(** * Lecture 03 *)

(** include some useful libraries *)

Require Import Bool.
Require Import List.
Require Import String.
Require Import Omega.

(** List provides the cons notation "::"
"x :: xs" is the same as "cons x xs"
*)

Fixpoint my_length {A: Set} (l: list A) : nat :=
  match l with
  | nil => O
  | x :: xs => S (my_length xs)
  end.

(** List provides the append notation "++"
"xs ++ ys" is the same as "app xs ys"
*)

Fixpoint my_rev {A: Set} (l: list A) : list A :=
  match l with
  | nil => nil
  | x :: xs => rev xs ++ x :: nil
  end.

(** some interesting types *)

(** Prop is like Set, but for propositions *)

Inductive myTrue : Prop :=
  I : myTrue.

Lemma foo:
  myTrue.
  Proof. constructor. (** exact I. *)
  Qed.

Lemma foo':
  Set.
  Proof.
  exact (list nat).
  (** exact bool. *)
  Qed.

Inductive myFalse : Prop :=
  .

Print False.

Lemma bogus:
  False -> 1 = 2.
  Proof.
  intros.
  (** inversion does case analysis on a hypothesis. For each way
  that hypothesis could have been proved, you need to complete the
  subgoal *)
  inversion H.
  Qed.

Lemma also_bogus:
  1 = 2 -> False.

Print eq.

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

Lemma yoyo:
  yo -> False.
  Proof.
  intros.
  inversion H. inversion H0. inversion H1.
  (** well, that didn't work *)
  induction H.
  assumption. (** but that did! *)
  Qed.

(** check out negation *)

Print not.

(** ** Expression Syntax *)

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

Inductive expr : Set :=
  Const : nat -> expr
  | Var   : string -> expr
  | Add   : expr -> expr -> expr
  | Mul   : expr -> expr -> expr
  | Cmp   : expr -> expr -> expr.

Check (Const 0).
Check (Var "x").
Check (Add (Const 0) (Var "x")).
Check (Mul (Add (Const 0) (Var "x")) (Add (Const 0) (Var "y")))
  (Mul (Var "y") (Const 0))).

(** On paper, this would be written as a
"BNF grammar" as:

<<
expr ::= N          |  V          |  expr <+> expr          |  expr <*> expr          |  expr <?> expr
>>

(** Coq provides mechanism to define
your own notation which we can use

to get "concrete syntax" *)

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).

Check (C 0).
Check (V"x").
Check (C 0 <+> V"x").
Check (C 0 <+> V"x" <+> C 0 <+> V"x").
Check \((C \ 0 \ <+> V{}^*x) \ <==> \ (C \ 0 \ <+> V{}^*x))\).
Check \((C \ 0 \ <+> V{}^*x) \ <==> \ V{}^*y \ <+> C \ 0\).

(** try View == Display all basic low-level contents *)

(** parsing is classic CS topic, but won’t say much more *)

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

(** we can write functions to analyze expressions *)

Lemma expr_w_3_consts:
exists e,  nconsts e = 3.
Proof.  exists (C 3 <+> C 2 <+> C 1).
simpl. reflexivity.
Qed.

Fixpoint esize (e: expr) : nat :=
match e with
| Const _   => 1 (* same as S O *)
| Var _     => 1 | Add e1 e2 => esize e1 + esize e2
(* same as plus (esize e1) (esize e2) *)
| Mul e1 e2 => esize e1 + esize e2
| Cmp e1 e2 => esize e1 + esize e2
end.

(** and do proofs about programs *)

Lemma nconsts_le_size: forall e, nconsts e <= esize e.
Proof. intros.
  induction e.
  + simpl.
    auto.
    (** auto will solve many simple goals *)
    + simpl. auto.
    + simpl. omega.
    (** omega will solve many arithmetic goals *)
    + simpl. omega.
    + simpl. omega.
Qed.

(** that proof had a lot of copy-past : ( *)

Lemma nconsts_le_size':
forall e, nconsts e <= esize e.
Proof. intros.
  (**
    induction e;"
    on every resulting subgoal do simpl, then
    on every resulting subgoal do auto, then
    on every resulting subgoal do omega
    *)
  induction e; simpl; auto; omega.
  (** note that after the auto,
    only the Add, Mul, and Cmp subgoals remain,
    but it's hard to tell since
    the proof does not "pause" *)
Qed.

Locate "\<=".

(* take a second to consider \<= *)
Print le.

(** it's a relation defined as an inductive predicate *)

(** we give rules for when the relation holds *)

(** we can define our own relations
to encode properties of expressions *)

Inductive has_const : expr -> Prop :=
  | hc_const : forall n, has_const (Const n)
  | hc_add_l : forall e1 e2, has_const e1 -> has_const (Add e1 e2)
  | hc_add_r : forall e1 e2, has_const e2 -> has_const (Add e1 e2)
  | hc_mul_l : forall e1 e2, has_const e1 -> has_const (Mul e1 e2)
  | hc_mul_r : forall e1 e2, has_const e2 -> has_const (Mul e1 e2)
  | hc_cmp_l : forall e1 e2, has_const e1 -> has_const (Cmp e1 e2)
  | hc_cmp_r :forall e1 e2, has_const e2 -> has_const (Cmp e1 e2).

Lemma add_mul_comm:
forall e1 e2, Add e1 e2 = Add e2 e1 -> False.
Proof.
  intros.
  specialize (H (Const 0) (Const 1)).
  inversion H.
Qed.

Inductive has_var : expr -> Prop :=
  | hv_var : forall s, has_var (Var s)
  | hv_add_l :forall e1 e2, has_var e1 -> has_var (Add e1 e2)
  | hv_add_r :forall e1 e2, has_var e2 -> has_var (Add e1 e2)
  | hv_mul_l :forall e1 e2, has_var e1 -> has_var (Mul e1 e2)
**Fixpoint** hasConst (e: expr) : bool :=
match e with
| Const _ => true
| Var _ => false
| Add e1 e2 => orb (hasConst e1) (hasConst e2)
| Mul e1 e2 => orb (hasConst e1) (hasConst e2)
| Cmp e1 e2 => orb (hasConst e1) (hasConst e2)
end.

(* we could also write boolean functions
   to check the same properties *)

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

(* the Bool library provides "||" as a notation for orb *)

**Lemma** has_const_hasConst:
forall e, has_const e -> hasConst e = true.
Proof.
  intros. induction e.
  + (* we can prove this case with a constructor *)
  apply hc_const.
  (* constructor. *)
  apply hc_const. (* this uses hc_const *)
  + (* Uh oh, no constructor for has_const can possibly produce a value of our goal type! It's OK though because we have a bogus hypothesis. *)
  discriminate.
  + (* now do Add case *)
  simp in H.  (* either e1 or e2 had a Const *)
  apply orb_true_iff in H.  (* consider cases for H *)
  destruct H.
  - (* e1 had a Const *)
    apply hc_add_l.  (* constructor. *)
    apply hc_add_l.  (* constructor. *)
    apply hc_add_l.  (* constructor. *)
    apply hc_add_l.  (* constructor. *)
   + (* Mul case is similar *)
   simp in H; apply orb_true_iff in H; destruct H.
   - (* constructor will just use hc_mul_l *)
   constructor. apply lcmul. assumption.
   - (* constructor will screw up and try hc_mul_l again! *)
   constructor. (** OOPS! *)
   Undo.

End hasConst.
apply hc_mul_r. apply IH2. assumption.
+ (** Cmp case is similar *)
         simpl in H; apply orb_true_iff in H; destruct H.
         - constructor; auto.
         - apply hc_cmp_r; auto.
Qed.

(** all that was only for the true cases! *)
(** can also use not and do the false cases *)

Lemma not_has_const_hasConst:
 forall e,  ~ has_const e -> hasConst e = false.
Proof.
  unfold not. intros.
  induction e.
  + simpl.
    (** uh oh, trying to prove something bogus *)
    (** better exploit a bogus hypothesis *)
    exfalso. (** proof by contradiction *)
    apply H. constructor.
  + simpl. reflexivity.
  + simpl. apply orb_false_iff.
  (** prove conjunction by proving left and right *)
  split.
    - apply IH1. intro.
      apply H. apply hc_add_l. assumption.
    - apply IH2. intro.
      apply H. apply hc_add_r. assumption.
  + (** Mul case is similar *)
  simpl; apply orb_false_iff.
  split.
    - apply IH1. intro.
      apply H. apply hc_mul_l. assumption.
    - apply IH2; intro.
      apply H. apply hc_mul_r. assumption.
  + (** Cmp case is similar *)
  simp; apply orb_false_iff.
  split.
    - apply IH1; intro.
      apply H. apply hc_cmp_l. assumption.
    - apply IH2; intro.
      apply H. apply hc_cmp_r. assumption.
Qed.

Lemma false_hasConst_hasConst:
 forall e,  hasConst e = false -> ~ has_const e.
Proof.
  unfold not. intros.
  induction e; (* crunch down everything in subgoals *)
  simpl in *.
  + discriminate.
  + inversion H0.
  + apply orb_false_iff in H.
  (** get both proofs out of a conjunction *)
  by destructing it *)
  destruct H.
  (** case analysis on H0 *)
  (** DISCUSS: how do we know to do this? *)
  inversion H0.
  - subst. auto. (** auto will chain things for us *)
  - subst. auto.
  + (** Mul case similar *)
  appl orb_false_iff in H; destruct H.
  inversion H0; subst; auto.
  Qed.

(/** we can stitch all these together */)

Lemma has_const_iff_hasConst:
 forall e,  has_const e <-> hasConst e = true.
Proof.
  intros. split.
    + (** -> *)
      apply has_const_hasConst.
    + (** <- *)
      apply hasConst_has_const.
Qed.

(/** We can also do all the same sorts of proofs for has_var and hasVar *)

Lemma has_var_hasVar:
 forall e,  has_var e -> hasVar e = true.
Proof.
  (** TODO: try this without copying from above *)
  Admitted.

Lemma hasVar_has_var:
 forall e,  hasVar e = true -> has_var e.
Proof.
  (** TODO: try this without copying from above *)
  Admitted.

Lemma has_var_iff_hasVar:
 forall e,  has_var e <-> hasVar e = true.
Proof.
  (** TODO: try this without copying from above *)
  Admitted.

(/** we can also prove things about expressions *)

Lemma expr_bottoms_out:
 forall e,  has_const e / has_var e.
Proof.
  intros. induction e.
  + (** prove left side of disjunction *)
    left.
    constructor.
  + (** prove right side of disjunction *)
    right.
    constructor.
  + (** case analysis on IH1 *)
    destruct IH1.
    - left. constructor. assumption.
    - right. constructor. assumption.
  + (** Mul case similar *)
    destruct IH1.
    - left. constructor. assumption.
    - right. constructor. assumption.
  + (** Cmp case similar *)
    destruct IH1.
    - left. constructor. assumption.
    - right. constructor. assumption.
Qed.
(* we could have gotten some of the
  has_const lemmas by being a little clever!
  (but then we wouldn’t have
  learned as many tactics ;) *)

Lemma has_const_hasConst’:
  forall e, has_const e -> hasConst e = true.
Proof.  intros.
  induction H; simpl; reflexivity. 
  Qed.  Admitted.

Lemma has_const_hasConst’’:
  forall e, has_const e -> hasConst e = true.
Proof.  intros.
  induction H; simpl; auto; rewrite orb_true_iff; auto.
  Qed.

Lemma not_has_const_hasConst’:
  forall e, ~ has_const e -> hasConst e = false.
Proof.  unfold not; intros.
  destruct (hasConst e) eqn:?:.
  - exfalso. apply H.
  - rewrite has_const_hasConst in Heqb.
  Qed.

Lemma false_hasConst_hasConst’:
  forall e, hasConst e = false -> ~ has_const e.
Proof.  unfold not; intros.
  destruct (hasConst e) eqn:?:.
  - discriminate.
  - rewrite has_const_hasConst.
  (* NOTE: we got another subgoal! *)

Qed.

(* In general:
  - relational defs are nice when you want to use inversion
  - functional defs are nice when you want to use simpl *)