```
Require Import List.
Require Import String.
Open Scope string_scope.

Inductive expr : Set :=
| Var : string -> expr
| App : expr -> expr -> expr
| Lam : string -> expr -> expr.

Coercion Var : string >-> expr.

Notation "X @ Y" := (App X Y) (at level 49).
Notation "\ X , Y" := (Lam X Y) (at level 50).

Check (\"x", \"y", "x").
Check (\"x", \"y", "y").
Check ((\"x", "x" @ "x") @ (\"x", "x" @ "x")).

(** e1[e2/x] = e3 *)
Inductive Subst : expr -> expr -> string ->
                  expr -> Prop :=
| 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').

Lemma subst_test_1:
  Subst (\"x", "y") "z" "y"
        (\"x", "z").
Proof.
  apply SubstLam_diff.
  - discriminate.
  - apply SubstVar_same.
Qed.

Lemma subst_test_2:
  Subst (\"x", "x") "z" "x"
        (\"x", "x").
Proof.
  apply SubstLam_same.
Qed.

(**
Call By Name
<<
       e1 --> e1'
  ---------------------
    e1 e2 --> e1' e2

  -----------------------------
    (\x. e1) e2 --> e1[e2/x]
>>
*)
```



```
Inductive step_cbn : expr -> expr -> Prop :=
| CBN_crunch:
    forall e1 e1' e2,
      step_cbn e1 e1' ->
      step_cbn (App e1 e2) (App e1' e2)
| CBN_subst:
    forall x e1 e2 e1',
      Subst e1 e2 x e1' ->
      step_cbn (App (Lam x e1) e2) e1'.

Notation "e1 ==> e2" := (step_cbn e1 e2) (at level 51).

Lemma sstep_test_1:
  (\"x", "x") @ "z" ==> "z".
Proof.
  apply CBN_subst.
  apply SubstVar_same.
Qed.

Lemma Lam_nostep_cbn:
  forall x e1 e2,
  ~ (\x, e1 ==> e2).
Proof.
  intros. intro. inversion H.
Qed.

Ltac inv H := inversion H; subst.

(** careful to make IH sufficiently strong *)
Lemma Subst_det:
  forall e1 e2 x e3,
    Subst e1 e2 x e3 ->
    forall e3',
      Subst e1 e2 x e3' ->
      e3 = e3'.
Proof.
  induction 1; intros.
  - inv H; auto. congruence.
  - inv H0; auto. congruence.
  - inv H1.
    apply IHSubst1 in H4.
    apply IHSubst2 in H8.
    subst; auto.
  - inv H; auto. congruence.
  - inv H1; auto. congruence.
    apply IHSubst in H8; subst; auto.
Qed.

Lemma step_cbn_det:
  forall e e1,
  e ==> e1 ->
  forall e2,
  e ==> e2 ->
  e1 = e2.
Proof.
  induction 1; intros.
  - inv H0.
    + f_equal. apply IHstep_cbn; auto.
    + exfalso. apply Lam_nostep_cbn in H; auto.
  - inv H0.
    + exfalso. apply Lam_nostep_cbn in H4; auto.
    + eapply Subst_det; eauto.
Qed.

(**
Call By Value
<<
```

```
v ::= \ x . e

       e1 --> e1'
  ---------------------
    e1 e2 --> e1' e2

       e2 --> e2'
  ---------------------
    v e2 --> v e2'

  -----------------------------
    (\x. e1) v --> e1[v/x]
>>
*)

Inductive value : expr -> Prop :=
| VLam :
    forall x e,
      value (Lam x e).

Inductive step_cbv : expr -> expr -> Prop :=
| CBV_crunch_l:
    forall e1 e1' e2,
      step_cbv e1 e1' ->
      step_cbv (App e1 e2) (App e1' e2)
| CBV_crunch_r:
    forall v e2 e2',
      value v ->
      step_cbv e2 e2' ->
      step_cbv (App v e2) (App v e2')
| CBV_subst:
    forall x e1 v e1',
      value v ->
      Subst e1 v x e1' ->
      step_cbv (App (Lam x e1) v) e1'.

Notation "e1 --> e2" := (step_cbv e1 e2) (at level 51).

Inductive star (step: expr -> expr -> Prop) :
  expr -> expr -> Prop :=
| star_refl:
    forall s,
      star step s s
| star_step:
    forall s1 s2 s3,
      step s1 s2 ->
      star step s2 s3 ->
      star step s1 s3.

Notation "e1 ==>* e2" := (star step_cbn e1 e2) (at level 51).
Notation "e1 -->* e2" := (star step_cbv e1 e2) (at level 51).

Ltac zex x := exists x.

Lemma can_subst:
  forall e1 e2 x,
  exists e3, Subst e1 e2 x e3.
Proof.
  intros. induction e1.
  - case (string_dec s x); intros; subst.
    + zex e2; constructor.
    + zex (Var s); constructor; auto.
  - destruct IHe1_1; destruct IHe1_2.
    exists (x0 @ x1); constructor; auto.
  - case (string_dec s x); intros; subst.
    + zex (\ x, e1); constructor; auto.
    + destruct IHe1.
      exists (\ s, x0); constructor; auto.
Qed.
```



```
Lemma cbv_cbn_can_step:
  forall e1 e2,
    e1 --> e2 ->
    exists e3, e1 ==> e3.
Proof.
  induction 1.
  - destruct IHstep_cbv as [e3 He3].
    exists (e3 @ e2). constructor; auto.
  - inv H. destruct (can_subst e e2 x).
    exists x0; constructor; auto.
  - exists e1'; constructor; auto.
Qed.

(** is the other way true? *)

(** * Church Encodings *)

(** generally assume no free vars! *)

Definition lcTrue :=
  \"x", \"y", "x".

Definition lcFalse :=
  \"x", \"y", "y".

Definition lcCond (c t f: expr) :=
  c @ t @ f.

(** <<

lcCond lcTrue e1 e2 -->* e1

>> *)

Definition lcNot :=
  \"b", "b" @ lcFalse @ lcTrue.

Definition lcAnd :=
  \"a", \"b", "a" @ "b" @ lcFalse.

Definition lcOr :=
  \"a", \"b", "a" @ lcTrue @ "b".

Definition lcMkPair :=
  \"x", \"y",
    (\"s", "s" @ "x" @ "y").

Definition lcFst :=
  \"p", "p" @ (\"x", \"y", "x").

Definition lcSnd :=
  \"p", "p" @ (\"x", \"y", "y").

(** <<

lcSnd (lcFst (lcMkPair (lcMkPair e1 e2) e3)) -->* e2

lcFst = \"p", "p" @ lcTrue

lcSnd = \"p", "p" @ lcFalse

>> *)

Definition lcNil :=
  lcMkPair @ lcFalse @ lcFalse.

Definition lcCons :=
  \"h", \"t", lcMkPair @ lcTrue @ (lcMkPair @ "h" @ "t").
```

```
Definition lcIsEmpty :=
  lcFst.

Definition lcHead :=
  \"l", lcFst @ (lcSnd @ "l").

Definition lcTail :=
  \"l", lcSnd @ (lcSnd @ "l").

(** <<

Note that lcTail lcNil does some weird stuff,
but then so does dereferencing null in C or
following null.next() in Java.

>> *)

Definition lc0 :=
  \"s", \"z", "z".

Definition lc1 :=
  \"s", \"z", "s" @ "z".

Definition lc2 :=
  \"s", \"z", "s" @ ("s" @ "z").

Definition lc3 :=
  \"s", \"z", "s" @ ("s" @ ("s" @ "z")).

Definition lc4 :=
  \"s", \"z", "s" @ ("s" @ ("s" @ ("s" @ "z"))).

(** <<

Number "n" composes first arg with itself n times,
starting with the second arg.

>> *)

Definition lcSucc :=
  \"n", \"s", \"z", "s" @ ("n" @ "s" @ "z").

Definition lcAdd :=
  \"n", \"m",
    (\"s", \"z", "n" @ lcSucc @ "m").

Definition lcMul :=
  \"n", \"m",
    (\"s", \"z", "n" @ (lcAdd @ "m") @ lc0).

Definition lcIsZero :=
  \"n",
    "n" @ (\"x", lcFalse) @ lcTrue.

(** <<
  Can keep going to get pred, minus, div, is_equal, ...
>> *)

Definition lcPred :=
  (** TODO : define pred on Church numerals *)
  "x".

(** only works for CBN! *)
Definition lcY :=
  \"f",
    (\"x", "f" @ ("x" @ "x")) @
    (\"x", "f" @ ("x" @ "x")).
```



```
(** <<

Y F
  -->* (\f, (\x, f (x x)) (\x, f (x x))) F
  -->* (\x, F (x x)) (\x, F (x x))
  -->* F ((\x, F (x x)) (\x, F (x x)))
  -->* F (Y F)

>> *)

Definition lcFactAux :=
  \"f", \"n",
  lcCond (lcIsZero @ "n")
         lc1
         (lcMul @ "n" @ ("f" @ (lcPred @ "n"))).

Definition lcFact :=
  lcY @ lcFactAux.

(** <<

lcFact 3 -->* 6

>> *)

Definition lcLet v e1 e2 :=
  (\v, e2) @ e1.
```