(* Lecture 02 *)

(** infer some type arguments automatically *)

Set Implicit Arguments.

Inductive list (A: Set) : Set :=  
  | nil : list A  |
  | cons : A -> list A -> list A. 

Fixpoint length (A: Set) (l: list A) : nat :=  
  match l with  |
  | nil => O  |
  | cons x xs => S (length xs)  end. (*
  (** so far, Coq will not infer the type argument for nil: <<
  Check (cons 1 nil). Error: The term "nil" has type "forall A : Set, list A" while it is expected to have type "list nat".
  >>
  *)

Check (cons 1 (nil nat)). (*
  we can tell Coq to always try though *)

Arguments nil {A}.
Check (cons 1 nil).

(** we can tell Coq to always try though *)

Fixpoint countdown (n: nat) :=  
  match n with  |
  | O => cons n nil  |
  | S m => cons n (countdown m)  end.
Eval cbv in (countdown 0).
Eval cbv in (countdown 3).
Eval cbv in (countdown 10).

Fixpoint map (A B C: Set) (f: A -> B) (l: list A) : list B :=  
  match l with  |
  | nil => nil  |
  | cons x xs => cons (f x) (map f xs)  end.
Eval cbv in (map (plus 1) (countdown 3)).
Eval cbv in (map (fun _ => true) (countdown 3)).

Definition is_zero (n: nat) : bool :=  
  match n with  |
  | O => true  |
  | S m => false  end.
Eval cbv in (is_zero (countdown 3)).

Fixpoint is_even (n: nat) : bool :=  
  match n with  |
  | O => true  |
  | S 0 => false  |
  | S (S m) => is_even m  end.
Eval cbv in (is_even (countdown 3)).

(** Note that this proof uses bullets (*).
  See the course web page for more information about bullets. *)

Lemma map_length:  
  forall (A B C: Set) (f: A -> B) (l: list A),  
  length (map f l) = length l.
Proof.  intros.  induction 1.  + simpl. reflexivity.  + simpl. (** Replace "length (map f l)" with "length l" *")
  rewrite IHl.  reflexivity. Qed.

(* Induction is what we use to prove properties about infinite sets.  
  We do this by proving that the property holds on the "base cases"--nonrecursive constructors.
  For example, (O : nat) and (nil : list A) are base cases.
  Then, we prove that the property is "preserved" by the recursive constructor.
  To prove the inductive case for a property P of nat's, we'll need to prove
  forall n, P n -> P (S n).
  For lists, it will be
  forall l x, P l => P (cons x l).
  *)

Definition compose (A B C: Set) (f: B -> C) (g: A -> B) : A -> C :=  
  fun x => f (g x).

Lemma map_map_compose:  
 forall (A B C: Set) (g: A -> B) (f: B -> C) (l: list A),  
  map f (map g l) = map (compose f g) l.
Proof.  intros.  induction 1.  + simpl. reflexivity.  + simpl. rewrite IHl. (** need to "unfold" compose to simpl *)
  unfold compose. reflexivity. Qed.

Fixpoint foldr (A B: Set) (f: A -> B -> B) (l: list A) (b: B) : B :=  
  match l with  |
  | nil => b  |
  | cons x xs => f x (foldr f xs b)  end.  
Eval cbv in (foldr plus (countdown 10) 0).

Fixpoint fact (n: nat) : nat :=  
  match n with  |
  | O => 1  |
  | S m => mult n (fact m)  end.

(** foldr f (cons 1 (cons 2 (cons 3 nil))) x  
  --> f 1 (f 2 (f 3 x))
  Notice how foldr replaces "cons" with "f" and "nil" with "x".
  *)

(** "foldr plus" is a summation.
Let's sum the values from 0 to 10 */
Eval cbv in (foldr plus (countdown 10) 0).

Fixpoint count (n: nat) : nat :=  
  match n with  |
  | O => 1  |
  | S m => mult n (fact m)  end.
Eval cbv in (fact 0).
Eval cbv in (fact 1).
Eval cbv in (fact 2).
Eval cbv in (fact 3).
Eval cbv in (fact 4).

Definition fact' (n: nat) : nat :=
match n with
| O => 1  | S m => foldr mult (map (plus 1) (countdown m)) 1  end.
Eval cbv in (fact' 0).
Eval cbv in (fact' 1).
Eval cbv in (fact' 2).
Eval cbv in (fact' 3).
Eval cbv in (fact' 4).

Lemma fact_fact' :
forall n,  fact n = fact' n. Proof. (*
(* challenge problem *)
Admitted. (*
(* we can also define map using fold *)

Definition map' (A B: Set) (f: A -> B) (l: list A) : list B :=
foldr (fun x acc => cons (f x) acc) l nil.

Lemma map_map' :
forall (A B: Set) (f: A -> B) (l: list A),
map f l = map' f l. Proof. intros. induction l. + simpl. reflexivity. + simpl. rewrite IHl. reflexivity.

(** simple but inefficient way to reverse a list *)

Fixpoint fast_rev_aux (A: Set) (l: list A) (acc: list A) : list A :=
match l
with
| nil => acc
| cons x xs => fast_revAux xs (cons x acc)  end.

Definition fast_rev (A: Set) (l: list A) :=
fast_rev_aux l nil. (*
(* let's make sure we got that right *)

Theorem rev_ok :
forall A (l: list A),

(* app is associative, meaning we can freely re-associate (move parens around) *)
Theorem app_assoc:
forall A (l1 l2 l3: list A),
app (app l1 l2) l3 = app l1 (app l2 l3).
Proof. intros. induction l1. + simpl. reflexivity. + simpl. rewrite IHl. reflexivity. Qed.

(* tail recursion is faster, but more complicated. *acc* is short for "accumulator". we "accumulate" with each recursive call. note that fast_rev_aux calls itself in tail position, i.e., as its result. tail recursion is faster because compilers for functional programming languages often do tail-call optimization ("TCO"), in which stack frames are re-used by recursive calls. *)

Fixpoint rev (A: Set) (l: list A) : list A :=
match l
with
| nil => nil
| cons x xs => app (rev xs) (cons x nil)  end.

Theorem app_nil:
forall A (l: list A),
app l nil = l. Proof. intros. induction l. + simpl. reflexivity. + simpl. rewrite IHl. reflexivity.

(** another flavor of fold. what’s the difference? when would you use one or the other? *)

Fixpoint foldl (A B: Set) (f: A -> B -> B) (l: list A) : list B :=
match l
with
| nil => b
| cons x xs => foldl f xs (f x b)  end.

(** add one list to the end of another *)

Fixpoint app (A: Set) (l1: list A) (l2: list A) : list A :=
match l1 with
| nil => l2
| cons x xs => cons x (app xs l2)  end.
Eval cbv in (app (cons 1 (cons 2 nil)) (cons 3 nil)).

Theorem app_nil:
forall A (l: list A),
app l nil = l. Proof. intros. induction l. + simpl. reflexivity. + simpl. rewrite IHl. reflexivity. Qed.

Theorem app_assoc:
forall A (l1 l2 l3: list A),
app (app l1 l2) l3 = app l1 (app l2 l3).
Proof. intros. induction l. + simpl. reflexivity. + simpl. rewrite IHl. reflexivity. Qed.

(* TIP: if simp doesn’t work, try unfolding! *)

(** reduces rev, but does nothing to rev_fast *)
unfold fast_rev. (** unfold fast_rev to fast_rev_aux *)
simpl. (** now we can simplify the term *)
reflexivity. (** TIP: if simp doesn’t work, try unfolding! *)
+ unfold fast_rev in *. (** this looks like it could be trouble... *)
simpl. rewrite <− IHl. (** STUCK! need to know about the rev_aux accumulator (acc) *)
(** TIP: if your IH seems weak, try proving something more general *)
Abort.

Lemma fast_revAuxOk :
forall A (l: list A),
fast_revAux l l l = app (rev l) l. Proof. intros. induction l. + simpl. reflexivity.
Abort.

Lemma fast_rev_aux_ok:
  forall A (l1 l2: list A),
  fast_rev_aux l1 l2 = app (rev l1) l2.
Proof.
  intros A l1.
  induction l1.
  + intros. simpl. reflexivity.
  + (* compare the induction hypothesis (IHl1) here with the one we had before. what’s different? why is this called “generalizing” the induction hypothesis? *)
    intros. simpl.
    rename l2 into foo.
    (* Note that we can rewrite by IHl1 even though it is universally quantified (i.e. there’s a forall). Coq will figure out what to replace “l2” with in IHl1 (cons a foo). *)
    rewrite IHl1.
    rewrite app_assoc.
    simpl.
    reflexivity.
Qed.

Lemma rev_ok:
  forall A (l: list A),
  fast_rev l = rev l.
Proof.
  intros. unfold fast_rev.
  rewrite fast_rev_aux_ok.
  rewrite app_nil.
  reflexivity.
Qed.

(* <<

(** STUCK AGAIN! need to know for *any* l2 *)
(** TIP: if your IH seems weak, only intro up to the induction variable *)

decompose the IH.

end.

Theorem snoc_app_singleton:
  forall A (l: list A) (x: A),
  snoc l x = app l (cons x nil).
Proof.
  intros.
  induction l.
  + simpl. reflexivity.
  + simpl. rewrite IHl. reflexivity.
Qed.

Theorem app_snoc_l:
  forall A (l1: list A) (l2: list A) (x: A),
  app (snoc l1 x) l2 = app l1 (cons x l2).
Proof.
  intros.
  induction l1.
  + simpl. reflexivity.
  + simpl. rewrite IHl1. reflexivity.
Qed.

Theorem app_snoc_r:
  forall A (l1: list A) (l2: list A) (x: A),
  app l1 (snoc l2 x) = snoc (app l1 l2) x.
Proof.
  intros.
  induction l1.
  + simpl. reflexivity.
  + simpl. rewrite IHl1. reflexivity.
Qed.

(*

(* simple but inefficient way to reverse a list *)

Fixpoint rev_snoc (A: Set) (l: list A) : list A :=
  match l with
  | nil => nil
  | cons x xs => snoc (rev_snoc xs) x
end.

Lemma fast_rev_aux_ok_snoc:
  forall A (l1 l2: list A),
  fast_rev_aux l1 l2 = app (rev_snoc l1) l2.
Proof.
  intros A l1.
  induction l1.
  + intros. simpl. reflexivity.
  + intros. simpl.
    rewrite IHl1.
    rewrite app_assoc.
    simpl.
    reflexivity.
Qed.

Lemma fast_rev_ok_snoc:
  forall A (l: list A),
  fast_rev l = rev_snoc l.
Proof.
  intros. unfold fast_rev.
  rewrite fast_rev_aux_ok_snoc.
  rewrite app_nil.
  reflexivity.
Qed.

Lemma length_app:
  forall A (l1 l2: list A),
  length (app l1 l2) = plus (length l1) (length l2).
Proof.
  intros.
  induction l1.
Lemma plus_1_S:
  forall n,  plus n 1 = S n.
Proof.
  intros.
  induction n.
  + simpl. reflexivity. 
  + simpl. rewrite IHn. reflexivity.
Qed.

Lemma rev_length:
  forall A (l: list A),  length (rev l) = length l.
Proof.
  intros.
  induction l.
  + simpl. reflexivity.
  + simpl. rewrite length_app.
    simpl. rewrite plus_1_S.
    rewrite IHl. reflexivity.
Qed.

Lemma rev_app:
  forall A (l1 l2: list A),  rev (app l1 l2) = app (rev l2) (rev l1).
Proof.
  intros.
  induction l1.
  + simpl. rewrite app_nil. reflexivity.
  + simpl. rewrite IHl1. rewrite app_assoc.
    reflexivity.
Qed.

Lemma rev_involutive:
  forall A (l: list A),  rev (rev l) = l.
Proof.
  intros.
  induction l.
  + simpl. reflexivity.
  + simpl. rewrite rev_app.
    simpl. rewrite IHl. reflexivity.
Qed.