Part I

Boxes vs. Variables

- State via box:

```
let x = box(1):
  begin:
    set_box(x, 2)
    unbox(x)
```

- State via variable:

```
let x = 1:
  begin:
    x := 2
    x
```

Boxes vs. Variables

```
let x = 5:
  let f = (fun (y):
             x + y):
    begin:
      x := 6
      f(1)
```

Boxes vs. Variables

```
let x = 5:
  let f = (fun (y):
             x + y):
    begin:
      x := 6
      f(1)
```

```
let x = box(5):
  let f = (fun (y):
             unbox(x) + y):
    begin:
      set_box(x, 6)
      f(1)
```

Boxes vs. Variables

```
let x = 5:
  let f = (fun (y):
             x + y):
    begin:
      x := 6
      f(1)
```

```
let x = box(5):
  let f = (fun (y):
             unbox(x) + y):
    begin:
      set_box(x, 6)
      f(1)
```

Boxes vs. Variables

```
let x = 5:
  let f = (fun (y):
             x + y):
    begin:
      x := 6
      f(1)
```

```
let x = box(5):
  let f = box((fun (y):
                 unbox(x) + y)):
    begin:
      set_box(x, 6)
      (unbox(f))(1)
```

Boxes vs. Variables

```
let x = 5:
  let f = (fun (y):
             x + y):
    begin:
      x := 6
      f(1)
```

```
let x = box(5):
  let f = box((fun (y):
                 unbox(x) + unbox(y))):
    begin:
      set_box(x, 6)
      (unbox(f))(box(1))
```

Boxes vs. Variables

```
let x = 5:
  let f = (fun (y):
             x + y):
    begin:
      x := 6
      f(1)
```

```
type Binding
| bind(name :: Symbol,
        location :: Location)
```

```
let x = box(5):
  let f = box((fun (y):
                 unbox(x) + unbox(y))):
    begin:
      set_box(x, 6)
      (unbox(f))(box(1))
```

Part 2

Variables

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> - <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  <Exp> := <Exp>
         |  begin: <Exp>; <Exp>
```

NEW

```
let b = 0:
  begin:
    b := 10
    b
```

⇒ 10

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(intE(5), mt_env, mt_store)
       ~is res(intV(5), mt_store)
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('let x = 5: x'),
              mt_env,
              mt_store)
       ~is res(....,
               ....)
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('let x = 5: x'),
              mt_env,
              mt_store)
       ~is res(intV(5),
               ....)
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('let x = 5: x'),
              mt_env,
              mt_store)
       ~is res(intV(5),
               ...., cell(1, intV(5)), ....)
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('let x = 5: x'),
              mt_env,
              mt_store)
       ~is res(intV(5),
               override_store(cell(1, intV(5)),
                              mt_store))
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('x'),
              ....,
              ....)
       ~is ....
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('x'),
              extend_env(bind(#'x, ....),
                         mt_env),
              ....)
       ~is ....
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('x'),
              extend_env(bind(#'x, 1),
                         mt_env),
              override_store(cell(1, ....),
                             mt_store))
       ~is ....
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('x'),
              extend_env(bind(#'x, 1),
                         mt_env),
              override_store(cell(1, intV(5)),
                             mt_store))
       ~is res(intV(5),
               ....)
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('x'),
              extend_env(bind(#'x, 1),
                         mt_env),
              override_store(cell(1, intV(5)),
                             mt_store))
       ~is res(intV(5),
               override_store(cell(1, intV(5)),
                              mt_store))
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('(fun (x): x + x)(8)'),
              mt_env,
              mt_store)
       ~is res(....,
               ....)
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('(fun (x): x + x)(8)'),
              mt_env,
              mt_store)
       ~is res(intV(16),
               ....)
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('(fun (x): x + x)(8)'),
              mt_env,
              mt_store)
       ~is res(intV(16),
               ...., cell(1, intV(8)), ....)
```

Variable Examples

```
interp :: (Exp, Env, Store) -> Result

check: interp(parse('(fun (x): x + x)(8)'),
              mt_env,
              mt_store)
       ~is res(intV(16),
               override_store(cell(1, intV(8)),
                              mt_store))
```

```
(fun (x): unbox(x) + unbox(x))(box(8))
```

Part 3

`interp` for Variables

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | idE(s): res(fetch(lookup(s, env), sto),
                  sto)
    ....
```

`interp` for Variables

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | letE(n, rhs, body):
        reslet (v_rhs, sto_rhs) = interp(rhs, env, sto):
          let l = new_loc(sto_rhs):
            interp(body,
                   extend_env(bind(n, l),
                              env),
                   override_store(cell(l, v_rhs),
                                  sto_rhs))
    ....
```

`interp` for Variables

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | appE(fn, arg):
        reslet (v_f, sto_f) = interp(fn, env, sto):
          reslet (v_a, sto_a) = interp(arg, env, sto_f):
            match v_f
            | closV(n, body, c_env):
                let l = new_loc(sto_a):
                  interp(body,
                         extend_env(bind(n, l),
                                    c_env),
                         override_store(cell(l, v_a),
                                        sto_a))
            | ~else: error(#'interp, "not a function")
    ....
```

Part 4

Boxes vs. Variables

```
let x = 5:
  let f = (fun (y):
             x + y):
    begin:
      x := 6
      f(1)
```

```
let x = box(5):
  let f = (fun (y):
             unbox(x) + y):
    begin:
      set_box(x, 6)
      f(1)
```

Boxes as Values

```
let fill = (fun (b):
             set_box(b, 5)):
  let a = box(0):
    begin:
      fill(a)
      unbox(a)
```

⇒ 5

```
let maybe_fill = (fun (b):
                    b := 5):
  let a = 0:
    begin:
      maybe_fill(a)
      a
```

⇒ 0

Boxes as Values

```
let fill = (fun (b):
             set_box(b, 5)):
  let a = box(0):
    begin:
      fill(a)
      unbox(a)
```

⇒ 5

```
let fill = (fun (b):
             b(5)):
  let a = 0:
    begin:
      fill(fun (v): a := v)
      a
```

⇒ 5

Boxes as Variables and Functions

```
fun crate(val):
  def mutable var = val
  values(fun (): var,
         fun (x): var := x)

fun uncrate(b):
  let get = fst(b):
    get()

fun set_crate(b, new_v):
  let set = snd(b):
    set(new_v)
```

Boxes as Variables and Functions

```
let crate = (fun (v):
              fun (sel):
                sel(fun (x): v)(fun (x): v := x)):
  let uncrate = (fun (b):
                   b(fun (x): fun (y): x)(0)):
    let set_crate = (fun (b):
                       fun (v):
                         b(fun (x): fun (y): y)(v)):
      let b = crate(0):
        begin:
          set_crate(b)(5)
          uncrate(b)
```

Boxes vs. Variables

Mutable variables and mutable structures have the same expressive power

Part 5

Mutating Variables

```
fun swap(x, y):
  let z = y:
    y := x
    x := z

let a = 10:
  let b = 20:
    begin:
      swap(a, b)
      a
```

Result would be `10`: assignment in `swap` cannot affect `a`

Mutating Variables

```
let maybe_fill = (fun (b):
                    b := 5):
  let a = 0:
    begin:
      maybe_fill(a)
      a
```

Result is `0`...

but what if we want a language where the result is `5`?

Call by Value

```
let maybe_fill = (fun (b):
                    b := 5):
  let a = 0:
    begin:
      maybe_fill(a)
      a
```

⇒

```
let maybe_fill = (fun (b_b):
                    set_box(b_b, 5)):
  let a = box(0):
    begin:
      maybe_fill(box(unbox(a)))
      unbox(a)
```

Call by Reference

```
let maybe_fill = (fun (b):
                    b := 5):
  let a = 0:
    begin:
      maybe_fill(a)
      a
```

⇒

```
let maybe_fill = (fun (b_b):
                    set_box(b_b, 5)):
  let a = box(0):
    begin:
      // maybe_fill(box(unbox a))
      maybe_fill(a)
      unbox(a)
```

This is called ***call by reference***, as opposed to ***call by value***

Implementing Call-by-Reference

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | appE(fn, arg):
        reslet (v_f, sto_f) = interp(fn, env, sto):
          reslet (v_a, sto_a) = interp(arg, env, sto_f):
            ....
    ....
```

Implementing Call-by-Reference

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | appE(fn, arg):
        reslet (v_f, sto_f) = interp(fn, env, sto):
          match arg
          | idE(s):
              ....
          | ~else:
              reslet (v_a, sto_a) = interp(arg, env, sto_f):
                ....
    ....
```

Implementing Call-by-Reference

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | appE(fn, arg):
        reslet (v_f, sto_f) = interp(fn, env, sto):
          match arg
          | idE(s):
              match v_f
              | closV(n, body, c_env):
                  interp(body,
                         extend_env(bind(n, lookup(s, env)),
                                    c_env),
                         sto_f)
              | ~else: error(....)
          | ~else:
              reslet (v_a, sto_a) = interp(arg, env, sto_f):
                ....
    ....
```

Implementing Call-by-Reference

```
def interp :: (Exp, Env, Store) -> Result:
  fun (a, env, sto):
    ....
    | appE(fn, arg):
        reslet (v_f, sto_f) = interp(fn, env, sto):
          match arg
          | idE(s):
              match v_f
              | closV(n, body, c_env):
                  interp(body,
                         extend_env(bind(n, lookup(s, env)),
                                    c_env),
                         sto_f)
              | ~else: error(....)
          | ~else:
              reslet (v_a, sto_a) = interp(arg, env, sto_f):
                ....
    ....
```