**Inductive** introduces a new inductive type.

"A : B" means "A has type B". So the type bool has type Set!

```coq
Inductive bool : Set :=
| true : bool| false : bool.
```

notb is a function that takes a bool and returns its negation.

Think of "match" like a case or switch statement.

Try removing the ": true" line and see what happens.

Coq's pattern-matching is required to be "exhaustive".

```coq
Definition notb (b : bool) : bool :=
match b
with
| true => false    | false => true  end.
```

andb returns the conjunction of b1 and b2.

```coq
Definition andb (b1 : bool) (b2 : bool) : bool :=
match b1
with
| true => b2    | false => false  end.
```

Let's try to *prove* that andb is commutative.

```coq
Lemma andb_comm :
forall b1 b2, andb b1 b2 = andb b2 b1.
Proof.
  intro x. (** assume an arbitrary boolean, and call it x *)
  intro y. (** assume an arbitrary boolean, and call it y *)
  destruct x. (** case analysis on x *)
  destruct y. (** case analysis on y *)
  + reflexivity. (** the goal is an equals sign with the same thing on both sides *)
  + simpl. (** simplify the goal by running the andb function *)
  reflexivity.
  destruct y. + simpl. reflexivity.
Qed.
```

This means that a multi-argument function is actually
a single-argument function that "returns" another function.

So

```coq
Inductive nat : Set :=
```

isZero checks to see if a natural number is, well, zero.

"_" is used as a variable name to indicate to Coq (and readers!)
that the argument it names will not be used.

```coq
Definition isZero (n : nat) : bool :=
match n
with
| O => true    | S _ => false  end.
```

**Lemma**

```coq
Lemma isZero_O :
isZero O = true.
Proof.
simpl.
reflexivity.
Qed.
```

Let's try to define addition

```coq
Fail
Definition add (n1 : nat) (n2 : nat) : nat :=
match n1
with
| O => n2    | S m1 => add m1 n2  end.
```

This fails with something like:

"The reference add was not found in the current environment."

This is because when we use "Definition", the thing we're defining
isn't available in the body of the definition.

Let's try again using "Fixpoint". Fixpoint will be how we define recursive fun-
ctions.

In Coq, recursive functions are guaranteed to terminate,
so Coq checks that recursive arguments are "smaller".

```coq
Fixpoint add (n1 : nat) (n2 : nat) : nat :=
match n1 with
| O => n2    | S m1 => add m1 (S n2)  end.
```

**Lemma**

```coq
Lemma O_add :
forall n, add O n = n.
Proof.
simpl.
reflexivity.
Qed.
```

```coq
Lemma add_O :
forall n, add n O = n.
Proof.
simpl.
reflexivity.
Abort.
```

**Check <term>** prints the "type" of <term> *

Check andb. (** andb : bool -> bool -> bool *)

Someone asked about how "andb" corresponds to this type.

Here's a "desugared" version of andb that should make this clearer. *)

```coq
Definition andb' : bool -> bool -> bool :=
fun b1 => fun b2 => match b1 with
| true => b2    | false => false  end.
```

Note that multi-argument functions in Coq are "curried".

**Check <term>** prints the "type" of <term> *

Check (andb true). (bool -> bool)

Inductive bool : Set :=
| true : bool| false : bool.
```

Addition is defined recursively:

```coq
Definition add (n1 : nat) (n2 : nat) : nat :=
match n1
with
| O => n2    | S m1 => S (add m1 n2)  end.
```

**Lemma**

```coq
Lemma O_add :
forall n, add O n = n.
Proof.
simpl.
reflexivity.
Qed.
```

**Lemma**

```coq
Lemma add_O :
forall n, add n O = n.
Proof.
simpl.
reflexivity.
Abort.
```

**Check <term>** prints the "type" of <term> *

Check andb. (** andb : bool -> bool -> bool *)

Someone asked about how "andb" corresponds to this type.

Here's a "desugared" version of andb that should make this clearer. *)

```coq
Definition andb' (b1 : bool) (b2 : bool) : bool :=
match b1
with
| true => b2    | false => false  end.
```

Note that multi-argument functions in Coq are "curried".
add n O = n.
Proof.
  intro n.
  destruct n.
  - (** n is O *) reflexivity.
  - (** n is S n (for some other n) *)
    destruct n.
      + simpl. reflexivity.
      + simpl. (** starting to get worried, here *)
        destruct n.
        * simpl. reflexivity.
        * simpl. (** Seems like we’re going to need a different strategy *)
Abort. (** Let’s try *induction*. *)
Lemma add_O :
  forall n, add n O = n.
Proof.
  intro n.
  induction n.
  - simpl. reflexivity.
  - simpl. (** Seems like we’re going to need a different strategy *)
    rewrite IHn. (** find the left-hand side of IHn in the goal and replace it by the right-hand side *)
    reflexivity.
Qed
(** in class, Zach first defined add as follows: *)
Fixpoint add' (n1 : nat) (n2 : nat) : nat :=
  match n1 with
  | O => n2
  | S m1 => add' m1 (S n2)
  end.
(** Optional exercise: complete the following proof. *)
Lemma S_add'_add'_S :
  forall x y, add' x (S y) = S (add' x y).
Proof. (** at some point in the proof, rewrite by S_add'_add'_S *)
  induction x.
  - intros. reflexivity.
  - intros. simpl. rewrite IHx. reflexivity.
Qed.
Lemma add_add' : (** at some point in the proof, rewrite by S_add'_add'_S *)
  forall x y, add x y = add' x y.
Proof. (** FILL IN PROOF HERE *)
Admitted.
Inductive list (A : Set) : Set :=
  nil : list A
  cons : A -> list A -> list A.
(** We’ll do more list stuff next time *)