```
Require Import List.
Require Import ZArith.
Require Import String.

Open Scope string_scope.

Notation length := List.length.

Ltac inv H := inversion H; subst.

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.

(** syntax *)

Inductive expr : Set :=
| Bool   : bool -> expr
| Int    : Z -> expr
| Var    : string -> expr
| App    : expr -> expr -> expr
| Lam    : string -> expr -> expr
| Addr   : nat -> expr
| Ref    : expr -> expr
| Deref  : expr -> expr
| Assign : expr -> expr -> expr.

Coercion Bool : bool >-> expr.
Coercion Int  : Z >-> expr.
Coercion Var  : string >-> expr.

Notation "X @ Y"   := (App X Y)    (at level 49).
Notation "\ X , Y" := (Lam X Y)    (at level 50).
Notation "'ref' X" := (Ref X)      (at level 51).
Notation "! X"     := (Deref X)    (at level 51).
Notation "X <- Y"  := (Assign X Y) (at level 51).

(** Substitution. *)

(** e1[e2/x] = e3 *)
Inductive Subst : expr -> expr -> string ->
                  expr -> Prop :=
| SubstBool:
    forall b e x,
      Subst (Bool b) e x
            (Bool b)
| SubstInt:
    forall i e x,
      Subst (Int i) e x
            (Int i)
| SubstVar_same:
    forall e x,
      Subst (Var x) e x
            e
| SubstVar_diff:
    forall e x1 x2,
      x1 <> x2 ->
      Subst (Var x1) e x2
            (Var x1)
| SubstApp:
    forall e1 e2 e x e1' e2',
```

```
      Subst e1 e x e1' ->
      Subst e2 e x e2' ->
      Subst (App e1 e2) e x
            (App e1' e2')
| SubstLam_same:
    forall e1 x e,
      Subst (Lam x e1) e x
            (Lam x e1)
| SubstLam_diff:
    forall e1 x1 x2 e e1',
      x1 <> x2 ->
      Subst e1 e x2 e1' ->
      Subst (Lam x1 e1) e x2
            (Lam x1 e1')
| SubstAddr :
    forall a e x,
      Subst (Addr a) e x
            (Addr a)
| SubstRef :
    forall r e x r',
      Subst r e x r' ->
      Subst (Ref r) e x (Ref r')
| SubstDeref :
    forall r e x r',
      Subst r e x r' ->
      Subst (Deref r) e x (Deref r')
| SubstAssign:
    forall e1 e2 e x e1' e2',
      Subst e1 e x e1' ->
      Subst e2 e x e2' ->
      Subst (Assign e1 e2) e x
            (Assign e1' e2').

Inductive free : expr -> string -> Prop :=
| FreeVar:
    forall x,
      free (Var x) x
| FreeApp_l:
    forall x e1 e2,
      free e1 x ->
      free (App e1 e2) x
| FreeApp_r:
    forall x e1 e2,
      free e2 x ->
      free (App e1 e2) x
| FreeLam:
    forall x1 x2 e,
      free e x1 ->
      x1 <> x2 ->
      free (Lam x2 e) x1
| FreeRef :
    forall x r,
      free r x ->
      free (Ref r) x
| FreeDeref :
    forall x r,
      free r x ->
      free (Deref r) x
| FreeAssign_l:
    forall x e1 e2,
      free e1 x ->
      free (Assign e1 e2) x
| FreeAssign_r:
    forall x e1 e2,
      free e2 x ->
      free (Assign e1 e2) x.

Lemma subst_only_free:
  forall e1 e2 x e3,
    Subst e1 e2 x e3 ->
    ~ free e1 x ->
    e1 = e3.
Proof.
  induction 1; intros; auto.
  - destruct H. constructor.
  - f_equal.
    + apply IHSubst1; intuition.
      apply H1; apply FreeApp_l; auto.
    + apply IHSubst2; intuition.
      apply H1; apply FreeApp_r; auto.
  - rewrite IHSubst; auto.
    intuition. apply H1.
    constructor; auto.
  - rewrite IHSubst; auto.
    intuition. apply H0.
    constructor; auto.
  - rewrite IHSubst; auto.
    intuition. apply H0.
    constructor; auto.
  - f_equal.
    + apply IHSubst1; intuition.
      apply H1; apply FreeAssign_l; auto.
    + apply IHSubst2; intuition.
      apply H1; apply FreeAssign_r; auto.
Qed.

(** Closed terms have no free variables *)
Definition closed (e: expr) : Prop :=
  forall x, ~ free e x.

(** These are a bunch of boring facts about closed terms.
   We've completed the proofs, but look over them because
   they are will be useful later.
 *)
Lemma closed_app_intro:
  forall e1 e2,
    closed e1 ->
    closed e2 ->
    closed (e1 @ e2).
Proof.
  unfold closed, not; intros.
  inv H1.
  - eapply H; eauto.
  - eapply H0; eauto.
Qed.

Lemma closed_app_inv:
  forall e1 e2,
    closed (e1 @ e2) ->
    closed e1 /\ closed e2.
Proof.
  unfold closed, not; split; intros.
  - eapply H; eauto.
    apply FreeApp_l; eauto.
  - eapply H; eauto.
    apply FreeApp_r; eauto.
Qed.

Lemma closed_lam_intro:
  forall x e,
    (forall y, y <> x -> ~ free e y) ->
    closed (\x, e).
Proof.
  unfold closed, not; intros.
  inv H0. eapply H; eauto.
Qed.

Lemma closed_lam_inv:
```

```
  forall x e,
    closed (\x, e) ->
    (forall y, y <> x -> ~ free e y).
Proof.
  unfold closed, not; intros.
  cut (free (\x, e) y); intros.
  - eapply H; eauto.
  - constructor; auto.
Qed.

Lemma closed_ref_intro:
  forall e,
    closed e ->
    closed (ref e).
Proof.
  unfold closed, not; intros.
  inv H0. eauto.
Qed.

Lemma closed_ref_inv:
  forall e,
    closed (ref e) ->
    closed e.
Proof.
  unfold closed, not; intros.
  eapply H.
  constructor. eauto.
Qed.

Lemma closed_deref_intro:
  forall e,
    closed e ->
    closed (! e).
Proof.
  unfold closed, not; intros.
  inv H0. eauto.
Qed.

Lemma closed_deref_inv:
  forall e,
    closed (! e) ->
    closed e.
Proof.
  unfold closed, not; intros.
  eapply H.
  constructor. eauto.
Qed.

Lemma closed_assign_intro:
  forall e1 e2,
    closed e1 ->
    closed e2 ->
    closed (e1 <- e2).
Proof.
  unfold closed, not; intros.
  inv H1.
  - eapply H; eauto.
  - eapply H0; eauto.
Qed.

Lemma closed_assign_inv:
  forall e1 e2,
    closed (e1 <- e2) ->
    closed e1 /\ closed e2.
Proof.
  unfold closed, not; split; intros.
  - eapply H; eauto.
    apply FreeAssign_l; eauto.
  - eapply H; eauto.
```

```
    apply FreeAssign_r; eauto.
Qed.

(*

Here we define "heaps", which our references will index into.

A heap is just a list of expressions (in our uses later below,
they will always be values) which can be indexed into.

*)

Definition heap := list expr.

(*

We define lookup in terms of the "nth" function:
  https://coq.inria.fr/library/Coq.Lists.List.html

nth takes a default argument (consider why!), but
we will not actually end up in the default case
in our code later below.

*)

Definition lookup (h : heap) n :=
  nth n h true.

(** snoc is cons, backwards. It adds an element to the end of a list. *)
Fixpoint snoc {A:Type} (l:list A) (x:A) : list A :=
  match l with
    | nil => x :: nil
    | h :: t => h :: snoc t x
  end.

(** We will need some boring lemmas about [snoc]. We’ve completed the
proofs for you, but look over them since you’ll need them later.  *)

Lemma length_snoc : forall A (l:list A) n,
  length (snoc l n) = S (length l).
Proof.
  induction l; intros; auto.
  simpl. rewrite IHl. auto.
Qed.

Lemma nth_lt_snoc : forall A (l:list A) x d n,
  n < length l ->
  nth n l d = nth n (snoc l x) d.
Proof.
  induction l; intros; simpl in *.
  - omega.
  - break_match; auto.
    apply IHl. omega.
Qed.

Lemma nth_eq_snoc : forall A (l:list A) x d,
  nth (length l) (snoc l x) d = x.
Proof.
  induction l; intros; auto.
  simpl. rewrite IHl. auto.
Qed.

(** To update the heap, we use the [replace] function, which replaces
    the contents of a cell at a particular index. *)

Fixpoint replace {A:Type} (n:nat) (x:A) (l:list A) : list A :=
  match l with
  | nil    => nil
  | h :: t =>
```

```
    match n with
    | O    => x :: t
    | S n’ => h :: replace n’ x t
    end
  end.

(** Of course, we also need some boring lemmas about [replace], which
    are also fairly straightforward to prove. *)

Lemma replace_nil : forall A n (x:A),
  replace n x nil = nil.
Proof.
  destruct n; auto.
Qed.

Lemma length_replace : forall A (l:list A) n x,
  length (replace n x l) = length l.
Proof.
  induction l; intros; simpl;
  destruct n; simpl; eauto.
Qed.

Lemma lookup_replace_eq : forall h a t,
  a < length h ->
  lookup (replace a t h) a = t.
Proof.
  unfold lookup.
  induction h; intros; simpl in *; auto.
  - omega.
  - destruct a0; simpl; auto.
    apply IHh. omega.
Qed.

(*

Now that we have heaps, let’s define our semantics!

Since we’re writing a call-by-value semantics,
we first need to define "values".

*)
Inductive isValue : expr -> Prop :=
| VLam  : forall x e, isValue (\ x, e)
| VInt  : forall i : Z, isValue i
| VBool : forall b : bool, isValue b
| VAddr : forall n, isValue (Addr n).

(*

Our step relation includes heaps as well as expressions, since heaps
can change. Look carefully over this step relation and make sure you
understand every rule!  Really, you’ll need to grok this to finish the
homework.

*)
Inductive step_cbv : heap -> expr -> heap -> expr -> Prop :=
| SAppLeft :
    forall h h’ e1 e1’ e2,
      step_cbv h e1
               h’ e1’ ->
      step_cbv h (e1 @ e2)
               h’ (e1’ @ e2)
| SAppRight :
    forall h h’ e1 e2 e2’,
      isValue e1 ->
      step_cbv h e2
               h’ e2’ ->
      step_cbv h (e1 @ e2)
               h’ (e1 @ e2’)
```

```
| SApp :
    forall h x e1 e2 e1',
      isValue e2 ->
      Subst e1 e2 x e1' ->
      step_cbv h ((\ x, e1) @ e2)
               h e1'
| SRef :
    forall h h' e e',
      step_cbv h e
               h' e' ->
      step_cbv h (ref e)
               h' (ref e')
| SRefValue :
    forall h e,
      isValue e ->
      step_cbv h (ref e)
               (snoc h e) (Addr (length h))
| SDeref :
    forall h h' e e',
      step_cbv h e
               h' e' ->
      step_cbv h (! e)
               h' (! e')
| SDerefAddr :
    forall h a,
      a < length h ->
      step_cbv h (! (Addr a))
               h (lookup h a)
| SAssignLeft :
    forall h h' e1 e1' e2,
      step_cbv h e1
               h' e1' ->
      step_cbv h (e1 <- e2)
               h' (e1' <- e2)
| SAssignRight :
    forall h h' e1 e2 e2',
      isValue e1 ->
      step_cbv h e2
               h' e2' ->
      step_cbv h (e1 <- e2)
               h' (e1 <- e2')
| SAssign :
    forall h a e,
      isValue e ->
      a < length h ->
      step_cbv h (Addr a <- e)
               (replace a e h) (Bool true).

Notation "h1 ; e1 '==>' h2 ; e2" :=
  (step_cbv h1 e1 h2 e2) (at level 40, e1 at level 39, h2 at level 39).

(** any number of steps *)
Inductive star_cbv : heap -> expr -> heap -> expr -> Prop :=
| scbn_refl:
    forall h e,
      star_cbv h e h e
| scbn_step:
    forall h1 e1 h2 e2 h3 e3,
      step_cbv h1 e1 h2 e2 ->
      star_cbv h2 e2 h3 e3 ->
      star_cbv h1 e1 h3 e3.

Notation "h1 ; e1 ==>* h2 ; e2" :=
  (star_cbv h1 e1 h2 e2) (at level 40, e1 at level 39, h2 at level 39).

(** Let's talk about types! *)

(** We'll need to add a type for references to the set of types we saw lecture.
*)
```

```
Inductive typ : Set :=
| TBool : typ
| TInt  : typ
| TFun  : typ -> typ -> typ
| TRef  : typ -> typ.

Notation "X ~> Y" := (TFun X Y) (at level 60).

(** An environment maps variables to types. Make sure you understand
the difference between this and a heap, which maps indices to
terms! *)

Definition env : Type :=
  string -> option typ.

(** E0 is the empty environment *)
Definition E0 : env :=
  fun _ => None.

(*

Update an environment with a new variable and type.

NOTE: Environments are different from heaps!
      We change heaps with snoc and replace.
      We change environments with extend.

*)
Definition extend (e: env) x t : env :=
  fun y =>
    if string_dec y x then
      Some t
    else
      e y.

(** In addition to our usual environments,
   we also need to type our heaps in order
   to type references. *)

(** A heap type is just a list of types. *)
Definition heap_typ :=
  list typ.

(** lookup_typ works just like lookup. *)
Definition lookup_typ (h : heap_typ) n :=
  nth n h TBool.

(** What does it mean for a term to be well-typed?
   The first 5 constructors are the same as those
   in the STLC without references.
*)

Inductive typed : env -> heap_typ -> expr -> typ -> Prop :=
| WTBool :
    forall env ht b,
      typed env ht (Bool b) TBool
| WTInt :
    forall env ht i,
      typed env ht (Int i) TInt
| WTVar :
    forall env ht x t,
      env x = Some t ->
      typed env ht (Var x) t
| WTApp :
    forall env ht e1 e2 tA tB,
      typed env ht e1 (tA ~> tB) ->
      typed env ht e2 tA ->
      typed env ht (e1 @ e2) tB
```

```
| WTLam :
    forall env ht x e tA tB,
      typed (extend env x tA) ht e tB ->
      typed env ht (\x, e) (tA ~> tB)
| WTAddr :
    forall env ht a,
      a < length ht ->
      typed env ht (Addr a) (TRef (lookup_typ ht a))
| WTRef :
    forall env ht e t,
      typed env ht e t ->
      typed env ht (ref e) (TRef t)
| WTDeref :
    forall env ht e t,
      typed env ht e (TRef t) ->
      typed env ht (! e) t
| WTAssign :
    forall env ht e1 e2 t,
      typed env ht e1 (TRef t) ->
      typed env ht e2 t ->
      typed env ht (e1 <- e2) TBool.

(** Q: What does it mean for a heap to be well-typed?

   A: The heap must be the same length as the heap type, and the term
      stored at any valid address in the heap (i.e. any address less than
      the length of the heap) should have the type it has in the heap type.
 *)

Definition heap_typed (ht : heap_typ) (h : heap) :=
  length h = length ht /\
  forall a,
    a < length h ->
    typed E0 ht (lookup h a) (lookup_typ ht a).
```