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

Require Import StructTactics.

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

Definition identity_function := Lam "x" (Var "x").

Definition ap_id_id := App (Lam "x" (Var "x")) (Lam "x" (Var "x")).

Coercion Var : string >-> expr.
Coercion App : expr >-> Funclass.
Notation "\ X , Y" := (Lam X Y) (at level 50).

Definition better_id : expr := \"x", "x".

Definition better_ap_id := (\"x", "x") (\"x", "x").

(** Set Printing All. *)

Print better_ap_id.

(** e[to/from] *)
Definition subst (from : string) (to : expr) (e : expr) : expr.
Admitted.

(** Notation "e [ to / from ]" := (subst from to e) (at level 42). *)

Axiom subst_var : forall from to, subst from to from = to.

(***
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 x e2 e1 = e1' ->
      step_cbn (App (Lam x e1) e2) e1'.
Notation "e1 -cbn-> e2" := (step_cbn e1 e2) (at level 51).

Lemma sstep_test_1:
  (\"x", "x") "z" -cbn-> "z".
Proof.
  apply CBN_subst.
  apply subst_var.
Qed.

Lemma Lam_nostep_cbn:
  forall x e1 e2,
  ~ (\x, e1 -cbn-> e2).
```



```
Proof.
  intros. intro. inversion H.
Qed.

Lemma step_cbn_det:
  forall e e1,
  e -cbn-> e1 ->
  forall e2,
  e -cbn-> e2 ->
  e1 = e2.
Proof.
  induction 1; intros.
  - invc 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.
    + auto.
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 x v e1 = e1' ->
      step_cbv (App (Lam x e1) v) e1'.

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

Lemma cbv_cbn_can_step:
  forall e1 e2,
    e1 -cbv-> e2 ->
    exists e3, e1 -cbn-> e3.
Proof.
  induction 1.
  - destruct IHstep_cbv as [e3 He3].
    exists (e3 e2). constructor; auto.
  - inv H.
    eexists.
    eapply CBN_subst; eauto.
  - 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

  -->* Y lcFactAux 3

  -->* lcFactAux (Y lcFactAux) 3

  -->* (\f, \n, if (n = 0) then 1 else (n * f (n - 1))) (Y lcFactAux) 3

  -->* (\n if (n = 0) then 1 else (n * (Y lcFactAux (n - 1)))) 3

  -->* if (3 = 0) then 1 else (3 * (Y lcFactAux (3 - 1)))

  -->* 3 * (Y lcFactAux (3 - 1))

  -->* 3 * (Y lcFactAux 2)

  -->* 3 * (Y (\f, \n, if (n = 0) then 1 else (n * f (n - 1))) 2)

  -->* 3 * ((\n if (n = 0) then 1 else (n * (Y lcFactAux (n - 1)))) 2)

  -->* 3 * (if (2 = 0) then 1 else (2 * (Y lcFactAux (2 - 1))))

  -->* 3 * (2 * (Y lcFactAux (2 - 1)))

  -->* 3 * (2 * (Y lcFactAux 1))

  -->* 3 * (2 * (Y (\f, \n, if (n = 0) then 1 else (n * f (n - 1))) 1))

  -->* 3 * (2 * ((\n if (n = 0) then 1 else (n * (Y lcFactAux (n - 1)))) 1))

  -->* 3 * (2 * (if (1 = 0) then 1 else (1 * (Y lcFactAux (1 - 1)))))

  -->* 3 * (2 * (1 * (Y lcFactAux (1 - 1))))

  -->* 3 * (2 * (1 * (Y lcFactAux 0)))

  -->* 3 * (2 * (1 * (Y (\f, \n, if (n = 0) then 1 else (n * f (n - 1))) 0)))
```



```
  -->* 3 * (2 * (1 * ((\n if (n = 0) then 1 else (n * (Y lcFactAux (n - 1)))) 0)
)

  -->* 3 * (2 * (1 * (if (0 = 0) then 1 else (0 * (Y lcFactAux (0 - 1))))

  -->* 3 * (2 * (1 * 1))

  -->* 6

>> *)

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

Module attempt1.

Fixpoint subst (from : string) (to : expr) (e : expr) : expr :=
  match e with
  | Var x =>
    if string_dec from x
    then to
    else Var x
  | App e1 e2 =>
    App (subst from to e1) (subst from to e2)
  | Lam x e =>
    Lam x (subst from to e)
  end.

Eval compute in subst "x" (\"y", "y") "x".

Eval compute in fun e => subst "x" e "x". (** hehe, nice! *)

Eval compute in subst "y" (\"z", "z") (\"x", "x" "y").

Eval compute in subst "x" (\"x", "x" "x") ("x" "x").

Eval compute in subst "x" (\"z", "z") (\"x", "x").
(**   = \ "x", (\ "z", "z")
but should be \x. x :( *)
End attempt1.

Module attempt2.

Fixpoint subst (from : string) (to : expr) (e : expr) : expr :=
  match e with
  | Var x =>
    if string_dec from x
    then to
    else Var x
  | App e1 e2 =>
    App (subst from to e1) (subst from to e2)
  | Lam x e =>
    if string_dec from x
    then Lam x e
    else Lam x (subst from to e)
  end.

Eval compute in subst "x" (\"y", "y") "x".

Eval compute in fun e => subst "x" e "x".
(** hehe, nice! *)

Eval compute in subst "y" (\"z", "z") (\"x", "x" "y").

Eval compute in subst "x" (\"x", "x" "x") ("x" "x").

Eval compute in subst "x" (\"z", "z") (\"x", "x").
(** works! *)
```

```
Eval compute in subst "x" "y" (\"y", "x").
(**   = \ "y", "y"
feels wrong; doesn't respect renaming bound variables *)

End attempt2.

Fixpoint free_in (x : string) (e : expr) : bool :=
  match e with
  | Var y => if string_dec x y then true else false
  | Lam y e => if string_dec x y then false else free_in x e
  | App e1 e2 => free_in x e1 || free_in x e2
  end.

Eval compute in free_in "x" "x".
Eval compute in free_in "x" "y".
Eval compute in free_in "x" (\"y", "x").
Eval compute in free_in "x" (\"x", "x").


Module attempt3.

Fixpoint subst (from : string) (to : expr) (e : expr) : option expr :=
  match e with
  | Var x =>
    if string_dec from x
    then Some to
    else Some (Var x)
  | App e1 e2 =>
    match subst from to e1 with None => None
    | Some e1' =>
    match subst from to e2 with None => None
    | Some e2' =>
      Some (App e1' e2')
    end
    end
  | Lam x e =>
    if string_dec from x
    then Some (Lam x e)
    else if free_in x to
         then None
         else match subst from to e with None => None
              | Some e' => Some (Lam x e')
              end
  end.

Eval compute in subst "x" (\"y", "y") "x".

Eval compute in fun e => subst "x" e "x".
(** hehe, nice! *)

Eval compute in subst "y" (\"z", "z") (\"x", "x" "y").

Eval compute in subst "x" (\"x", "x" "x") ("x" "x").

Eval compute in subst "x" (\"z", "z") (\"x", "x").
(** works! *)

Eval compute in subst "x" "y" (\"y", "x").
(**   = None
okay, at least we detect when we're wrong!
kind of a bummer since substitution can get stuck now.
worse, whether it gets stuck *depends on choice of bound variable names* :O
*)

End attempt3.

(** ignore this stuff and skip to attempt4 *)
```



```
Fixpoint string_ltb (s1 s2 : string) : bool :=
  match s1 with
  | EmptyString =>
    match s2 with
    | EmptyString => false
    | _ => true
    end
  | String a1 s1 =>
    match s2 with
    | EmptyString => false
    | String a2 s2 =>
      let n1 := Ascii.nat_of_ascii a1 in
      let n2 := Ascii.nat_of_ascii a2 in
      (Nat.ltb n1 n2)
      || (Nat.eqb n1 n2 && string_ltb s1 s2)
    end
  end.

Definition max_string (s1 s2 : string) : string :=
  if string_ltb s1 s2 then s2 else s1.
Eval compute in max_string "x" "y".
Eval compute in max_string "x" "".

Fixpoint maximum_string (l : list string) : string :=
  match l with
  | [] => "x" (** shrug *)
  | s :: l => max_string s (maximum_string l)
  end.

Definition succ_string (s : string) : string :=
  append s "'".
Eval compute in succ_string "x".

Definition fresh (l : list string) : string :=
  succ_string (maximum_string l).

Fixpoint allVars (e : expr) : list string :=
  match e with
  | Var x => [x]
  | App e1 e2 => nodup string_dec (allVars e1 ++ allVars e2)
  | Lam x e => nodup string_dec (x :: allVars e)
  end.

Definition freshExpr (e : expr) : string :=
  fresh (allVars e).

Module attempt4.

Fail Fixpoint subst (from : string) (to : expr) (e : expr) : expr :=
  match e with
  | Var x =>
    if string_dec from x
    then to
    else (Var x)
  | App e1 e2 =>
    App (subst from to e1) (subst from to e2)
  | Lam x e =>
    if string_dec from x
    then Lam x e
    else
      (** tricky way of getting something fresh for e and to,
          get something fresh for e applied to to :) *)
      let z := freshExpr (e to) in
      let e' := subst x z e in
      Lam z (subst from to e')
  end.
(** doesn't work because not structurally recursive :( :( :( *)

End attempt4.
```

```
Definition freshEnv (env : list (string * expr)) : string :=
  fresh (nodup string_dec
              (flat_map (fun p : string * expr => let (s, e) := p in s :: allVa
rs e) env)).

Fixpoint lookup (l : list (string * expr)) (x : string) : option expr :=
  match l with
  | [] => None
  | (y, e) :: l => if string_dec x y then Some e else lookup l x
  end.

Module attempt5.

Fixpoint multisubst (env : list (string * expr)) (e : expr) : expr :=
  match e with
  | Var x =>
    match lookup env x with
    | None => Var x
    | Some e => e
    end
  | App e1 e2 =>
    App (multisubst env e1) (multisubst env e2)
  | Lam x e =>
    let z := freshEnv ((x, e) :: env) in
    Lam z (multisubst ((x, Var z) :: env) e)
  end.

Definition subst (from : string) (to : expr) (e : expr) : expr :=
  multisubst [(from, to)] e.

Eval compute in subst "x" (\"y", "y") "x".

Eval compute in fun e => subst "x" e "x".
(** hehe, nice! *)

Eval compute in subst "y" (\"z", "z") (\"x", "x" "y").
(** z' is confusing, but it's right *)

Eval compute in subst "x" (\"x", "x" "x") ("x" "x").

Eval compute in subst "x" (\"z", "z") (\"x", "x").
(** works! *)

Eval compute in subst "x" "y" (\"y", "x").

(** This is pretty good! About the best you can do in Coq with explicit
   names. Just remember that substitution is only defined up to
   alpha-equivalence. The implementation is free to rename bound vars
   whenever it wants, and we just have to deal with it. *)

(** To be honest, I actually have no idea whether the above code is
   correct. We could increase our confidence by proving some nice
   facts about substitution. Even better, we could prove it equivalent
   to de Bruijn. *)

End attempt5.
```