_not_empty` that tests whether a regular expression matches some string. Prove that your function is correct.

```
Fixpoint re_not_empty {T : Type} (re : reg_exp T) : bool
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.

Lemma re_not_empty_correct : ∀ T (re : reg_exp T),
  (∃ s, s =~ re) ↔ re_not_empty re = true.
Proof.
  (* FILL IN HERE *) Admitted.
☐
```

The `remember` Tactic

One potentially confusing feature of the `induction` tactic is that it will let you try to perform an induction over a term that isn't sufficiently general. The effect of this is to lose information (much as `destruct` without an `eqn:` clause can do), and leave you unable to complete the proof. Here's an example:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re : reg_exp T),
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
Proof.
  intros T s1 s2 re H1.
```

Just doing an `inversion` on H_1 won't get us very far in the recursive cases. (Try it!). So we need induction (on evidence!). Here is a naive first attempt:

```
  generalize dependent s2.
  induction H1
    as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
        |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
        |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].
```

But now, although we get seven cases (as we would expect from the definition of `exp_match`), we have lost a very important bit of information from H_1: the fact that s_1 matched something of the form `Star re`. This means that we have to give proofs for *all* seven constructors of this definition, even though all but two of them (`MStar0` and `MStarApp`) are contradictory. We can still get the proof to go through for a few constructors, such as `MEmpty`...

```
  - (* MEmpty *)
    simpl. intros s2 H. apply H.
```

... but most cases get stuck. For `MChar`, for instance, we must show that

```
s2 =~ Char x' → x' :: s2 =~ Char x',
```

which is clearly impossible.

```
  - (* MChar. *) intros s2 H. simpl. (* Stuck... *)
Abort.
```

The problem is that `induction` over a Prop hypothesis only works properly with hypotheses that are completely general, i.e., ones in which all the arguments are variables, as opposed to more complex expressions, such as `Star re`.

(In this respect, `induction` on evidence behaves more like `destruct`-without-`eqn:` than like `inversion`.)

An awkward way to solve this problem is "manually generalizing" over the problematic expressions by adding explicit equality hypotheses to the lemma:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re re' : reg_exp T),
  re' = Star re →
  s1 =~ re' →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
```

We can now proceed by performing induction over evidence directly, because the argument to the first hypothesis is sufficiently general, which means that we can discharge most cases by inverting the `re' = Star re` equality in the context.

This idiom is so common that Coq provides a tactic to automatically generate such equations for us, avoiding thus the need for changing the statements of our theorems.

```
Abort.
```

The tactic `remember e as x` causes Coq to (1) replace all occurrences of the expression `e` by the variable `x`, and (2) add an equation `x = e` to the context. Here's how we can use it to show the above result:

```
Lemma star_app: ∀ T (s1 s2 : list T) (re : reg_exp T),
  s1 =~ Star re →
  s2 =~ Star re →
  s1 ++ s2 =~ Star re.
```

```
Proof.
  intros T s1 s2 re H1.
  remember (Star re) as re'.
```

We now have `Heqre' : re' = Star re`.

```
  generalize dependent s2.
  induction H1
    as [|x'|s1 re1 s2' re2 Hmatch1 IH1 Hmatch2 IH2
        |s1 re1 re2 Hmatch IH|re1 s2' re2 Hmatch IH
        |re''|s1 s2' re'' Hmatch1 IH1 Hmatch2 IH2].
```

The `Heqre'` is contradictory in most cases, allowing us to conclude immediately.

```
  - (* MEmpty *) discriminate.
  - (* MChar *) discriminate.
  - (* MApp *) discriminate.
  - (* MUnionL *) discriminate.
  - (* MUnionR *) discriminate.
```

The interesting cases are those that correspond to `Star`. Note that the induction hypothesis `IH2` on the `MStarApp` case mentions an additional premise `Star re'' = Star re`, which results from the equality generated by `remember`.

```
  - (* MStar0 *)
    injection Heqre' as Heqre''. intros s H. apply H.

  - (* MStarApp *)
    injection Heqre' as Heqre''.
    intros s2 H1. rewrite <- app_assoc.
    apply MStarApp.
    + apply Hmatch1.
    + apply IH2.
      × rewrite Heqre''. reflexivity.
      × apply H1.
Qed.
```

Exercise: 4 stars, standard, optional (exp_match_ex2)

The `MStar''` lemma below (combined with its converse, the `MStar'` exercise above), shows that our definition of `exp_match` for `Star` is equivalent to the informal one given previously.

```
Lemma MStar'' : ∀ T (s : list T) (re : reg_exp T),
  s =~ Star re →
  ∃ ss : list (list T),
    s = fold app ss []
    ∧ ∀ s', In s' ss → s' =~ re.
Proof.
  (* FILL IN HERE *) Admitted.
☐
```

Exercise: 5 stars, advanced (weak_pumping)

One of the first really interesting theorems in the theory of regular expressions is the so-called *pumping lemma*, which states, informally, that any sufficiently long string `s` matching a regular expression `re` can be "pumped" by repeating some middle section of `s` an arbitrary number of times to produce a new string also matching `re`. (For the sake of simplicity in this exercise, we consider a slightly weaker theorem than is usually stated in courses on automata theory.)

To get started, we need to define "sufficiently long." Since we are working in a constructive logic, we actually need to be able to calculate, for each regular expression `re`, the minimum length for strings `s` to guarantee "pumpability."

```
Module Pumping.

Fixpoint pumping_constant {T} (re : reg_exp T) : nat :=
  match re with
  | EmptySet ⇒ 1
  | EmptyStr ⇒ 1
  | Char _ ⇒ 2
  | App re1 re2 ⇒
      pumping_constant re1 + pumping_constant re2
  | Union re1 re2 ⇒
      pumping_constant re1 + pumping_constant re2
  | Star r ⇒ pumping_constant r
  end.
```

You may find these lemmas about the pumping constant useful when proving the pumping lemma below.

```
Lemma pumping_constant_ge_1 :
  ∀ T (re : reg_exp T),
    pumping_constant re ≥ 1.

Lemma pumping_constant_0_false :
  ∀ T (re : reg_exp T),
    pumping_constant re = 0 → False.
```

Next, it is useful to define an auxiliary function that repeats a string (appends it to itself) some number of times.

```
Fixpoint napp {T} (n : nat) (l : list T) : list T :=
  match n with
  | 0 ⇒ []
  | S n' ⇒ l ++ napp n' l
  end.
```

This auxiliary lemma might also be useful in your proof of the pumping lemma.

```
Lemma napp_plus: ∀ T (n m : nat) (l : list T),
  napp (n + m) l = napp n l ++ napp m l.

Lemma napp_star :
  ∀ T m s1 s2 (re : reg_exp T),
    s1 =~ re → s2 =~ Star re →
    napp m s1 ++ s2 =~ Star re.
```

The (weak) pumping lemma itself says that, if `s =~ re` and if the length of `s` is at least the pumping constant of `re`, then `s` can be split into three substrings `s1 ++ s2 ++ s3` in such a way that `s2` can be repeated any number of times and the result, when combined with s_1 and s_3 will still match re. Since s_2 is also guaranteed not to be the empty string, this gives us a (constructive!) way to generate strings matching re that are as long as we like.

```
Lemma weak_pumping : ∀ T (re : reg_exp T) s,
  s =~ re →
  pumping_constant re ≤ length s →
  ∃ s1 s2 s3,
    s = s1 ++ s2 ++ s3 ∧
    s2 ≠ [] ∧
    ∀ m, s1 ++ napp m s2 ++ s3 =~ re.
```

You are to fill in the proof. Several of the lemmas about le that were in an optional exercise earlier in this chapter may be useful.

```
Proof.
  intros T re s Hmatch.
  induction Hmatch
    as [ | x | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
       | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
       | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2 ].
  - (* MEmpty *)
    simpl. intros contra. inversion contra.
  (* FILL IN HERE *) Admitted.
☐
```

Exercise: 5 stars, advanced, optional (pumping)

Now here is the usual version of the pumping lemma. In addition to requiring that s_2 ≠ [], it also requires that length s_1 + length s_2 ≤ pumping_constant re.

```
Lemma pumping : ∀ T (re : reg_exp T) s,
  s =~ re →
  pumping_constant re ≤ length s →
  ∃ s1 s2 s3,
    s = s1 ++ s2 ++ s3 ∧
    s2 ≠ [] ∧
    length s1 + length s2 ≤ pumping_constant re ∧
    ∀ m, s1 ++ napp m s2 ++ s3 =~ re.
```

You may want to copy your proof of weak_pumping below.

```
Proof.
  intros T re s Hmatch.
  induction Hmatch
    as [ | x | s1 re1 s2 re2 Hmatch1 IH1 Hmatch2 IH2
       | s1 re1 re2 Hmatch IH | re1 s2 re2 Hmatch IH
       | re | s1 s2 re Hmatch1 IH1 Hmatch2 IH2 ].
  - (* MEmpty *)
    simpl. intros contra. inversion contra.
  (* FILL IN HERE *) Admitted.

End Pumping.
☐
```

Case Study: Improving Reflection

We've seen in the Logic chapter that we often need to relate boolean computations to statements in Prop. But performing this conversion as we did it there can result in tedious proof scripts. Consider the proof of the following theorem:

```
Theorem filter_not_empty_In : ∀ n l,
  filter (fun x ⇒ n =? x) l ≠ [] →
  In n l.
Proof.
  intros n l. induction l as [|m l' IHl'].
  - (* l =  *)
    simpl. intros H. apply H. reflexivity.
  - (* l = m :: l' *)
    simpl. destruct (n =? m) eqn:H.
    + (* n =? m = true *)
      intros _. rewrite eqb_eq in H. rewrite H.
      left. reflexivity.
    + (* n =? m = false *)
      intros H'. right. apply IHl'. apply H'.
Qed.
```

In the first branch after destruct, we explicitly apply the eqb_eq lemma to the equation generated by destructing n =? m, to convert the assumption n =? m = true into the assumption n = m; then we had to rewrite using this assumption to complete the case.

We can streamline this by defining an inductive proposition that yields a better case-analysis principle for n =? m. Instead of generating an equation such as (n =? m) = true, which is generally not directly useful, this principle gives us right away the assumption we really need: n = m.

```
Inductive reflect (P : Prop) : bool → Prop :=
| ReflectT (H : P) : reflect P true
| ReflectF (H : ¬ P) : reflect P false.
```

The reflect property takes two arguments: a proposition P and a boolean b. Intuitively, it states that the property P is *reflected* in (i.e., equivalent to) the boolean b: that is, P holds if and only if b = true. To see this, notice that, by definition, the only way we can produce evidence for reflect P true is by showing P and then using the ReflectT constructor. If we invert this statement, this means that it should be possible to extract evidence for P from a proof of reflect P true. Similarly, the only way to show reflect P false is by combining evidence for ¬ P with the ReflectF constructor.

It is easy to formalize this intuition and show that the statements P ↔ b = true and reflect P b are indeed equivalent. First, the left-to-right implication:

```
Theorem iff_reflect : ∀ P b, (P ↔ b = true) → reflect P b.
Proof.
  (* WORKED IN CLASS *)
  intros P b H. destruct b.
  - apply ReflectT. rewrite H. reflexivity.
  - apply ReflectF. rewrite H. intros H'. discriminate.
Qed.
```

Now you prove the right-to-left implication:

Exercise: 2 stars, standard, especially useful (reflect_iff)

```
Theorem reflect_iff : ∀ P b, reflect P b → (P ↔ b = true).
Proof.
  (* FILL IN HERE *) Admitted.
☐
```

The advantage of `reflect` over the normal "if and only if" connective is that, by destructing a hypothesis or lemma of the form `reflect P b`, we can perform case analysis on `b` while at the same time generating appropriate hypothesis in the two branches (`P` in the first subgoal and `¬ P` in the second).

```
Lemma eqbP : ∀ n m, reflect (n = m) (n =? m).
Proof.
  intros n m. apply iff_reflect. rewrite eqb_eq. reflexivity.
Qed.
```

A smoother proof of `filter_not_empty_In` now goes as follows. Notice how the calls to `destruct` and `rewrite` are combined into a single call to `destruct`.

(To see this clearly, look at the two proofs of `filter_not_empty_In` with Coq and observe the differences in proof state at the beginning of the first case of the `destruct`.)

```
Theorem filter_not_empty_In' : ∀ n l,
  filter (fun x ⇒ n =? x) l ≠ [] →
  In n l.
Proof.
  intros n l. induction l as [|m l' IHl'].
  - (* l =  *)
    simpl. intros H. apply H. reflexivity.
  - (* l = m :: l' *)
    simpl. destruct (eqbP n m) as [H | H].
    + (* n = m *)
      intros _. rewrite H. left. reflexivity.
    + (* n <> m *)
      intros H'. right. apply IHl'. apply H'.
Qed.
```

Exercise: 3 stars, standard, especially useful (eqbP_practice)

Use `eqbP` as above to prove the following:

```
Fixpoint count n l :=
  match l with
  | [] ⇒ 0
  | m :: l' ⇒ (if n =? m then 1 else 0) + count n l'
  end.

Theorem eqbP_practice : ∀ n l,
  count n l = 0 → ~(In n l).
Proof.
  (* FILL IN HERE *) Admitted.
☐
```

This small example shows how reflection gives us a small gain in convenience; in larger developments, using `reflect` consistently can often lead to noticeably shorter and clearer proof scripts. We'll see many more examples in later chapters and in *Programming Language Foundations*.

The use of the `reflect` property has been popularized by *SSReflect*, a Coq library that has been used to formalize important results in mathematics, including as the 4-color theorem and the Feit-Thompson theorem. The name SSReflect stands for *small-scale reflection*, i.e., the pervasive use of reflection to simplify small proof steps with boolean computations.

Additional Exercises

Exercise: 3 stars, standard, especially useful (nostutter_defn)

Formulating inductive definitions of properties is an important skill you'll need in this course. Try to solve this exercise without any help at all.

We say that a list "stutters" if it repeats the same element consecutively. (This is different from not containing duplicates: the sequence `[1;4;1]` repeats the element `1` but does not stutter.) The property "`nostutter mylist`" means that `mylist` does not stutter. Formulate an inductive definition for `nostutter`.

```
Inductive nostutter {X:Type} : list X → Prop :=
 (* FILL IN HERE *)
.
```

Make sure each of these tests succeeds, but feel free to change the suggested proof (in comments) if the given one doesn't work for you. Your definition might be different from ours and still be correct, in which case the examples might need a different proof. (You'll notice that the suggested proofs use a number of tactics we haven't talked about, to make them more robust to different possible ways of defining `nostutter`. You can probably just uncomment and use them as-is, but you can also prove each example with more basic tactics.)

```
Example test_nostutter_1: nostutter [3;1;4;1;5;6].
(* FILL IN HERE *) Admitted.
(*
  Proof. repeat constructor; apply eqb_neq; auto.
  Qed.
*)

Example test_nostutter_2: nostutter (@nil nat).
(* FILL IN HERE *) Admitted.
(*
  Proof. repeat constructor; apply eqb_neq; auto.
  Qed.
*)

Example test_nostutter_3: nostutter [5].
(* FILL IN HERE *) Admitted.
(*
  Proof. repeat constructor; auto. Qed.
*)

Example test_nostutter_4: not (nostutter [3;1;1;4]).
(* FILL IN HERE *) Admitted.
(*
  Proof. intro.
  repeat match goal with
    h: nostutter _ ⊢ _ => inversion h; clear h; subst
  end.
  contradiction; auto. Qed.
*)
```

```
(* Do not modify the following line: *)
Definition manual_grade_for_nostutter : option (nat×string) := None.
☐
```

Exercise: 4 stars, advanced (filter_challenge)

Let's prove that our definition of `filter` from the `Poly` chapter matches an abstract specification. Here is the specification, written out informally in English:

A list `l` is an "in-order merge" of l_1 and l_2 if it contains all the same elements as l_1 and l_2, in the same order as l_1 and l_2, but possibly interleaved. For example,

```
[1;4;6;2;3]
```

is an in-order merge of

```
[1;6;2]
```

and

```
[4;3].
```

Now, suppose we have a set `X`, a function `test: X→bool`, and a list `l` of type `list X`. Suppose further that `l` is an in-order merge of two lists, l_1 and l_2, such that every item in l_1 satisfies `test` and no item in l_2 satisfies test. Then `filter test l` = l_1.

Translate this specification into a Coq theorem and prove it. (You'll need to begin by defining what it means for one list to be a merge of two others. Do this with an inductive relation, not a `Fixpoint`.)

```
(* FILL IN HERE *)

(* Do not modify the following line: *)
Definition manual_grade_for_filter_challenge : option (nat×string) := None.
☐
```

Exercise: 5 stars, advanced, optional (filter_challenge_2)

A different way to characterize the behavior of `filter` goes like this: Among all subsequences of `l` with the property that `test` evaluates to `true` on all their members, `filter test l` is the longest. Formalize this claim and prove it.

```
(* FILL IN HERE *)
☐
```

Exercise: 4 stars, standard, optional (palindromes)

A palindrome is a sequence that reads the same backwards as forwards.

- Define an inductive proposition `pal` on `list X` that captures what it means to be a palindrome. (Hint: You'll need three cases. Your definition should be based on the structure of the list; just having a single constructor like
  ```
  c : ∀ l, l = rev l → pal l
  ```
  may seem obvious, but will not work very well.)
- Prove (`pal_app_rev`) that
  ```
  ∀ l, pal (l ++ rev l).
  ```
- Prove (`pal_rev` that)
  ```
  ∀ l, pal l → l = rev l.
  ```

```
(* FILL IN HERE *)

(* Do not modify the following line: *)
Definition manual_grade_for_pal_pal_app_rev_pal_rev : option (nat×string) := None.
☐
```

Exercise: 5 stars, standard, optional (palindrome_converse)

Again, the converse direction is significantly more difficult, due to the lack of evidence. Using your definition of `pal` from the previous exercise, prove that

```
∀ l, l = rev l → pal l.
```

```
(* FILL IN HERE *)
☐
```

Exercise: 4 stars, advanced, optional (NoDup)

Recall the definition of the `In` property from the `Logic` chapter, which asserts that a value `x` appears at least once in a list `l`:

```
(* Fixpoint In (A : Type) (x : A) (l : list A) : Prop :=
   match l with
   |  => False
   | x' :: l' => x' = x \/ In A x l'
   end *)
```

Your first task is to use `In` to define a proposition `disjoint X` l_1 l_2, which should be provable exactly when l_1 and l_2 are lists (with elements of type X) that have no elements in common.

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

Next, use `In` to define an inductive proposition `NoDup X l`, which should be provable exactly when `l` is a list (with elements of type `X`) where every member is different from every other. For example, `NoDup nat [1;2;3;4]` and `NoDup bool []` should be provable, while `NoDup nat [1;2;1]` and `NoDup bool [true;true]` should not be.

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

Finally, state and prove one or more interesting theorems relating `disjoint`, `NoDup` and `++` (list append).

```
(* FILL IN HERE *)

(* Do not modify the following line: *)
Definition manual_grade_for_NoDup_disjoint_etc : option (nat×string) := None.
☐
```

Exercise: 4 stars, advanced, optional (pigeonhole_principle)

The *pigeonhole principle* states a basic fact about counting: if we distribute more than `n` items into `n` pigeonholes, some pigeonhole must contain at least two items. As often happens, this apparently trivial fact about numbers requires non-trivial machinery to prove, but we now have enough...

First prove an easy useful lemma.

```
Lemma in_split : ∀ (X:Type) (x:X) (l:list X),
  In x l →
  ∃ l1 l2, l = l1 ++ x :: l2.
Proof.
  (* FILL IN HERE *) Admitted.
```

Now define a property `repeats` such that `repeats X l` asserts that `l` contains at least one repeated element (of type `X`).

```
Inductive repeats {X:Type} : list X → Prop :=
  (* FILL IN HERE *)
.

(* Do not modify the following line: *)
Definition manual_grade_for_check_repeats : option (nat×string) := None.
```

Now, here's a way to formalize the pigeonhole principle. Suppose list `l2` represents a list of pigeonhole labels, and list `l1` represents the labels assigned to a list of items. If there are more items than labels, at least two items must have the same label -- i.e., list `l1` must contain repeats.

This proof is much easier if you use the `excluded_middle` hypothesis to show that `In` is decidable, i.e., `∀ x l, (In x l) ∨ ¬ (In x l)`. However, it is also possible to make the proof go through *without* assuming that `In` is decidable; if you manage to do this, you will not need the `excluded_middle` hypothesis.

```
Theorem pigeonhole_principle: ∀ (X:Type) (l1 l2:list X),
   excluded_middle →
   (∀ x, In x l1 → In x l2) →
   length l2 < length l1 →
   repeats l1.
Proof.
   intros X l1. induction l1 as [|x l1' IHl1'].
  (* FILL IN HERE *) Admitted.
☐
```

Extended Exercise: A Verified Regular-Expression Matcher

We have now defined a match relation over regular expressions and polymorphic lists. We can use such a definition to manually prove that a given regex matches a given string, but it does not give us a program that we can run to determine a match autmatically.

It would be reasonable to hope that we can translate the definitions of the inductive rules for constructing evidence of the match relation into cases of a recursive function that reflects the relation by recursing on a given regex. However, it does not seem straightforward to define such a function in which the given regex is a recursion variable recognized by Coq. As a result, Coq will not accept that the function always terminates.

Heavily-optimized regex matchers match a regex by translating a given regex into a state machine and determining if the state machine accepts a given string. However, regex matching can also be implemented using an algorithm that operates purely on strings and regexes without defining and maintaining additional datatypes, such as state machines. We'll implemement such an algorithm, and verify that its value reflects the match relation.

We will implement a regex matcher that matches strings represented as lists of ASCII characters:

```
Require Import Coq.Strings.Ascii.

Definition string := list ascii.
```

The Coq standard library contains a distinct inductive definition of strings of ASCII characters. However, we will use the above definition of strings as lists as ASCII characters in order to apply the existing definition of the match relation.

We could also define a regex matcher over polymorphic lists, not lists of ASCII characters specifically. The matching algorithm that we will implement needs to be able to test equality of elements in a given list, and thus needs to be given an equality-testing function. Generalizing the definitions, theorems, and proofs that we define for such a setting is a bit tedious, but workable.

The proof of correctness of the regex matcher will combine properties of the regex-matching function with properties of the `match` relation that do not depend on the matching function. We'll go ahead and prove the latter class of properties now. Most of them have straightforward proofs, which have been given to you, although there are a few key lemmas that are left for you to prove.

Each provable `Prop` is equivalent to `True`.

```
Lemma provable_equiv_true : ∀ (P : Prop), P → (P ↔ True).
Proof.
  intros.
  split.
  - intros. constructor.
  - intros _. apply H.
Qed.
```

Each `Prop` whose negation is provable is equivalent to `False`.

```
Lemma not_equiv_false : ∀ (P : Prop), ¬P → (P ↔ False).
Proof.
  intros.
  split.
  - apply H.
  - intros. destruct H0.
Qed.
```

`EmptySet` matches no string.

```
Lemma null_matches_none : ∀ (s : string), (s =~ EmptySet) ↔ False.
Proof.
  intros.
  apply not_equiv_false.
  unfold not. intros. inversion H.
Qed.
```

`EmptyStr` only matches the empty string.

```
Lemma empty_matches_eps : ∀ (s : string), s =~ EmptyStr ↔ s = [ ].
Proof.
  split.
  - intros. inversion H. reflexivity.
  - intros. rewrite H. apply MEmpty.
Qed.
```

`EmptyStr` matches no non-empty string.

```
Lemma empty_nomatch_ne : ∀ (a : ascii) s, (a :: s =~ EmptyStr) ↔ False.
Proof.
  intros.
  apply not_equiv_false.
  unfold not. intros. inversion H.
Qed.
```

`Char a` matches no string that starts with a non-`a` character.

```
Lemma char_nomatch_char :
  ∀ (a b : ascii) s, b ≠ a → (b :: s =~ Char a ↔ False).
Proof.
  intros.
  apply not_equiv_false.
  unfold not.
  intros.
  apply H.
  inversion H0.
  reflexivity.
Qed.
```

If `Char a` matches a non-empty string, then the string's tail is empty.

```
Lemma char_eps_suffix : ∀ (a : ascii) s, a :: s =~ Char a ↔ s = [ ].
Proof.
  split.
  - intros. inversion H. reflexivity.
  - intros. rewrite H. apply MChar.
Qed.
```

`App re0 re1` matches string `s` iff `s = s0 ++ s1`, where `s0` matches `re0` and `s1` matches `re1`.

```
Lemma app_exists : ∀ (s : string) re0 re1,
    s =~ App re0 re1 ↔
    ∃ s0 s1, s = s0 ++ s1 ∧ s0 =~ re0 ∧ s1 =~ re1.
Proof.
  intros.
  split.
  - intros. inversion H. ∃ s1, s2. split.
    × reflexivity.
    × split. apply H3. apply H4.
  - intros [ s0 [ s1 [ Happ [ Hmat0 Hmat1 ] ] ] ].
    rewrite Happ. apply (MApp s0 _ s1 _ Hmat0 Hmat1).
Qed.
```

Exercise: 3 stars, standard, optional (app_ne)

`App re0 re1` matches `a::s` iff `re0` matches the empty string and `a::s` matches `re1` or `s=s0++s1`, where `a::s0` matches `re0` and `s1` matches `re1`.

Even though this is a property of purely the match relation, it is a critical observation behind the design of our regex matcher. So (1) take time to understand it, (2) prove it, and (3) look for how you'll use it later.

```
Lemma app_ne : ∀ (a : ascii) s re0 re1,
    a :: s =~ (App re0 re1) ↔
    ([ ] =~ re0 ∧ a :: s =~ re1) ∨
    ∃ s0 s1, s = s0 ++ s1 ∧ a :: s0 =~ re0 ∧ s1 =~ re1.
Proof.
  (* FILL IN HERE *) Admitted.
☐
```

`s` matches `Union re0 re1` iff `s` matches `re0` or `s` matches `re1`.

```
Lemma union_disj : ∀ (s : string) re0 re1,
    s =~ Union re0 re1 ↔ s =~ re0 ∨ s =~ re1.
Proof.
  intros. split.
  - intros. inversion H.
    + left. apply H2.
    + right. apply H1.
  - intros [ H | H ].
    + apply MUnionL. apply H.
    + apply MUnionR. apply H.
Qed.
```

Exercise: 3 stars, standard, optional (star_ne)

`a::s` matches `Star re` iff `s = s0 ++ s1`, where `a::s0` matches `re` and `s1` matches `Star re`. Like `app_ne`, this observation is critical, so understand it, prove it, and keep it in mind.

Hint: you'll need to perform induction. There are quite a few reasonable candidates for `Prop`'s to prove by induction. The only one that will work is splitting the `iff` into two implications and proving one by induction on the evidence for `a :: s =~ Star re`. The other implication can be proved without induction.

In order to prove the right property by induction, you'll need to rephrase `a :: s =~ Star re` to be a `Prop` over general variables, using the `remember` tactic.

```
Lemma star_ne : ∀ (a : ascii) s re,
    a :: s =~ Star re ↔
    ∃ s0 s1, s = s0 ++ s1 ∧ a :: s0 =~ re ∧ s1 =~ Star re.
Proof.
  (* FILL IN HERE *) Admitted.
☐
```

The definition of our regex matcher will include two fixpoint functions. The first function, given regex `re`, will evaluate to a value that reflects whether `re` matches the empty string. The function will satisfy the following property:

```
Definition refl_matches_eps m :=
  ∀ re : reg_exp ascii, reflect ([ ] =~ re) (m re).
```

Exercise: 2 stars, standard, optional (match_eps)

Complete the definition of `match_eps` so that it tests if a given regex matches the empty string:

```
Fixpoint match_eps (re: reg_exp ascii) : bool
  (* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.
☐
```

Exercise: 3 stars, standard, optional (match_eps_refl)

Now, prove that `match_eps` indeed tests if a given regex matches the empty string. (Hint: You'll want to use the reflection lemmas `ReflectT` and `ReflectF`.)

```
Lemma match_eps_refl : refl_matches_eps match_eps.
Proof.
```

```
    (* FILL IN HERE *) Admitted.
  □
```

We'll define other functions that use `match_eps`. However, the only property of `match_eps` that you'll need to use in all proofs over these functions is `match_eps_refl`.

The key operation that will be performed by our regex matcher will be to iteratively construct a sequence of regex derivatives. For each character `a` and regex `re`, the derivative of `re` on `a` is a regex that matches all suffixes of strings matched by `re` that start with `a`. I.e., `re'` is a derivative of `re` on `a` if they satisfy the following relation:

```
  Definition is_der re (a : ascii) re' :=
    ∀ s, a :: s =~ re ↔ s =~ re'.
```

A function `d` derives strings if, given character `a` and regex `re`, it evaluates to the derivative of `re` on `a`. I.e., `d` satisfies the following property:

```
  Definition derives d := ∀ a re, is_der re a (d a re).
```

Exercise: 3 stars, standard, optional (derive)

Define `derive` so that it derives strings. One natural implementation uses `match_eps` in some cases to determine if key regex's match the empty string.

```
  Fixpoint derive (a : ascii) (re : reg_exp ascii) : reg_exp ascii
    (* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.
  □
```

The `derive` function should pass the following tests. Each test establishes an equality between an expression that will be evaluated by our regex matcher and the final value that must be returned by the regex matcher. Each test is annotated with the match fact that it reflects.

```
  Example c := ascii_of_nat 99.
  Example d := ascii_of_nat 100.
```

"c" =~ EmptySet:

```
  Example test_der0 : match_eps (derive c (EmptySet)) = false.
  Proof.
    (* FILL IN HERE *) Admitted.
```

"c" =~ Char c:

```
  Example test_der1 : match_eps (derive c (Char c)) = true.
  Proof.
    (* FILL IN HERE *) Admitted.
```

"c" =~ Char d:

```
  Example test_der2 : match_eps (derive c (Char d)) = false.
  Proof.
    (* FILL IN HERE *) Admitted.
```

"c" =~ App (Char c) EmptyStr:

```
  Example test_der3 : match_eps (derive c (App (Char c) EmptyStr)) = true.
  Proof.
    (* FILL IN HERE *) Admitted.
```

"c" =~ App EmptyStr (Char c):

```
  Example test_der4 : match_eps (derive c (App EmptyStr (Char c))) = true.
  Proof.
    (* FILL IN HERE *) Admitted.
```

"c" =~ Star c:

```
  Example test_der5 : match_eps (derive c (Star (Char c))) = true.
  Proof.
    (* FILL IN HERE *) Admitted.
```

"cd" =~ App (Char c) (Char d):

```
  Example test_der6 :
    match_eps (derive d (derive c (App (Char c) (Char d)))) = true.
  Proof.
    (* FILL IN HERE *) Admitted.
```

"cd" =~ App (Char d) (Char c):

```
  Example test_der7 :
    match_eps (derive d (derive c (App (Char d) (Char c)))) = false.
  Proof.
    (* FILL IN HERE *) Admitted.
```

Exercise: 4 stars, standard, optional (derive_corr)

Prove that `derive` in fact always derives strings.

Hint: one proof performs induction on `re`, although you'll need to carefully choose the property that you prove by induction by generalizing the appropriate terms.

Hint: if your definition of `derive` applies `match_eps` to a particular regex `re`, then a natural proof will apply `match_eps_refl` to `re` and destruct the result to generate cases with assumptions that the `re` does or does not match the empty string.

Hint: You can save quite a bit of work by using lemmas proved above. In particular, to prove many cases of the induction, you can rewrite a `Prop` over a complicated regex (e.g., `s =~ Union` re_0 re_1) to a Boolean combination of `Prop`'s over simple regex's (e.g., `s =~` re_0 ∨ `s =~` re_1) using lemmas given above that are logical equivalences. You can then reason about these `Prop`'s naturally using `intro` and `destruct`.

```
  Lemma derive_corr : derives derive.
  Proof.
    (* FILL IN HERE *) Admitted.
  □
```

We'll define the regex matcher using `derive`. However, the only property of `derive` that you'll need to use in all proofs of properties of the matcher is `derive_corr`.

A function `m` matches regexes if, given string `s` and regex `re`, it evaluates to a value that reflects whether `s` is matched by `re`. I.e., `m` holds the following property:

```
  Definition matches_regex m : Prop :=
    ∀ (s : string) re, reflect (s =~ re) (m s re).
```

Exercise: 2 stars, standard, optional (regex_match)

Complete the definition of `regex_match` so that it matches regexes.

```
  Fixpoint regex_match (s : string) (re : reg_exp ascii) : bool
    (* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.
  □
```

Exercise: 3 stars, standard, optional (regex_refl)

Finally, prove that `regex_match` in fact matches regexes.

Hint: if your definition of `regex_match` applies `match_eps` to regex `re`, then a natural proof applies `match_eps_refl` to `re` and destructs the result to generate cases in which you may assume that `re` does or does not match the empty string.

Hint: if your definition of `regex_match` applies `derive` to character `x` and regex `re`, then a natural proof applies `derive_corr` to `x` and `re` to prove that `x :: s =~ re` given `s =~ derive x re`, and vice versa.

```
Theorem regex_refl : matches_regex regex_match.
Proof.
  (* FILL IN HERE *) Admitted.
☐

(* 2020-09-09 20:51 *)
```