```
(** * Lecture 03 *)

(** Include some useful libraries. *)
Require Import Bool.
Require Import List.
Require Import String.
Require Import ZArith.
Require Import Omega.

(** List provides the cons notation "::":

    [x :: xs]   is the same as    [cons x xs]
*)
Fixpoint my_length {A: Type} (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: Type} (l: list A) : list A :=
  match l with
  | nil => nil
  | x :: xs => rev xs ++ x :: nil
  end.

(**
  [Prop] is the type we give to propositions.

  [Prop] basically works the same as [Type].
  In fact, its type is [Type]:
*)
Check Prop.
(**
<<
Prop
     : Type

>>

  The difference is that Coq ensures our programs
  never depend on inputs whose type is in [Prop].
  This helps when we extract programs: we can ensure
  that all [Prop]-related computations are only
  done at compile time and that our extracted programs never
  compute over data whose type is in Prop.
*)

(**
  [myTrue] is a proposition.  It has a single constructor, [I],
  which takes no arguments.  This means we can always produce
  a value of type [myTrue] by just using the [I] constructor.

  In Coq, we say that a proposition [P] is true if we can
  produce some term [t] that has type [P].  That is, [P] is
  true if there exists [t : P].
*)
Inductive myTrue : Prop :=
| I : myTrue.

Lemma foo :
  myTrue.
Proof.
  (** We use the constructor tactic to ask Coq
      to find a constructor that satisfies the goal. *)
```



```
  constructor.
  (** Alternatively, we could have explicitly told
      Coq exactly which constructor to use with:
<<
        exact I.
>>
  *)
Qed.

(**
  [myFalse] is also a proposition.  It has ZERO constructors.
  That means there is no way to produce a value of type [myFalse].
  Since a proposition [P] is true only when we can produce a value
  of type [P], we know that [myFalse] is in fact false.
*)
Inductive myFalse : Prop :=
  .


(**
  Given a proof of [myFalse], we can prove _anything_.
*)
Lemma bogus:
  False -> 1 = 2.
Proof.
  intros.
  (**
    Inversion performs case analysis on a hypothesis.
    Suppose our goal is [G] and we ask Coq to do inversion
    on hypothesis [H : T].  For each constructor [C] of [T]
    that could have produced [H], we get a new subgoal
    requiring us to prove [G] under the assumption that
    [H = C x1 x2 ...] for some arguments [x1 x2 ...] to [C].
    Often it will be the case that some constructors of [T]
    could _not_ have produced [H].  Coq will automatically
    dispatch those subgoals for us which greatly simplifies
    proofs.  This is the primary difference between [destruct]
    and [inversion].

    For [False] (the builtin equivalent to myFalse),
    there are 0 constructors, so when we perform
    inversion, we end up with 0 subgoals and the
    proof is done!
  *)
  inversion H.
Qed.

(**
  When we write a definition using "Lemma" or
  "Theorem", what we're actually doing is giving
  a type [T] and then using tactics to construct a term
  of type [T].

  So far, we generally only do this for types whose
  type is in [Prop], but we can do it for "normal"
  types too:
*)
Lemma foo' :
  Type.
Proof.
  exact bool.
  (** exact (list nat). *)
  (** exact nat. *)
Qed.

(**
  Having a contradiction in a hypothesis will
  let us prove any goal.
*)
```

```
Lemma also_bogus:
  1 = 2 -> False.
Proof.
  intros.
  (**
    [discriminate] is a tactic that looks
    for mismatching constructors in a hypothesis
    and uses that contradiction to prove any goal.
    It is a less-general version of the [congruence]
    tactic we saw in lecture last time.  In Coq,
    there are often many tactics that can solve
    a particular goal.  Sometimes we will use a
    less general tactic because it can help a reader
    understand our proof and will usually be faster.

    Under the hood, [1] looks like [S 0] and
    [2] looks like [S (S 0)]. [discriminate] will
    peel off one [S], to get [0 = S 0]. Since
    [0] and [S] are different constructors of an
    inductive type (where all constructors are
    _distinct_) they are not equal, and Coq can
    use the contradiction to complete our proof.
  *)
  discriminate.
Qed.

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

(**
  Here's [yo], another definition of an empty type.
*)
Inductive yo : Prop :=
| yolo : yo -> yo.

(**
  We will have to do a little more work to
  show that [yo] is empty though.
*)
Lemma yoyo:
  yo -> False.
Proof.
  intros.
  inversion H.
  (** well, that didn't work *)
  induction H.
  assumption. (** but that did! *)
Qed.

(**
  Negation in Coq is encoded in the [not] type.

  It sort of works like our [yoyo] proof above.
*)
Print not.

(** ** Expression Syntax *)

(** Use String and Z notations. *)
Open Scope string_scope.
Open Scope Z_scope.

(**
  Now let's build a programming language.

  We can define the syntax of a language
  as an inductive datatype.
```



```
*)
Inductive expr : Type :=
  (** constant expressions, like [3] or [0] *)
| Eint : Z -> expr
  (** program variables, like ["x"] or ["foo"] *)
| Evar : string -> expr
  (** adding expressions,  [e1 + e2] *)
| Eadd : expr -> expr -> expr
  (** multiplying expressions, [e1 * e2] *)
| Emul : expr -> expr -> expr
  (** comparing expressions, [e1 <= e2] *)
| Elte : expr -> expr -> expr.

(**
  On paper, we would typically write this
  type down using a "BNF grammar" as:
<<
    expr ::= Z
          |  Var
          |  expr + expr
          |  expr * expr
          |  expr <= expr
>>
*)

(**
  Coq provides mechanisms to define
  your own notation which we can use
  to get "concrete syntax".

  Feel free to ignore most of this, especially
  the "level" and "associativity" stuff.
*)
Coercion Eint : Z >-> expr.
Coercion Evar : string >-> expr.
Notation "X [+] Y" := (Eadd X Y)
  (at level 83, left associativity).
Notation "X [*] Y" := (Emul X Y)
  (at level 82, left associativity).
Notation "X [<=] Y" := (Elte X Y)
  (at level 84, no associativity).

Check (1 [+] 2).
Check ("x" [+] 2).
Check ("x" [+] 2 [<=] "y").

(**
  Parsing is a classic CS topic, but won't say
  much more about it in this course.  Parsing is
  still a rich and active research topic, and there
  are many decent tools out there to help practioners
  build good parsers.
*)

(**
  Note that all we've done so far is define the
  _syntax_ of expressions in our language.  We
  have said NOTHING about what these expressions
  _mean_.  In upcoming lectures we will spend
  some time studying how we can describe the meanings
  of programs by giving _semantics_ for our programming
  languages.

  To make this clear, note that operations like addition
  and multiplication are NOT commutative in syntax.  They
  will only be commutative in the meaning of expressions
  later.
*)
```

```
Lemma add_comm_bogus :
  (forall e1 e2, Eadd e1 e2 = Eadd e2 e1) ->
  False.
Proof.
  intros.
  (**
    The [specialize] tactic gives concrete
    arguments to a forall quantified hypothesis.
  *)
  specialize (H 0 1).
  (** inversion is smart *)
  inversion H.
Qed.

(**
  Although we have not yet defined what programs mean,
  we can write some functions to analyze their syntax.

  Here we simply count the number of [Eint] subexpressions
  in a given expression.
*)
Fixpoint nconsts (e: expr) : nat :=
  match e with
  | Eint _ =>
     1 (** same as [S O] *)
  | Evar _ =>
     0 (** same as [O] *)
  | Eadd e1 e2 =>
     nconsts e1 + nconsts e2
     (** same as [plus (nconsts e1) (nconsts e2)] *)
  | Emul e1 e2 =>
      nconsts e1 + nconsts e2
  | Elte e1 e2 =>
      nconsts e1 + nconsts e2
  end.

(**
  We can also use existential quantifiers in Coq.

  To prove an existential, you must provide a
  _witness_, that is, a concrete example of the
  type you're existentially quantifying.
*)
Lemma expr_w_3_consts:
  exists e,
  nconsts e = 3%nat.
Proof.
  (** Here we give a concrete example. *)
  exists (3 [+] 2 [+] 1).
  (** Now we have to show that the example satisfies the property *)
  simpl. reflexivity.
Qed.

(** Compute the size of an expression. *)
Fixpoint esize (e: expr) : nat :=
  match e with
  | Eint _
  | Evar _ =>
      1
  | Eadd e1 e2
  | Emul e1 e2
  | Elte e1 e2 =>
      esize e1 + esize e2
  end.

(**
  Notice how we grouped similar cases together
  in the definition of [esize].  This is just
  sugar, you can see the full definition with:
```



```
*)
Print esize.
(**
<<
esize =
fix esize (e : expr) : nat :=
  match e with
  | Eint _ => 1%nat
  | Evar _ => 1%nat
  | e1 [+] e2 => (esize e1 + esize e2)%nat
  | e1 [*] e2 => (esize e1 + esize e2)%nat
  | e1 [<=] e2 => (esize e1 + esize e2)%nat
  end
     : expr -> nat
>>
*)

(**
  We can use our analyses to prove properties
  of the syntax of programs.

  For example, always have at least as many
  nodes in the AST as constants.
*)
Lemma nconsts_le_size:
  forall e,
  (nconsts e <= esize e)%nat.
Proof.
  intros.
  induction e.
  + simpl. auto.
    (**
      The [auto] tactic will solve many simple
      goals, including those that reflexivity
      would solve.  [auto] also has the property
      that it will never fail.  If it cannot
      solve your goal, then it just does nothing.
      This will be particularly useful when we
      start chaining together sequences of tactics
      to operate simultaneously over multiple subgoals.
    *)
  + simpl. auto.
  (**
    The [omega] will solve many arithemetic goals.
    Unlike [auto], [omega] will fail if it cannot
    solve your goal.
  *)
  + simpl. omega.
  + simpl. omega.
  + simpl. omega.
Qed.

(** that proof had a lot of copy-pasta :( *)
Lemma nconsts_le_size':
  forall e,
  (nconsts e <= esize e)%nat.
Proof.
  intros.
  (**
    Here we see our first "tactic combinator",
    the powerful semicolon ";".

    For any tactics [a] and [b], [a; b] runs
    [a] on the goal and then runs [b] on all
    of the subgoals generate by [a].

    We can chain tactics toghether in this
    way to make shorter, more automated proofs.
```

```
    In the case of this lemma, we'd like to:
<<
     do induction, then
     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 Eadd, Emul, and Elte subgoals remain,
     but it's hard to tell since
     the proof does not "pause".
  *)
Qed.

(**
  Notice how sometime we have to use the scope
  specifier "%nat" so that Coq knows we want
  the [nat] version of some notation instead of
  the [Z] version.

  To figure out where a notation is coming from,
  you can use the [Locate] command:
*)
Locate "<=".

(**
  This generates a lot of output.  In this file
  we really only care about the [nat] and [Z]
  entries:
<<
    "x <= y" := Z.le x y : Z_scope (default interpretation)
    "n <= m" := le n m   : nat_scope
>>
*)

(** We can also print the definition of [le]: *)
Print le.
(**
<<
Inductive le (n : nat) : nat -> Prop :=
  | le_n : (n <= n)%nat
  | le_S : forall m : nat,
             (n <= m)%nat -> (n <= S m)%nat

>>
*)

(**
  [le] is a relation defined as an "inductive predicate".

  We give rules for when the relation holds:
  (1) all nats are less than or equal to themselves and
  (2) if n <= m, then also n <= S m.

  All proofs of [le] are built up from just these
  two constructors!


  We can define our own relations
  to encode properties of expressions.
  In the [has_const] inductive predicate
  below, each constructor corresponds to
  one way you could prove that an expression
  has a constant.
*)
```



```
Inductive has_const : expr -> Prop :=
| hc_in :
    forall c, has_const (Eint c)
| hc_add_l :
    forall e1 e2,
    has_const e1 ->
    has_const (Eadd e1 e2)
| hc_add_r :
    forall e1 e2,
    has_const e2 ->
    has_const (Eadd e1 e2)
| hc_mul_l :
    forall e1 e2,
    has_const e1 ->
    has_const (Emul e1 e2)
| hc_mul_r :
    forall e1 e2,
    has_const e2 ->
    has_const (Emul e1 e2)
| hc_cmp_l :
    forall e1 e2,
    has_const e1 ->
    has_const (Elte e1 e2)
| hc_cmp_r :
    forall e1 e2,
    has_const e2 ->
    has_const (Elte e1 e2).

(**
  Similarly, we can define a relation
  that holds on expressions that contain
  a variable.
*)
Inductive has_var : expr -> Prop :=
| hv_var :
    forall s, has_var (Evar s)
| hv_add_l :
    forall e1 e2,
    has_var e1 ->
    has_var (Eadd e1 e2)
| hv_add_r :
    forall e1 e2,
    has_var e2 ->
    has_var (Eadd e1 e2)
| hv_mul_l :
    forall e1 e2,
    has_var e1 ->
    has_var (Emul e1 e2)
| hv_mul_r :
    forall e1 e2,
    has_var e2 ->
    has_var (Emul e1 e2)
| hv_cmp_l :
    forall e1 e2,
    has_var e1 ->
    has_var (Elte e1 e2)
| hv_cmp_r :
    forall e1 e2,
    has_var e2 ->
    has_var (Elte e1 e2).

(**
  We can also write boolean functions
  that check the same properties.

  Note that [orb] is disjuction over
  booleans:
*)
Print orb.
```

```
Fixpoint hasConst (e: expr) : bool :=
  match e with
  | Eint _ => true
  | Evar _ => false
  | Eadd e1 e2 => orb (hasConst e1) (hasConst e2)
  | Emul e1 e2 => orb (hasConst e1) (hasConst e2)
  | Elte e1 e2 => orb (hasConst e1) (hasConst e2)
  end.

(**
  We can write that a little more compactly using
  the "||" notation for [orb] provided by
  the Bool library.
*)
Fixpoint hasVar (e: expr) : bool :=
  match e with
  | Eint _ => false
  | Evar _ => true
  | Eadd e1 e2 => hasVar e1 || hasVar e2
  | Emul e1 e2 => hasVar e1 || hasVar e2
  | Elte 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 shows that the [hasConst] _function_ is COMPLETE
  with respect to the relation [has_const].
  Thus, anything that satisfies the relation evaluates
  to "true" under the function [hasConst].
*)
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.
    - apply IHe1 in H1; rewrite H1; auto.
    - apply IHe2 in H1; rewrite H1;
      rewrite orb_comm; auto.
  + (** Lte case is similar *)
    inversion H; simpl; subst.
    - apply IHe1 in H1; rewrite H1; auto.
    - apply IHe2 in H1; rewrite H1;
      rewrite orb_comm; auto.
Qed.

(**
  Now for the other direction.

  Here we'll prove that the [hasConst] _function_
  is SOUND with respect to the relation.
  That is, if [hasConst] produces true,
  then there is some proof of the inductive
  relation [has_const].
*)
Lemma hasConst_has_const:
  forall e,
  hasConst e = true ->
  has_const e.
Proof.
  intros.
  induction e.
  + simpl.
    (** we can prove this case with a constructor *)
    constructor. (** 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. *)
    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; apply orb_true_iff in H; destruct H.
     - (** constructor will just use hc_mul_l *)
       constructor. apply IHe1. assumption.
     - (** constructor will screw up and try hc_mul_l again! *)
       (** constructor is rather dim *)
       constructor. (** OOPS! *)
       Undo.
```

```
      apply hc_mul_r. apply IHe2. assumption.
  + (** Lte case is similar *)
    simpl in H; apply orb_true_iff in H; destruct H.
    - constructor; auto.
    - apply hc_cmp_r; auto.
Qed.

(** we can stitch these two lemmas 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.


(**
  Notice all that work was only for the "true" cases!

  We can prove analogous facts for the "false" cases too.

  Here we will prove the "false" cases directly.
  However, note that you could use [has_const_iff_hasConst]
  to get a much simpler proof.
*)

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 IHe1. intro.
      apply H. apply hc_add_l. assumption.
    - apply IHe2. intro.
      apply H. apply hc_add_r. assumption.
  + (** Mul case is similar *)
    simpl; apply orb_false_iff.
    split.
    - apply IHe1; intro.
      apply H. apply hc_mul_l. assumption.
    - apply IHe2; intro.
      apply H. apply hc_mul_r. assumption.
  + (** Lte case is similar *)
    simpl; apply orb_false_iff.
    split.
    - apply IHe1; intro.
      apply H. apply hc_cmp_l. assumption.
    - apply IHe2; intro.
      apply H. apply hc_cmp_r. assumption.
Qed.

(**
  Here is a more direct proof based on the
  iff we proved for the true case.
```



```
*)
Lemma not_has_const_hasConst':
  forall e,
  ~ has_const e ->
  hasConst e = false.
Proof.
  intros.
  (** do case analysis on hasConst e *)
  (** eqn:? remembers the result in a hypothesis *)
  destruct (hasConst e) eqn:?.
  - rewrite <- has_const_iff_hasConst in Heqb.
    (** now we have hasConst e = true in our hypothesis *)

    (** We have a contradiction in our hypotheses *)
    (** discriminate won't work this time though *)
    unfold not in H.
    apply H in Heqb.
    inversion Heqb.
  - reflexivity.
    (** For the other case, this is easy *)
Qed.

(** Now the other direction of the false case *)
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 *)
    apply orb_false_iff in H; destruct H.
    inversion H0; subst; auto.
  + (** Lte 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_const e.
Proof.
  intros.
  (** ~ X is just X -> False *)
  unfold not.
  intros.
  rewrite has_const_iff_hasConst in H0.
  rewrite H 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.
    constructor.
  + (** prove right side of disjunction *)
    right.
    constructor.
  + (** 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_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; 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.
```



```
  + 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; 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.
    apply hasConst_has_const; auto.
  - reflexivity.
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 in Heqb.
    (** NOTE: we got another subgoal! *)
    * discriminate.
    * assumption.
Qed.

(**
  In general:
   Relational defns are nice when you want to use inversion.
   Functional defns are nice when you want to use simpl.
*)
```