Inductive binop : Set :=
| Add | Sub | Mul | Div | Mod | Lt | Lte | Conj | Disj.

Inductive expr : Set :=
Coercion Int : Z >−> expr.
Coercion Var : string >−> expr.
Notation "X \[+\] Y" := (BinOp Add X Y) (at level 51, left associativity).
Notation "X \[−\] Y" := (BinOp Sub X Y) (at level 51, left associativity).
Notation "X \[*\] Y" := (BinOp Mul X Y) (at level 50, left associativity).
Notation "X \[/\] Y" := (BinOp Div X Y) (at level 50, left associativity).
Notation "X \[%\] Y" := (BinOp Mod X Y) (at level 50, left associativity).
Notation "X \[<\] Y" := (BinOp Lt X Y) (at level 52).
Notation "X \[<=\] Y" := (BinOp Lte X Y) (at level 52).
Notation "X \[&&\] Y" := (BinOp Conj X Y) (at level 53, left associativity).
Notation "X \[||\] Y" := (BinOp Disj X Y) (at level 54, left associativity).

Definition fib_x_y : stmt :=
"y"  <− 0;;
"y0"  <− 1;;
"y1"  <− 0;;
"i"  <− 0;;
while ("i" \[<\] "x") {{
  "y"  <- "y0"  [+] "y1";;
  "y0"  <- "y1";;
  "y1"  <- "y";;
  "i"  <- "i"  [+] 1
}}.

Definition gcd_xy_i : stmt :=
"i"  <− "x";;

(* SYNTAX *)

(* SEMANTICS *)

(* Heaps :
To evaluate an expression containing variables, we need some representation of memory to get the value of variables from.
We need to model memory as some mapping from variables to ints. Functions can do just that! *)

Definition heap : Type :=
  string → Z.

(* The empty memory just maps everything to zero. *)

Definition empty : heap :=
  fun v => 0.

(* We will also need to evaluate our operators over ints. Since there's a bunch, we'll define a helper function for this. *)

Definition exec_op (op: binop) (i1 i2: Z) : Z :=
match op
with
| Add => i1 + i2  | Sub => i1 − i2  | Mul => i1 * i2  | Div => i1 / i2  | Mod => i1 mod i2  | Lt  => if Z_lt_dec i1 i2 then 0 else 1  | Disj => if Z_eq_dec i1 0 then              if Z_eq_dec i2 0 then 1 else 1            else 1  end.

(* Now we can define a relation to capture the semantics of expressions *)

Inductive eval : heap → expr → Z → Prop :=
  eval_int:
    forall h i, eval h (Int i) i
  eval_var:
    forall h v, eval h (Var v) (h v)
  eval_binop:
    forall h op e1 e2 i1 i2 i3, eval h (BinOp op e1 e2) i3 →
    eval h e1 i1 → eval h e2 i2 → eval h (BinOp op e1 e2) i3.
Lemma eval_ex1:
evempty ("x" [+ ] 1) 1.
Proof.
- apply eval_binop with (i1 := 0).
- apply eval_var.
- apply eval_int.
- simpl. reflexivity.
Qed.

Lemma eval_ex2:
evempty ("x" [+] 1) 2.
Proof.
unfold not.
inintro.
inversion H. subst.
inversion H4.
unfold empty in H1. subst.
inversion H6. subst.
simpl in H7.
discriminate.
Qed.

(* We can also define an interpreter for expressions. *)

Fixpoint interp_expr (h: heap) (e: expr) : Z :=
match e with
| Int i => i
| Var v => h v
| BinOp op e1 e2 =>
  exec_op op (interp_expr h e1) (interp_expr h e2)
end. Eval cbv in (interp_expr empty ("x" [+] 1)).

(* ... and prove relational and functional versions agree *)

Lemma interp_expr_ok:
forall h e i, interp_expr h e = i ⇒ eval h e i.Proof.
inintro.
inversion e; simpl in *; intros.
- subst; constructor.
- subst; constructor.
- (** OK, now IHe1 and IHe2 look stronger *)
  apply eval_binop with (i1 := interp_expr h e1).
  + apply IHe1. auto.
  + apply IHe2. auto.
  + assumption.
Qed.

(* interp_expr_ok only shows that if the interpreter produces 'i' as the result of evaluating expr 'e' in heap 'h', then eval relates 'h', 'e', and 'i' as well. We can prove the other direction: if the eval relates 'h', 'e', and 'i', then the interpreter will produce 'i' as the result of evaluating expr 'e' in heap 'h'. *)

Lemma eval_interp:
forall h e i, eval h e i ⇒ interp_expr h e = i.Proof.
inintro.
inversion e; simpl in *; intros.
- inversion H.
- subst; reflexivity.
- inversion H; subst; reflexivity.
- inversion H; subst; rewrite (IHe1 i1 H4).
- (** we can "fill in" an equality to rewrite with *)
  rewrite (IHe2 i2 H6).
  reflexivity.
Qed.

(* we actually could have proved the above lemma in an even cooler way: by doing induction on the derivation of eval! *)

Lemma eval_interp':
forall h e i, eval h e i ⇒ interp_expr h e = i.Proof.
inintro.
inversion H; subst; reflexivity.
inversion H; subst; rewrite (IHe1 i1 H4).
rewrite (IHe2 i2 H6).
reflexivity.
Qed.

(* notice how much cleaner that was! *)

(* we can also write the one of the earlier lemmas in a slightly cleaner way *)
Lemma interp_eval:
  forall h e, eval h (interp_expr h e).
Proof.
  intros. induction e; simpl.
  - constructor.
  - constructor.
  - (** 'constructor.' will not work here because
    the goal does not unify with the eval_binop
    case. 'econstructor' is a more flexible
    version of constructor that introduces
    existentials that will allow things to
    unify behind the scenes. Check it out! *)
    econstructor.
  + (** 'assumption.' will not work here because
    our goal has an existential in it. However
    'eassumption' knows how to handle it! *)
    eassumption.
  + reflexivity.
Qed.

(** OK, so we've shown that our relational semantics
  for expr agrees with our functional interpreter.    One nice consequence of this is that we can easily
  show that our eval relation is deterministic. *)

Lemma eval_det:
  forall h e i1 i2,
  eval h e i1 -> eval h e i2 ->
  i1 = i2.
Proof.
  intros.
  apply interp_eval in H.
  subst. reflexivity.
Qed.

(** it's a bit more work without interp, but not too bad *)

Lemma eval_swap_add:
  forall h e1 e2 i,
  eval h (BinOp Add e1 e2) i <-> eval h (BinOp Add e2 e1) i.
Proof.
  split; intros.
  - inversion H. subst.
    econstructor.
  - inversion H; subst.
    eauto.    
  - simplified. omega.
  - inversion H; subst.
  - inversion H; subst.
    econstructor; eauto.
  - simplified. omega.
Qed.

Lemma interp_expr_swap_add:
  forall h e1 e2,
  interp_expr h (BinOp Add e1 e2) = interp_expr h (BinOp Add e2 e1).
Proof.
  intros; simplified. omega.
Qed.

Lemma eval_add_zero:
  forall h e i,
  eval h (BinOp Add e (Int 0)) i <-> eval h e i.
Proof.
  split; intros.
  - inversion H; subst.
    cut (i1 + 0 = i1).    + intros. rewrite H0.      assumption.    + omega.
  - inversion H; subst.
  - inversion H; subst.
  - simplified. omega.
  - simplified. omega.
  (** replace lets us rewrite a subterm *)
  replace (i1 + 0) with i by omega.
  (** replace lets us rewrite a subterm *)
  replace (i1 + 0) with i1 by omega.
  assumption.        
  - econstructor; eauto.
  - econstructor; eauto.
  - simplified. omega.
Qed.

Lemma interp_expr_add_zero:
  forall h e,
  interp_expr h (BinOp Add e (Int 0)) = interp_expr h e.
Proof.
  intros; simplified. omega.
Qed.
Lemma eval_mul_zero:
  forall h e i, eval h (BinOp Mul e (Int 0)) i <−> i = 0.
Proof.  split; intros.  − inversion H; subst.  + simpl. omega.  − subst. pose (interp_expr h e). eapply eval_binop with (i1 := z).  + apply interp_expr_ok. auto.  + econstructor; eauto.  + simpl. omega.
Qed.

Lemma interp_expr_mul_zero:
  forall h e, interp_expr h (BinOp Mul e (Int 0)) = 0.
Proof.  intros; simpl. omega. Qed.

(* Huh, so why ever have relational semantics? *)

(* To define the semantics for statements, we'll need to be able to update the heap. *)

Definition update (h: heap) (v: string) (i: Z) : heap :=
  fun v' =>
    if string_dec v' v then
      i
    else
      h v'.

Inductive step : heap −> stmt −> heap −> stmt −> Prop :=
  | step_assign:  forall h v e i, eval h e i ->
    step h (Assign v e) (update h v i) Nop
  | step_seq_nop:  forall h s, step h s h s i
  | step_while_true:  forall h e s i, eval h e i ->
    step h (While e s) h (Seq s (While e s))
  | step_while_false:  forall h e s i, eval h e i ->
    i = 0 ->
    step h (While e s) h Nop.

(** note that there are several other ways we could have done semantics for while *)

(** We can also define an interpreter to run a single step of a stmt, but we'll have to learn some new types to write it down.

Note that, unlike eval, the step relation is partial: not every heap and stmt is related to another heap and stmt! *)

Lemma step_partial:
  exists h, exists s, forall h' s', ~ step h s h' s'.
Proof.  exists empty.  exists Nop. intros. unfold not. intros. inversion H. (* umpossible! *)
Qed.

(* In general, we say that any stmt that cannot step is "stuck" *)

Definition stuck (s: stmt) : Prop :=
  forall h h' s', ~ step h s h' s'.

Lemma nop_stuck:
  stuck Nop.
Proof. unfold stuck, not; intros. inversion H.
Qed.

(* Since the step relation is partial, but all functions have to be total, we will use the 'option' type to represent the results of step interpreter. *)

Print option.

(** We could define our interpreter this way, but we end up with a case explosion in the Seq nop / non-nop cases... *)

(*
Fixpoint interp_step (h: heap) (s: stmt) : option (heap * stmt) :=
  match s with
  | Nop => None
  | Assign v e =>
    ...
  | Seq s1 s2 =>
    ...
  | Cond e s1 s2 =>
    ...
  | While e s1 s2 =>
    ...
  | Some (h, s1) => Some (h, s1)
  | None => None
  end

Printed by Zach Tatlock Thursday October 22, 2015 L04/L04_in_class.v 4/6
else
  | While e s =>
    if Z_eq_dec (interp_expr h e) 0 then
      Some (h, Nop)
    else
      Some (h, Seq s (While e s))
  end.

(*
(** So instead, we’ll define a helper to simplify the match. *)

Definition isNop (s: stmt) : bool :=
  match s
  | Nop => true
  | _ => false
  end.

Lemma isNop_ok:
  forall s, isNop s = true <-> s = Nop.
Proof.
  (** a lot of times we don’t really need intros *)
  destruct s; simpl; split; intros;    auto; discriminate.
Qed

Fixpoint interp_step (h: heap) (s: stmt) : option (heap * stmt) :=
  match s
  | Nop => None
  | Assign v e =>
    Some (update h v (interp_expr h e), Nop)
  | Seq s1 s2 =>
    if isNop s1 then
      Some (h, s2)
    else
      match interp_step h s1 with
      | Some (h', s1') => Some (h', Seq s1' s2)
      | None => None
      end
  | Cond e s =>
    if Z_eq_dec (interp_expr h e) 0 then
      Some (h, Nop)
    else
      Some (h, s)
  | While e s =>
    if Z_eq_dec (interp_expr h e) 0 then
      Some (h, Nop)
    else
      Some (h, Seq s (While e s))
  end.

(*
(** and we can prove that our step interpreter
    agrees with our relational semantics *)

Lemma interp_step_ok:
  forall h s s' h',
  interp_step h s = Some (h', s') ->
  step h s h' s'.
Proof.
  intros h s. revert h.
  induction s; simpl; intros.
  - discriminate.
  - inversion H. subst.
  - constructor. apply interp_eval.
  - destruct (isNop s) eqn:?:
    (** use the weird 'eqn:? after a destruct
        to remember what you destructed! *)
    + rewrite isNop_ok in Heqo. constructor. assumption.
  - discriminate.
  - destructor (Z.eq_dec (interp_expr h e) 0) eqn:?:
    + inversion H. subst.
    + eapply step_while_false; eauto.
  - eapply step_while_true; eauto.
  Qed

(*
(** So far, 'step' only does one "step" of
    an execution of a stmt. We can build
    the transitive closure of this relation
    through to reason about with more than one step. *)

Inductive step_n : heap -> stmt -> nat -> heap -> stmt -> Prop :=
  | sn_refl:  forall h s, step_n h s 0 h s
  | sn_step:
    forall h1 s1 n h2 s2 h3 s3,
    step_n h1 s1 n h2 s2 ->
    step_n h2 s3 h3 s3 ->
    step_n h1 s1 (S n) h3 s3.

(*
(** Notice how we "add a step" to the end for step_n.
    We can also "add a step" on the beginning. *)

Lemma step_n_left:
  forall h1 s1 h2 s2 n h3 s3,
  step_n h1 s1 h2 s2 n h3 s3 ->
  step_n h1 s1 n h2 s2 ->
  step_n h1 s1 (S n) h3 s3.
Proof.
  intros. induction H0.
  - eapply step_cond_true; eauto.
  - eapply interp_eval; eauto.
  Qed

(*
(** Defining an interpreter for more than one step is trickier!

\[\text{\small L04/L04_in_class.v 5/6}  \]
Since a stmt may not terminate, we can’t just naively write a recursive function to run a stmt. Instead, we’ll use a notion of "fuel" to guarantee that our function always terminates.

Fixpoint run (fuel: nat) (h: heap) (s: stmt) : option (heap * stmt) :=
  match fuel
  with
  | O => None
  | S n =>
    match interp_step h s with
    | Some (h', s') => run n h' s'
    | None => Some (h, s) (** why not None? *)
  end.

(** and we can verify our interpreter too *)

Lemma run_ok:forall fuel h s h' s', run fuel h s = Some (h', s') -> exists n, step_n h s n h' s'.
Proof. induction fuel; simpl; intros.
- discriminate.
- destruct (interp_step h s) as [[foo bar]] eqn:?
- apply HHfuel in H.
  - apply interp_step_ok in Heqo.
  - destruct H. exists (S x).
    eapply step_n_left; eauto.
  - inversion H; subst.
    exists O. constructor; auto.
Qed.

(** TAH DAH! We have a verified interpreter! *)

Extraction interp_expr.
Extraction interp_step.
Extraction run.

Extraction fib_x_y.
Extraction gcd_xy_i.

(** we can also make versions of our programs
that set up an useful initial heaps *)

Definition fib_prog (x: Z) : stmt :=
  "x" <- x;
  fib_x_y.

Definition gcd_prog (x y: Z) : stmt :=
  "x" <- x;
  "y" <- y;
  gcd_xy_i.

(** use analogous OCaml types *)
Require Import ExtrOcamlBasic.
Require Import ExtrOcamlNatInt.
Require Import ExtrOcamlZInt.
Require Import ExtrOcamlString.

(** we can even put these in a file and run them! *)
Cd "~/505-au15/www/L04/"
Extraction "Imp.ml" run empty fib_prog gcd_prog.