```
(** * IMP Semantics *)

Require Import ZArith.
Require Import String.

Open Scope string_scope.
Open Scope Z_scope.

Require Import IMPSyntax.

Inductive value : Set :=
| Vint : Z -> value
| Vpair : value -> value -> value.

Definition heap : Type :=
  string -> value.

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

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 1 else 0
  | Lte => if Z_le_dec i1 i2 then 1 else 0
  | Conj => if Z_eq_dec i1 0 then 0 else
              if Z_eq_dec i2 0 then 0 else 1
  | Disj => if Z_eq_dec i1 0 then
              if Z_eq_dec i2 0 then 0 else 1
            else 1
  end.

Inductive eval : heap -> expr -> value -> Prop :=
| eval_int:
    forall h i,
    eval h (Int i) (Vint i)
| eval_var:
    forall h v,
    eval h (Var v) (h v)
| eval_binop:
    forall h op e1 e2 i1 i2 i3,
    eval h e1 (Vint i1) ->
    eval h e2 (Vint i2) ->
    exec_op op i1 i2 = i3 ->
    eval h (BinOp op e1 e2) (Vint i3)
| eval_pair:
    forall h e1 e2 v1 v2,
      eval h e1 v1 ->
      eval h e2 v2 ->
      eval h (Pair e1 e2) (Vpair v1 v2)
| eval_projl:
    forall h e v1 v2,
      eval h e (Vpair v1 v2) ->
      eval h (ProjL e) v1
| eval_projr:
    forall h e v1 v2,
      eval h e (Vpair v1 v2) ->
      eval h (ProjR e) v2
.

Fixpoint interp_expr (h: heap) (e: expr) : option value :=
  match e with
  | Int i => Some (Vint i)
  | Var v => Some (h v)
  | BinOp op e1 e2 =>
```



```
    match interp_expr h e1, interp_expr h e2 with
      | Some (Vint i1), Some (Vint i2) =>
        Some (Vint (exec_op op i1 i2))
      | _, _ => None
    end
  | Pair e1 e2 =>
    match interp_expr h e1, interp_expr h e2 with
      | Some v1, Some v2 =>
        Some (Vpair v1 v2)
      | _, _ => None
    end
  | ProjL e =>
    match interp_expr h e with
      | Some (Vpair v1 v2) => Some v1
      | _ => None
    end
  | ProjR e =>
    match interp_expr h e with
      | Some (Vpair v1 v2) => Some v2
      | _ => None
    end
  end.

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.

Ltac inv H := inversion H; subst.

Lemma interp_expr_eval:
  forall h e v,
  interp_expr h e = Some v ->
  eval h e v.
Proof.
  induction e; simpl; intros.
  - inv H. constructor.
  - inv H. constructor.
  - repeat break_match; subst; try discriminate.
    inv H. econstructor; eauto.
  - repeat break_match; subst; try discriminate.
    inv H. econstructor; eauto.
  - repeat break_match; subst; try discriminate.
    inv H. econstructor; eauto.
  - repeat break_match; subst; try discriminate.
    inv H. econstructor; eauto.
Qed.

Lemma eval_interp_expr:
  forall h e v,
  eval h e v ->
  interp_expr h e = Some v.
Proof.
  induction 1; simpl; auto.
  - repeat break_match; subst; try discriminate.
    inv IHeval1; inv IHeval2; auto.
  - repeat break_match; subst; try discriminate.
    inv IHeval1; inv IHeval2; auto.
  - repeat break_match; subst; try discriminate.
    inv IHeval; auto.
  - repeat break_match; subst; try discriminate.
    inv IHeval; auto.
```

```
Qed.

Lemma eval_det:
  forall h e v1 v2,
  eval h e v1 ->
  eval h e v2 ->
  v1 = v2.
Proof.
  intros.
  apply eval_interp_expr in H.
  apply eval_interp_expr in H0.
  rewrite H in H0; inv H0; auto.
Qed.

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

Inductive step : heap -> stmt -> heap -> stmt -> Prop :=
| step_assign:
  forall h x e v,
  eval h e v ->
  step h (Assign x e) (update h x v) Nop
| step_seq_nop:
  forall h s,
  step h (Seq Nop s) h s
| step_seq:
  forall h s1 s2 s1' h',
  step h s1 h' s1' ->
  step h (Seq s1 s2) h' (Seq s1' s2)
| step_cond_true:
  forall h e s i,
  eval h e (Vint i) ->
  i <> 0 ->
  step h (Cond e s) h s
| step_cond_false:
  forall h e s i,
  eval h e (Vint i) ->
  i = 0 ->
  step h (Cond e s) h Nop
| step_while_true:
  forall h e s i,
  eval h e (Vint i) ->
  i <> 0 ->
  step h (While e s) h (Seq s (While e s))
| step_while_false:
  forall h e s i,
  eval h e (Vint i) ->
  i = 0 ->
  step h (While e s) h Nop.

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

Lemma isNop_ok:
  forall s,
  isNop s = true <-> s = Nop.
Proof.
  destruct s; simpl; split; intros;
    auto; discriminate.
Qed.

Fixpoint interp_step (h: heap) (s: stmt) : option (heap * stmt) :=
```



```
  match s with
  | Nop => None
  | Assign x e =>
    match interp_expr h e with
      | Some v => Some (update h x v, Nop)
      | None => None
    end
  | 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 =>
    match interp_expr h e with
      | Some (Vint i) =>
        if Z_eq_dec i 0 then
          Some (h, Nop)
        else
          Some (h, s)
      | _ => None
    end
  | While e s =>
    match interp_expr h e with
      | Some (Vint i) =>
        if Z_eq_dec i 0 then
          Some (h, Nop)
        else
          Some (h, Seq s (While e s))
      | _ => None
    end
  end.

Lemma interp_step_step:
  forall s h h' s',
  interp_step h s = Some (h', s') ->
  step h s h' s'.
Proof.
  induction s; simpl; intros.
  - discriminate.
  - break_match; try discriminate.
    apply interp_expr_eval in Heqo.
    inv H. econstructor; eauto.
  - repeat break_match; try discriminate.
    + apply isNop_ok in Heqb; subst.
      inv H. econstructor; eauto.
    + inv H. apply IHs1 in Heqo.
      econstructor; eauto.
  - repeat break_match; subst; try discriminate.
    + inv H. apply interp_expr_eval in Heqo.
      econstructor; eauto.
    + inv H. apply interp_expr_eval in Heqo.
      econstructor; eauto.
  - repeat break_match; subst; try discriminate.
    + inv H. apply interp_expr_eval in Heqo.
      econstructor; eauto.
    + inv H. apply interp_expr_eval in Heqo.
      econstructor; eauto.
Qed.

Lemma step_interp_step:
  forall h s h' s',
  step h s h' s' ->
  interp_step h s = Some (h', s').
Proof.
  induction 1; simpl; auto.
  - break_match.
```

```
    + apply eval_interp_expr in H.
      rewrite H in Heqo; inv Heqo; auto.
    + apply eval_interp_expr in H.
      rewrite H in Heqo; discriminate.
  - break_match.
    + apply isNop_ok in Heqb; subst.
      inv H.
    + rewrite IHstep; auto.
  - apply eval_interp_expr in H.
    break_match; inv H.
    break_match; subst; auto.
    congruence.
  - apply eval_interp_expr in H.
    break_match; inv H.
    break_match; subst; auto.
    congruence.
  - apply eval_interp_expr in H.
    break_match; inv H.
    break_match; subst; auto.
    congruence.
  - apply eval_interp_expr in H.
    break_match; inv H.
    break_match; subst; auto.
    congruence.
Qed.

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

Fixpoint run (fuel: nat) (h: heap) (s: stmt) : (heap * stmt) :=
  match fuel with
  | O => (h, s)
  | S n =>
      match interp_step h s with
      | Some (h', s') => run n h' s'
      | None => (h, s)
      end
  end.

Lemma run_stepn:
  forall fuel h s h' s',
  run fuel h s = (h', s') ->
  exists n, step_n h s n h' s'.
Proof.
  induction fuel; simpl; intros.
  - inversion H; subst.
    exists O. constructor.
  - break_match.
    destruct p.
    + apply IHfuel in H.
      apply interp_step_step in Heqo.
      destruct H. exists (S x).
      econstructor; eauto.
    + inv H. exists O.
      constructor; auto.
Qed.

Lemma stepn_run:
  forall h s n h' s',
  step_n h s n h' s' ->
  run n h s = (h', s').
Proof.
```



```
  intros. induction H; simpl; auto.
  break_match.
  + destruct p.
    apply step_interp_step in H.
    rewrite H in Heqo; inv Heqo.
    assumption.
  + apply step_interp_step in H.
    rewrite H in Heqo; inv Heqo.
Qed.
```