```
(** * Lecture 6 *)

Require Import Bool.
Require Import ZArith.
Require Import IMPSyntax.
Require Import IMPSemantics.

Ltac break_match :=
  match goal with
    | _ : context [ if ?cond then _ else _ ] |- _ =>
     destruct cond as [] eqn:?
    | |- context [ if ?cond then _ else _ ] =>
      destruct cond as [] eqn:?
    | _ : context [ match ?cond with _ => _ end ] |- _ =>
     destruct cond as [] eqn:?
    | |- context [ match ?cond with _ => _ end ] =>
      destruct cond as [] eqn:?
  end.

Open Scope Z_scope.

Check 1.
Check 1%nat.

(** ** A Verified Analysis *)

Inductive expr_non_neg : expr -> Prop :=
| NNInt :
    forall i,
      0 <= i ->
      expr_non_neg (Int i)
| NNVar :
    forall v,
      expr_non_neg (Var v)
| NNBinOp :
    forall op e1 e2,
      op <> Sub ->
      expr_non_neg e1 ->
      expr_non_neg e2 ->
      expr_non_neg (BinOp op e1 e2).

Definition isSub (op: binop) : bool :=
  match op with
    | Sub => true
    | _ => false
  end.

Lemma isSub_ok:
  forall op,
    isSub op = true <-> op = Sub.
Proof.
  destruct op; split; simpl; intros;
    auto || discriminate.
Qed.

Lemma notSub_ok:
  forall op,
    isSub op = false <-> op <> Sub.
Proof.
  unfold not; destruct op;
    split; simpl; intros;
    auto; try discriminate.
  exfalso; auto.
Qed.

Print sumbool.
Set Printing All.
Print sumbool.
Unset Printing All.
```



```
Check Z_le_dec.

Fixpoint expr_nn (e: expr) : bool :=
  match e with
    | Int i =>
      (** won't work! wrong type (sumbool) *)
      (** Z_le_dec 0 i *)
      if Z_le_dec 0 i then true else false
    | Var v =>
      true
    | BinOp op e1 e2 =>
      negb (isSub op) && expr_nn e1 && expr_nn e2
  end.

(** && vs. /\ *)

Check andb.
Check and.

Locate "&&".
Locate "/\".

Lemma expr_nn_expr_non_neg:
  forall e,
    expr_nn e = true ->
    expr_non_neg e.
Proof.
  induction e; simpl; intros.
  - break_match.
    + constructor; auto.
    + discriminate.
  - constructor; auto.
  - SearchAbout andb.
    Check andb_true_iff.
    apply andb_true_iff in H. destruct H.
    apply andb_true_iff in H. destruct H.
    Check NNBinOp.
    constructor; auto.
    SearchAbout negb.
    Check negb_sym.
    symmetry in H.
    apply negb_sym in H. simpl in H.
    apply notSub_ok in H. assumption.
Qed.

Lemma expr_non_neg_expr_nn:
  forall e,
    expr_non_neg e ->
    expr_nn e = true.
Proof.
  induction 1; simpl; auto.
  - break_match; auto.
  - apply andb_true_iff; split; auto.
    apply andb_true_iff; split; auto.
    symmetry. apply negb_sym; simpl.
    apply notSub_ok; auto.
Qed.

Definition heap_non_neg (h: heap) : Prop :=
  forall v, 0 <= h v.

Lemma non_neg_exec_op:
  forall op i1 i2,
    op <> Sub ->
    0 <= i1 ->
    0 <= i2 ->
    0 <= exec_op op i1 i2.
Proof.
  (** TODO good exercise to learn Z lemmas *)
```

```
Admitted.

Lemma non_neg_eval:
  forall h e i,
    heap_non_neg h ->
    expr_non_neg e ->
    eval h e i ->
    0 <= i.
Proof.
  unfold heap_non_neg. induction 3.
  - inversion H0. auto.
  - apply H.
  - inversion H0; subst.
    (** auto will do a lot of work! *)
    apply non_neg_exec_op; auto.
Qed.

Inductive stmt_non_neg : stmt -> Prop :=
| NNNop :
    stmt_non_neg Nop
| NNAssign :
    forall v e,
      expr_non_neg e ->
      stmt_non_neg (Assign v e)
| NNSeq :
    forall s1 s2,
      stmt_non_neg s1 ->
      stmt_non_neg s2 ->
      stmt_non_neg (Seq s1 s2)
| NNCond :
    forall e s,
      stmt_non_neg s ->
      stmt_non_neg (Cond e s)
| NNWhile :
    forall e s,
      stmt_non_neg s ->
      stmt_non_neg (While e s).

Fixpoint stmt_nn (s: stmt) : bool :=
  match s with
    | Nop => true
    | Assign v e => expr_nn e
    | Seq s1 s2 => stmt_nn s1 && stmt_nn s2
    | Cond e s => stmt_nn s
    | While e s => stmt_nn s
  end.

(**

<<

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

>>
```



```
*)

Lemma stmt_nn_stmt_non_neg :
  forall s,
    stmt_nn s = true ->
    stmt_non_neg s.
Proof.
  induction s; simpl; intros;
    constructor; auto.
  (** from Assign constructor *)
  - apply expr_nn_expr_non_neg; auto.
  (** both of these from Seq constructor *)
  - apply andb_true_iff in H. destruct H; auto.
  - apply andb_true_iff in H. destruct H; auto.
Qed.

Lemma stmt_non_neg_stmt_nn:
  forall s,
    stmt_non_neg s ->
    stmt_nn s = true.
Proof.
  induction 1; simpl; intros; auto.
  - apply expr_non_neg_expr_nn; auto.
  - apply andb_true_iff; split; auto.
Qed.

Lemma non_neg_step:
  forall h s h' s',
    heap_non_neg h ->
    stmt_non_neg s ->
    step h s h' s' ->
    heap_non_neg h' /\ stmt_non_neg s'.
Proof.
  unfold heap_non_neg; intros.
  induction H1.
  - split; intros.
    + unfold update.
      break_match; subst; auto.
      eapply non_neg_eval; eauto.
      inversion H0; subst; auto.
    + apply NNNop. (** constructor. *)
  - split; intros; auto.
    inversion H0; subst.
    assumption.
  - inversion H0; subst.
    apply IHstep in H4; auto. destruct H4.
    split; intros; auto.
    constructor; auto.
  - split; intros; auto.
    inversion H0; subst; auto.
  - split; intros; auto.
    constructor.
  - split; intros; auto.
    inversion H0; subst; auto.
    constructor; auto.
  - split; intros; auto.
    constructor.
Qed.

Lemma non_neg_step_n:
  forall h s n h' s',
    heap_non_neg h ->
    stmt_non_neg s ->
    step_n h s n h' s' ->
    heap_non_neg h' /\ stmt_non_neg s'.
Proof.
  intros. induction H1; auto.
  apply non_neg_step in H1; auto.
```

```
  destruct H1.
  apply IHstep_n; auto.
Qed.

(** ** Termination *)

Lemma can_step:
  forall s,
    s <> Nop ->
    forall h, exists h', exists s', step h s h' s'.
Proof.
  (** TODO a good exercise *)
Admitted.

Definition diverges (s: stmt) : Prop :=
  forall h n,
  exists h', exists s',
    step_n h s n h' s'.

Definition furnace : stmt :=
  while 1 {{ Nop }}.

(** stupid auto indent *)
Ltac zex x := exists x.

Lemma warming_up:
  diverges furnace.
Proof.
  unfold diverges. intros.
  induction n.
  - zex h. zex furnace. constructor.
  - destruct IHn as [h' [s' H]].
    (** hmm, need to add next step to the *end* *)
Abort.

Lemma step_n_r:
  forall h1 s1 n h2 s2 h3 s3,
    step_n h1 s1 n h2 s2 ->
    step h2 s2 h3 s3 ->
    step_n h1 s1 (S n) h3 s3.
Proof.
  intros. induction H.
  - econstructor; eauto.
    constructor.
  - econstructor; eauto.
Qed.

Lemma warming_up:
  diverges furnace.
Proof.
  unfold diverges. intros.
  induction n.
  - zex h. zex furnace. constructor.
  - destruct IHn as [h' [s' H]].
    (** just need to show that s' can take one more step *)
    (** we know everything but Nop can step... *)
    (** hmmm, do not know much about h' s' *)
    (** need stronger IH ! *)
Abort.

Lemma warming_up:
  forall h n,
  exists h', exists s',
    step_n h furnace n h' s' /\
    s' <> Nop.
Proof.
  intros. induction n.
  - zex h. zex furnace.
    split.
```



```
    + constructor.
    + discriminate.
  - destruct IHn as [h' [s' [Hs Hn]]].
    destruct (can_step s' Hn h') as [h'' [s'' HS]].
    zex h''; zex s''. split.
    + eapply step_n_r; eauto.
    + (** ugh, don't know enough about s'' ! *)
      (** IH still too weak *)
Abort.

Lemma warming_up:
  forall h n,
  exists h',
    step_n h furnace n h' furnace.
Proof.
  intros. induction n.
  - zex h. constructor.
  - destruct IHn as [h' IH].
    eexists. eapply step_n_r; eauto.
    (** stuck! furnance doesn't step to itself! *)
    (** IH too strong!!! *)
Abort.

Definition furnace' : stmt :=
  Nop ;; while 1 {{ Nop }}.

Lemma warming_up:
  forall h n,
  exists h',
    step_n h furnace n h' furnace \/
    step_n h furnace n h' furnace'.
Proof.
  intros. induction n.
  - zex h. left. constructor.
  - destruct IHn as [h' [IH | IH]].
    + eexists. right.
      eapply step_n_r; eauto.
      unfold furnace'.
      econstructor; eauto.
      econstructor; eauto.
      omega.
    + eexists. left.
      eapply step_n_r; eauto.
      unfold furnace'.
      econstructor; eauto.
Qed.
```