(*) 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 the type of proofs *)
(* Just like Set, we use it as a type for things *)
(* Unlike Set, we mainly use it for the type of facts *)

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

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

Lemma also_bogus:
  1 = 2 -> False.
Proof.
  intros.
  (** discriminate is a tactic that looks for mismatching constructors *)
  (** under the hood, 1 looks like S (0) and 2 looks like S (S 0) *)
  (** It peels off one S, gets 0 = S 0 *)
  (** 0 and S are different constructors, thus they are not equal *)
discriminate.
Qed.

(* Note that even equality is defined, not built in *)
Print eq.

(* What’s wrong with this? *)
(* There’s no way to build any objects of type yo *)
Inductive yo : Prop :=
  yolo : yo -> yo.
(* We want to prove that no objects of type yo exist *)
(* We can prove that any object of that type would mean False *)
(* Thus there are none *)
Lemma yoyo:
  yo -> False.
Proof.
  intros.
  (** well, that didn’t work *)
  induction H.
  assumption.
  (** but that did! *)
Qed.

(* check out negation *)
(* It looks just like what we just did *)
Print not.

(* * Expression Syntax *)
(* Now let’s build a programming language! *)
(* We can define parts of a language *)
(* as an inductive datatype. *)

Inductive expr : Set :=
  (* A constant expression, like "3" or "0" *)
  Const : nat -> expr
  (* A program variable, like "x" or "foo" *)
  Var : string -> expr
  (* Adding expressions *)
  Add : expr -> expr -> expr
  (* Multiplying expressions *)
  Mul : expr -> expr -> expr
  (* Comparing expressions *)
  Cmp : expr -> expr -> expr.
  (** On paper, this would be written as a *)
  (** BNF grammar as:<<

Oct 07, 15 11:21 L03_annotated.v Page 1/11

Oct 07, 15 11:21 L03_annotated.v Page 2/11
expr ::= N          |  V          |  expr <+> expr          |  expr <*> expr          |  expr <?> expr
*> (*
while parsing is still an active research topic in some places, *)
(* feel free to ignore most of this, especially the stuff farther right *)
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, no associativity).
(* parsing is classic CS topic, but won’t say much more *)
(* we can write functions to analyze expressions *)
(* Here we’re simply going to count the number of const subexpressions in a given expression *)
Fixpoint nconsts (e: expr) : nat :=
match e
| Const _   => 1 (* same as S O *)
| Var _     => 0 (* same as O *)
| Add e1 e2 => nconsts e1 + nconsts e2
| Mul e1 e2 => nconsts e1 + nconsts e2
| Cmp e1 e2 => nconsts e1 + nconsts e2 end.
(* Coq also provides existential quantifiers *)
(* We prove them by providing concrete examples *)
Lemma expr_w_3_consts: exists e,  nconsts e = 3. Proof.  exists (C 3 <+> C 2 <+> C 1).  Qed.
(* we know how to do it pretty well in a lot of cases *)
(* we can define our own relations to encode properties of expressions *)
Inductive has_const : expr −> Prop :=
| hc_const :    forall c, has_const (Const c)| 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_cmp :    forall e1 e2,    has_const e1 −>    has_const (Cmp e1 e2)
(* Each of the constructors corresponds to how you can prove this fact *)

(** 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, no associativity).
(*)
(* feel free to ignore most of this, especially the stuff farther right *)
(* we know how to do it pretty well in a lot of cases *)
(* we can define our own relations to encode properties of expressions *)
(* we can define our own relations to encode properties of expressions *)
Inductive has_const : expr −> Prop :=
| hc_const :    forall c, has_const (Const c)| 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)

(** 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, no associativity).
(** parsing is classic CS topic, but won’t say much more *)
(** while parsing is still an active research topic in some places, *)
(** we know how to do it pretty well in a lot of cases *)
(** we can write functions to analyze expressions *)
(** Here we’re simply going to count the number of const subexpressions in a given expression *)
Fixpoint nconsts (e: expr) : nat :=
match e
| Const _   => 1 (* same as S O *)
| Var _     => 0 (* same as O *)
| Add e1 e2 => nconsts e1 + nconsts e2
| Mul e1 e2 => nconsts e1 + nconsts e2
| Cmp e1 e2 => nconsts e1 + nconsts e2 end.

(** 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, no associativity).
(** parsing is classic CS topic, but won’t say much more *)
(** while parsing is still an active research topic in some places, *)
(** we know how to do it pretty well in a lot of cases *)
(** we can define our own relations to encode properties of expressions *)
Inductive has_const : expr −> Prop :=
| hc_const :    forall c, has_const (Const c)| 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)
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).

(** Are add and mul commutative? *)
(** Not as just syntax *)

Lemma add_comm :
  forall e1 e2, Add e1 e2 = Add e2 e1 → False.
Proof.
  intros.
  (** specialize gives concrete arguments to hypotheses with forall *)
  specialize (H (Const 0) (Const 1)).
  (** inversion is smart *)
  inversion H.
Qed.

(** Similarly, we can define a relation for having a variable *)

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)
| hv_mul_r :
  forall e1 e2, has_var e2 → has_var (Mul e1 e2)
| hv_cmp_l :
  forall e1 e2, has_var e1 → has_var (Cmp e1 e2)
| hv_cmp_r :
  forall e1 e2, has_var e2 → has_var (Cmp e1 e2).

(** we could write boolean functions to check the same properties *)
(** orb is just or for booleans *)

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.

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

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.

(** That looks way easier! However, as the quarter progresses, we’ll see that sometime defining a property as an inductive relation is more convenient *)

(** We can prove that our relational and functional versions agree *)
(** This property is that the hasConst function is COMPLETE with respect to the relation *)
(** Thus, anything that satisfies the relation evaluates to "true" with the function *)

Lemma has_const_hasConst:
  forall e, has_const e → hasConst e = true.
Proof.
  intros.
  induction e.
  + simpl. reflexivity. + simpl. (** uh oh, trying to prove something false! *)
  (** it’s OK though because we have a bogus hyp! *)
  inversion H.
  (** inversion lets us do case analysis on how a hypothesis of an inductive type may have been built. In this case, there is no way to build a value of type “has_const (Var s)”, so we complete the proof of this subgoal for all zero ways of building such a value *)
  + (** here we use inversion to consider how a value of type “has_const (Add e1 e2)” could have been built *)
  inversion H.
  - (** built with hc_add_l *)
    subst. (** subst rewrites all equalities it can *)
    apply IHe1 in H1.
    simpl. (** remember notation "||" is same as orb *)
    rewrite H1. simpl. reflexivity.
  - (** built with hc_add_r *)
    subst. apply IHe2 in H1.
    simpl. rewrite H1. (** use fact that orb is commutative *)
    rewrite orb_comm.
    (** you can find this by turning on auto completion or using a search query *)
    SearchAbout orb.
    simpl. reflexivity.
  + (** Mul case is similar *)
    inversion H; simpl; subst.
Lemma hasConst_has_const:
\forall e, \ hasConst e = true \rightarrow \ has_const e.
Proof.
  intros.
  induction e.
    + simpl.
      (** we can prove this case with a constructor *)
      constructor. (** this uses has_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. *)
      simpl in \ H.
      discriminate.
    + (** now do Add case *)
      simpl 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.
        apply IHe1.
        assumption.
      - (** e2 had a Const *)
        apply hc_add_r.
        apply IHe2.
        assumption.
    + (** Mul case is similar *)
      simpl in \ H.
      (** constructor will screw up and try hc_mul_l again! *)
      (* constructor is rather dim *)
      constructor. (** OOPS! *)
      Undo.
      apply hc_mul_r. apply IHe2. assumption.
    + (** Cmp case is similar *)
      simpl in \ H.
      apply orb_true_iff in \ H.
      (** constructor will work again! (*)
      (** constructor will use hc_mul_l *)
      constructor. apply IHe1. assumption.
      (** constructor will work again! (*)
      (** constructor is rather dim *)
      constructor. (** OOPS! *)
      Undo.
      apply hc_mul_r. apply IHe2. assumption.
Qed.

Lemma has_const_iff_hasConst:
\forall e, \ has_const e \iff \ hasConst e = true.
Proof.
  intros.
  split.
    + (** \rightarrow *)
      apply hasConst_has_const.
    + (** \leftarrow *)
      apply hasConst_has_const.
Qed.
reflexivity.
Qed.

(** Now the other direction of the false case *)

Lemma false_hasConst_hasConst:
  forall e, hasConst e = false -> ~ has_constr 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 *)
  apply orb_false_iff in H; destruct H.
  inversion H0; subst; auto.
  + (** Cmp case similar *)
  apply orb_false_iff in H; destruct H.
  inversion H0; subst; auto.
Qed.

(** Since we've proven the iff for the true case *)
(** We can use it to prove the false case *)

This is the same lemma as above, but using our previous results *)

Lemma false_hasConst_hasConst':
 forall e, hasConst e = false -> ~ has_constr e.
Proof.
  intros. rewriting has Const hasConst in H0.
  discriminate.
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.
  + (** prove right side of disjunction *)
    right.
  + (** case analysis on IHe1 *)
    destruct IHe1.
    - left. constructor. assumption.
    - right. constructor. assumption.
  + (** Mul case similar *)
    destruct IHe1.
    - left. constructor. assumption.
    - right. constructor. assumption.
  + (** Cmp case similar *)
    destruct IHe1.
    - left. constructor. assumption.
    - right. constructor. assumption.
Qed.

(** we could have gotten some of the has_constr 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; simp; auto.
  + rewrite orb_true_iff. auto.
  + rewrite orb_true_iff. auto.
  + rewrite orb_true_iff. auto.
  + rewrite orb_true_iff. auto.
  + rewrite orb_true_iff. auto.
Qed.

(** or even better *)

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

Lemma not_has_const_hasConst'':
 forall e, ~ has_const e -> hasConst e = false.
Proof.
Lemma false_hasConst_hasConst'':
  forall e,
  hasConst e = false ->
  ~ has_const e.
Proof.
  unfold not; intros.
  destruct (hasConst e) eqn:?.
  - discriminate.
  - rewrite hasConst_hasConst in Hqb.
   (** NOTE: we got another subgoal! **)
   * discriminate.
   * assumption.
Qed.