```
(** * Lecture 01 *)

(** "Inductive" introduces a new inductive type.
    "A : B" means "A has type B". So the type bool has type Set!
*)
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 "| false => true" line and see what happens.
   coq's pattern-matching is required to be *exhaustive*.
 *)
Definition notb (b : bool) : bool :=
  match b with
    | true => false
    | false => true
  end.

(**
  andb returns the conjunction of b1 and b2.
*)
Definition andb (b1 : bool) (b2 : bool) : bool :=
  match b1 with
    | true => b2
    | false => false
  end.

(**
  Let's try to *prove* that andb is commutative.
*)
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 si
des *)
    + simpl. (** simplify the goal by running the andb function *)
      reflexivity.
  - destruct y.
    + simpl. reflexivity.
    + reflexivity.
Qed.

(** Here is a shorter version of the same proof.
   Don't worry about understanding it for now. *)
Lemma andb_comm' :
  forall b1 b2,
    andb b1 b2 = andb b2 b1.
Proof.
  destruct b1; destruct b2; auto.
Qed.

(** "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. *)
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".
```



```
   This means that a multi-argument function is actually
    a single-argument function that *returns* another function.
   So
     andb : bool -> bool -> bool
   is actually
     andb : bool -> (bool -> bool)  *)
Check (andb true). (** bool -> bool *)

Inductive nat : Set :=
| O : nat
| S : nat -> nat.

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

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

(** Let's try to define addition *)
Fail Definition add (n1 : nat) (n2 : nat) : nat :=
  match n1 with
    | O => n2
    | S m1 => add m1 (S 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 fu
nctions.
   In Coq, recursive functions are guaranteed to terminate,
    so Coq checks that recursive arguments are *smaller*.
*)
Fixpoint add (n1 : nat) (n2 : nat) : nat :=
  match n1 with
    | O => n2
    | S m1 => S (add m1 n2)
  end.

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

Lemma add_O :
  forall n,
    add n O = n.
Proof.
  intro n.
  simpl. (** simpl doesn't do anything. *)
Abort.
(** Coq can't simplify "add n O". Let's try case analysis with "destruct" *)
Lemma add_O :
  forall n,
```

```
    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.
    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.
  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 *)
```