```
(** * 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).
Eval cbv in (length (cons 1 nil)).
Eval cbv in (length (cons 2 (cons 1 nil))).
Eval cbv in (length nil).

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: 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 (map is_zero (countdown 3)).

Fixpoint is_even (n: nat) : bool :=
  match n with
  | O => true
  | S O => false
  | S (S m) => is_even m
  end.

Eval cbv in (map is_even (countdown 3)).
```



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

(** discuss how does induction works *)

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 l.
  + simpl. reflexivity.
  + (*
  map f (map g (cons a l))
  --> map f (cons (g a) (map g l))
  --> cons (f (g a)) (map f (map g l))
  *)
    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.

(**
  foldr f (cons 1 (cons 2 (cons 3 nil))) x
  -->
  f 1 (f 2 (f 3 x))
*)

Check plus.
Print plus.
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.

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.
    (** again, need to unfold so simpl can work *)
    unfold map'. simpl.
    reflexivity.
    (** note: very sensitive to order of rewrite and unroll! *)
Qed.

(** another flavor of fold *)
Fixpoint foldl (A B: Set) (f: A -> B -> B)
               (l: list A) (b: B) : 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 l1.
  + simpl. reflexivity.
  + simpl. rewrite IHl1. reflexivity.
```



```
Qed.

(** simple but inefficient way to reverse a list *)
Fixpoint rev (A: Set) (l: list A) : list A :=
  match l with
  | nil => nil
  | cons x xs => app (rev xs) (cons x nil)
  end.

(** tail recursion is faster, but more complicated *)
Fixpoint fast_rev_aux (A: Set) (l: list A) (acc: list A) : list A :=
  match l with
  | nil => acc
  | cons x xs => fast_rev_aux xs (cons x acc)
  end.

Definition fast_rev (A: Set) (l: list A) : list A :=
  fast_rev_aux l nil.

(** let's make sure we got that right *)
Theorem rev_ok:
  forall A (l: list A),
  fast_rev l = rev l.
Proof.
  intros.
  induction l.
  + simpl. (** 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 simpl 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_rev_aux_ok:
  forall A (l1 l2: list A),
  fast_rev_aux l1 l2 = app (rev l1) l2.
Proof.
  intros.
  induction l1.
  + simpl. reflexivity.
  + simpl.
    (** STUCK AGAIN! need to know for *any* l2 *)
    (** TIP: if your IH seems weak, only intro up to the induction variable *)
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.
  + intros. simpl.
    rename l2 into foo.
    rewrite IHl1.
    rewrite app_assoc.
    simpl. reflexivity.
Qed.

(** now we can prove rev_ok as a special case of rev_aux_ok *)
Lemma fast_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.

(**

<<

                               ~-.
         ,,,;             ~-.~-.~-
        (.../            ~-.~-.~-.~-.~-.
        } o~',          ~-.~-.~-.~-.~-.~-.
        (/    \       ~-.~-.~-.~-.~-.~-.~-.
         ;    \     ~-.~-.~-.~-.~-.~-.~-.
        ;     {_.~-.~-.~-.~-.~-.~-.~
       ;:  .-~'     ~-.~-.~-.~-.~-.
      ;.: :'    ._    ~-.~-.~-.~-.~-
       ;::'-.    '-._  ~-.~-.~-.~-
        ;::. '-.    '-,~-.~-.~-.
         ';::::.'''-.-'
          ';::;;;,:'
             '||"
             / |
           ~' ~"'

>>

*)

(** add an element to the end of a list *)
Fixpoint snoc (A: Set) (l: list A) (x: A) : list A :=
  match l with
  | nil => cons x nil
  | cons y ys => cons y (snoc ys x)
  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_snoc_l.
    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.
  + simpl. reflexivity.
  + simpl. rewrite IHl1. reflexivity.
Qed.

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.

(** * END LECTURE 02 *)

(** SYNTAX *)

(** We can define a programming language as an inductive datatype. *)

(** include string library for variable names *)
Require Import String.

(**
  E ::= N | V | E + E | E * E | E ? E
*)
Inductive expr : Set :=
| Const : nat -> expr
| Var   : string -> expr
| Add   : expr -> expr -> expr
| Mul   : expr -> expr -> expr
| Cmp   : expr -> expr -> expr.

(**
  S ::= Skip | V <- E | S ;; S |
        IF E THEN S ELSE S |
        WHILE E {{ S }}
*)
Inductive stmt : Set :=
| Skip  : stmt
| Asgn  : string -> expr -> stmt
| Seq   : stmt -> stmt -> stmt
| Cond  : expr -> stmt -> stmt -> stmt
| While : expr -> stmt -> stmt.

(** Programs are just elements of type stmt. *)

Definition prog_skip : stmt :=
  Skip.

Definition prog_set_x : stmt :=
  Asgn "x" (Const 1).

Definition prog_incr_x_forever : stmt :=
  While (Const 1) (Asgn "x" (Const 1)).

Definition prog_xth_fib_in_y : stmt :=
  Seq (Asgn "y"  (Const 0)) (
  Seq (Asgn "y0" (Const 1)) (
  Seq (Asgn "y1" (Const 0)) (
  Seq (Asgn "i"  (Const 0)) (
      While (Cmp (Var "i") (Var "x")) (
            Seq (Asgn "y"  (Add (Var "y0") (Var "y1"))) (
            Seq (Asgn "y0" (Var "y1")) (
            Seq (Asgn "y1" (Var "y")) (
                (Asgn "i"  (Add (Var "i") (Const 1)))
```



```
            ))))
  )))).

(** but nobody wants to write programs like this,
    so Coq provides a "Notation" mechanism *)

Notation "'C' X"   := (Const X) (at level 80).
Notation "'V' X"   := (Var X)   (at level 81).
Notation "X <+> Y" := (Add X Y) (at level 83, left associativity).
Notation "X <*> Y" := (Mul X Y) (at level 82, left associativity).
Notation "X <?> Y" := (Cmp X Y) (at level 84).

Notation "X <- Y" := (Asgn X Y) (at level 86).
Notation "X ;; Y" := (Seq X Y)  (at level 87, left associativity).
Notation "'IF' X 'THEN' Y 'ELSE' Z" := (Cond X Y Z) (at level 88).
Notation "'WHILE' X {{ Y }}" := (While X Y) (at level 89).

Definition prog_fib : stmt :=
  "y"  <- C 0;;
  "y0" <- C 1;;
  "y1" <- C 0;;
  "i"  <- C 0;;
  WHILE (V"i" <?> V"x") {{
    "y"  <- V"y0" <+> V"y1";;
    "y0" <- V"y1";;
    "y1" <- V"y";;
    "i"  <- V"i" <+> C 1
  }}.

(** Notation provides us with "concrete" syntax which
    "desugars" to the underlying "abstract syntax tree".
*)

Fixpoint nconsts (e: expr) : nat :=
  match e with
  | Const _ => 1
  | Var _ => 0
  | Add e1 e2 => plus (nconsts e1) (nconsts e2)
  | Mul e1 e2 => plus (nconsts e1) (nconsts e2)
  | Cmp e1 e2 => plus (nconsts e1) (nconsts e2)
  end.

Lemma has_3_consts:
  exists e, nconsts e = 3.
Proof.
  exists (Add (Const 1) (Add (Const 2) (Const 3))).
  simpl. reflexivity.
Qed.

Fixpoint expr_with_n_consts (n: nat) : expr :=
  match n with
  | O => Var "x"
  | S m => Add (Const 0) (expr_with_n_consts m)
  end.

Lemma has_n_consts:
  forall n,
  exists e, nconsts e = n.
Proof.
  intros.
  exists (expr_with_n_consts n).
  induction n.
  + simpl. reflexivity.
  + simpl. rewrite IHn. reflexivity.
Qed.

Definition orb (b1 b2: bool) : bool :=
  match b1 with
  | true => true
```

```
  | false => b2
  end.

Fixpoint has_const (e: expr) : bool :=
  match e with
  | Const _ => true
  | Var _ => false
  | Add e1 e2 => orb (has_const e1) (has_const e2)
  | Mul e1 e2 => orb (has_const e1) (has_const e2)
  | Cmp e1 e2 => orb (has_const e1) (has_const e2)
  end.

Fixpoint has_var (e: expr) : bool :=
  match e with
  | Const _ => false
  | Var _ => true
  | Add e1 e2 => orb (has_var e1) (has_var e2)
  | Mul e1 e2 => orb (has_var e1) (has_var e2)
  | Cmp e1 e2 => orb (has_var e1) (has_var e2)
  end.

Lemma expr_bottoms_out:
  forall e,
  orb (has_const e) (has_var e) = true.
Proof.
  intros.
  induction e.
  + (** const *)
    simpl. reflexivity.
  + (** var *)
    simpl. reflexivity.
  + (** add *)
    simpl.
    destruct (has_const e1).
    - (** e1 has a const *)
      simpl. reflexivity.
    - (** e1 does not have a const *)
      destruct (has_const e2).
      * (** e2 has a const *)
        simpl. reflexivity.
      * (** e2 does not have a const *)
        simpl.
        (** we also want to simplify in the hypotheses *)
        simpl in *.
        (** and rewrite with the results *)
        rewrite IHe1. rewrite IHe2.
        simpl. reflexivity.
  + (** mul *)
    simpl.
    destruct (has_const e1).
    - (** e1 has a const *)
      simpl. reflexivity.
    - (** e1 does not have a const *)
      destruct (has_const e2).
      * (** e2 has a const *)
        simpl. reflexivity.
      * (** e2 does not have a const *)
        simpl.
        (** we also want to simplify in the hypotheses *)
        simpl in *.
        (** and rewrite with the results *)
        rewrite IHe1. rewrite IHe2.
        simpl. reflexivity.
  + (** cmp *)
    simpl.
    destruct (has_const e1).
    - (** e1 has a const *)
      simpl. reflexivity.
    - (** e1 does not have a const *)
```



```
      destruct (has_const e2).
      * (** e2 has a const *)
        simpl. reflexivity.
      * (** e2 does not have a const *)
        simpl.
        (** we also want to simplify in the hypotheses *)
        simpl in *.
        (** and rewrite with the results *)
        rewrite IHe1. rewrite IHe2.
        simpl. reflexivity.
Qed.

(** THE ABOVE PROOF IS VERY BAD.  Make it better! *)
(** Hint: think about how to rearrange the orbs. *)

(** Some interesting types *)

Inductive True : Prop :=
| I : True.

Inductive False : Prop :=
.

Lemma bogus:
  False -> 1 = 2.
Proof.
  intros.
  inversion H.
Qed.

Lemma also_bogus:
  1 = 2 -> False.
Proof.
  intros.
  discriminate.
Qed.

Inductive yo : Prop :=
| yolo : yo -> yo.

Lemma yoyo:
  yo -> False.
Proof.
  intros.
  inversion H.
  (** well, that didn't work *)
  induction H.
  assumption. (** but that did! *)
Qed.
```