```
(** * IMP Semantics *)

Require Import ZArith.
Require Import String.

Open Scope string_scope.
Open Scope Z_scope.

Require Import IMPSyntax.

Definition heap : Type :=
  string -> Z.

Definition empty : heap :=
  fun v => 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 -> 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 e1 i1 ->
    eval h e2 i2 ->
    exec_op op i1 i2 = i3 ->
    eval h (BinOp op e1 e2) i3.

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.

Lemma interp_expr_eval:
  forall h e i,
  interp_expr h e = i ->
  eval h e i.
Proof.
  intros h e.
  induction e; simpl in *; intros.
  - subst; constructor.
  - subst; constructor.
  - apply eval_binop with (i1 := interp_expr h e1)
                          (i2 := interp_expr h e2).
    + apply IHe1. auto.
    + apply IHe2. auto.
    + assumption.
Qed.
```



```
Lemma eval_interp_expr:
  forall h e i,
  eval h e i ->
  interp_expr h e = i.
Proof.
  intros h e.
  induction e; simpl in *; intros.
  - inversion H; subst; auto.
  - inversion H; subst; reflexivity.
  - inversion H; subst.
    rewrite (IHe1 i1 H4).
    rewrite (IHe2 i2 H6).
    reflexivity.
Qed.

Lemma eval_interp:
  forall h e,
  eval h e (interp_expr h e).
Proof.
  intros. induction e; simpl.
  - constructor.
  - constructor.
  - econstructor.
    + eassumption.
    + eassumption.
    + reflexivity.
Qed.

Lemma eval_det:
  forall h e i1 i2,
  eval h e i1 ->
  eval h e i2 ->
  i1 = i2.
Proof.
  intros.
  apply eval_interp_expr in H.
  apply eval_interp_expr in H0.
  subst. reflexivity.
Qed.

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 (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 i ->
  i <> 0 ->
  step h (Cond e s) h s
| step_cond_false:
  forall h e s i,
  eval h e i ->
  i = 0 ->
```

```
  step h (Cond e s) h Nop
| step_while_true:
  forall h e s i,
  eval h e 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 i ->
  i = 0 ->
  step h (While e s) h Nop.

(**
| step_while:
  forall h e s,
  step h (While e s) h (Cond e (Seq s (While e s)))
*)

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 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.

Lemma interp_step_step:
  forall h s h' s',
  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_expr_eval; auto.
  - destruct (isNop s1) eqn:?.
    + rewrite isNop_ok in Heqb. subst.
      inversion H. subst. constructor.
    + destruct (interp_step h s1) as [[newHeap newStmt]|] eqn:?.
      * inversion H. subst.
        apply IHs1 in Heqo.
        constructor. assumption.
      * discriminate.
  - destruct (Z.eq_dec (interp_expr h e) 0) eqn:?.
    + inversion H. subst.
      clear Heqs0. apply interp_expr_eval in e0. econstructor.
      eassumption. auto.
    + inversion H; subst.
      eapply step_cond_true; eauto.
      apply interp_expr_eval; auto.
  - destruct (Z.eq_dec (interp_expr h e) 0) eqn:?.
    + inversion H; subst.
      eapply step_while_false; eauto.
      apply interp_expr_eval; auto.
    + inversion H; subst.
      eapply step_while_true; eauto.
      apply interp_expr_eval; auto.
Qed.

Lemma step_interp_step:
  forall h s h' s',
  step h s h' s' ->
  interp_step h s = Some (h', s').
Proof.
  intros. induction H; simpl; auto.
  - apply eval_interp_expr in H.
    subst; auto.
  - destruct (isNop s1) eqn:?.
    + apply isNop_ok in Heqb. subst.
      inversion H. (** nop can' step *)
    + rewrite IHstep. auto.
  - apply eval_interp_expr in H. subst.
    destruct (Z.eq_dec (interp_expr h e) 0); auto.
    exfalso. unfold not in H0. apply H0. assumption.
  - apply eval_interp_expr in H. subst.
    destruct (Z.eq_dec (interp_expr h e) 0); auto.
    unfold not in n. apply n in H0.
    exfalso. assumption.
  - apply eval_interp_expr in H. subst.
    destruct (Z.eq_dec (interp_expr h e) 0); auto.
    unfold not in H0. apply H0 in e0.
    inversion e0.
  - apply eval_interp_expr in H. subst.
    destruct (Z.eq_dec (interp_expr h e) 0); auto.
    omega.
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.
  - destruct (interp_step h s) as [[foo bar]|] eqn:?.
    + apply IHfuel in H.
      apply interp_step_step in Heqo.
      destruct H. exists (S x).
      econstructor; eauto.
    + inversion H; subst.
      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.
  destruct (interp_step h1 s1) as [[h' s']|] eqn:?.
  + apply step_interp_step in H.
    (** step is deterministic *)
    rewrite H in Heqo. inversion Heqo; subst.
    assumption.
  + apply step_interp_step in H.
    rewrite H in Heqo. discriminate.
Qed.
```