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").

(* 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 (Lam x1 e) |
| SubstApp: forall el e2 x el' e2', Subst el e x el' -> Subst e2 e x e2' -> Subst (App el e2) e x (App el' e2') |
| SubstLam_same: forall el e, Subst (Lam x el) e x (Lam x el) |
| SubstLam_diff: forall el x1 x2 el', x1 <> x2 -> Subst el x x el' -> Subst (Lam x el) e x2 (Lam x el') |

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

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

(** Call By Name <<
"e1" -> "el"
-----------------
"el" -> "el'"
-----------------
"el" e2 -> "el'" e2
-----------------
"el" e2 x e2 -> "el" e2/x
>>
**)
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" @ ("s", "y", "x")).

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

(* <<
  lcSnd (lcFst (lcMkPair lcMkPair e1 e2 e3)) -> e2
  lcFst = "p", "p" @ lcTrue
  lcSnd = "p", "p" @ lcFalse
  >> *)

Definition lcNil := lcMkPair @ lcFalse @ lcTrue.

Definition lcCons := "h", "h", 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", " s" @ 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, (\x, " f" @ (" x" @ " x")) @
   (\x, " f" @ (" x" @ " x"))) @
   (\x, (\x, " f" @ (" x" @ " x")) @
    (\x, " f" @ (" x" @ " x"))).

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

Definition lcFact :=
  lcY @ lcFactAux.

(*<< lcFact 3 -->* 6 >>*)

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