Part I

Boxes vs. Variables

```
{let {[x 5]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 6}
      {f 1}}}}
```

Boxes vs. Variables

```
{let {[x 5]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 6}
      {f 1}}}}
```

```
{let {[x {box 5}]}
  {let {[f {lambda {y}
             {+ {unbox x} y}}]}
    {begin
      {set-box! x 6}
      {f 1}}}}
```

Boxes vs. Variables

```
{let {[x 5]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 6}
      {f 1}}}}
```

```
{let {[x {box 5}]}
  {let {[f {lambda {y}
             {+ {unbox x} y}}]}
    {begin
      {set-box! x 6}
      {f 1}}}}
```

Boxes vs. Variables

```
{let {[x 5]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 6}
      {f 1}}}}
```

```
{let {[x {box 5}]}
  {let {[f {box {lambda {y}
                  {+ {unbox x} y}}}]}
    {begin
      {set-box! x 6}
      {{unbox f} 1}}}}
```

Boxes vs. Variables

```
{let {[x 5]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 6}
      {f 1}}}}
```

```
{let {[x {box 5}]}
  {let {[f {box {lambda {y}
                  {+ {unbox x} {unbox y}}}}]}
    {begin
      {set-box! x 6}
      {{unbox f} {box 1}}}}}
```

Boxes vs. Variables

```
{let {[x 5]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 6}
      {f 1}}}}
```

```
(define-type Binding
  [bind (name : symbol)
        (location : Location)])
```

```
{let {[x {box 5}]}
  {let {[f {box {lambda {y}
                  {+ {unbox x} {unbox y}}}}]}
    {begin
      {set-box! x 6}
      {{unbox f} {box 1}}}}}
```

Variables

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {- <Expr> <Expr>}
          |  <Sym>
          |  {lambda {<Sym>} <Expr>}
          |  {<Expr> <Expr>}
          |  {set! <Expr> <Expr>}
          |  {begin <Expr> <Expr>}
```

NEW

```
{let {[b 0]}
  {begin
    {set! b 10}
    b}}                ⇒ 10
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (numC 5) mt-env mt-store)
      (v*s (numV 5) mt-store))
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse '{let {[x 5]} x})
              mt-env
              mt-store)
      (v*s ...
           ...))
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse '{let {[x 5]} x})
              mt-env
              mt-store)
      (v*s (numV 5)
           ...))
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse '{let {[x 5]} x})
              mt-env
              mt-store)
      (v*s (numV 5)
           ... (cell 1 (numV 5)) ...))
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse '{let {[x 5]} x})
              mt-env
              mt-store)
      (v*s (numV 5)
           (override-store
            (cell 1 (numV 5))
            mt-store)))
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse `x)
              ...
              ...)
      ...)
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse `x)
              (extend-env (bind 'x ...)
                          mt-env)
              ...)
      ...)
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse `x)
              (extend-env (bind 'x 1)
                          mt-env)
              (override-store (cell 1 ...)
                              mt-store))
      ...)
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse `x)
              (extend-env (bind 'x 1)
                          mt-env)
              (override-store (cell 1 (numV 5))
                              mt-store))
      (v*s (numV 5)
           ...))
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse `x)
              (extend-env (bind 'x 1)
                          mt-env)
              (override-store (cell 1 (numV 5))
                              mt-store))
      (v*s (numV 5)
           (override-store (cell 1 (numV 5))
                           mt-store)))
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse '{{lambda {x} {+ x x}}
                       8})
              mt-env
              mt-store)
      (v*s ...
           ...))
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse '{{lambda {x} {+ x x}}
                       8})
              mt-env
              mt-store)
      (v*s (numV 16)
           ...))
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse '{{lambda {x} {+ x x}}
                       8})
              mt-env
              mt-store)
      (v*s (numV 16)
           ... (cell 1 (numV 8)) ...))
```

Variable Examples

```
interp : (ExprC Env Store -> Result)

(test (interp (parse '{{lambda {x} {+ x x}}
                       8})
              mt-env
              mt-store)
      (v*s (numV 16)
           (override-store (cell 1 (numV 8))
                           mt-store)))
```

```
{{lambda {x} {+ {unbox x} {unbox x}}}
 {box 8}}
```

Part 2

`interp` for Variables

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [idC (s) (v*s (fetch (lookup s env) sto)
                  sto)]
    ...))
```

`interp` for Variables

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [letC (n rhs body)
          (with [(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

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [setC (var val)
          (let ([l (lookup var env)])
            (with [(v-v sto-v) (interp val env sto)]
              (v*s v-v
                   (override-store (cell l v-v)
                                   sto-v))))]
    ...))
```

Part 3

Boxes vs. Variables

```
{let {[x 5]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 6}
      {f 1}}}}
```

```
{let {[x {box 5}]}
  {let {[f {lambda {y}
             {+ {unbox x} y}}]}
    {begin
      {set-box! x 6}
      {f 1}}}}
```

Boxes as Values

```
{let {[fill! {lambda {b}
               {set-box! b 5}}]}
  {let {[a {box 0}]}
    {begin
      {fill! a}
      {unbox a}}}} ⇒ 5
```

```
{let {[fill?! {lambda {b}
                {set! b 5}}]}
  {let {[a 0]}
    {begin
      {fill?! a}
      a}}}
⇒ 0
```

Boxes as Values

```
{let {[fill! {lambda {b}
               {set-box! b 5}}]}
  {let {[a {box 0}]}
    {begin
      {fill! a}
      {unbox a}}}} ⇒ 5
```

```
{let {[fill {lambda {b}
              {b 5}}]}
  {let {[a 0]}
    {begin
      {fill {lambda {v} {set! a v}}}
      a}}}
⇒ 5
```

Boxes as Variables and Functions

```
(define (crate v)
  (values (lambda () v)
          (lambda (x) (set! v x))))

(define (uncrate b)
  (local [(define-values (get set) b)]
    (get)))

(define (set-crate! b new-v)
  (local [(define-values (get set) b)]
    (set new-v)))
```

Boxes as Variables and Functions

```
{let {[crate
       {lambda {v}
         {lambda {sel}
           {{sel
             {lambda {x} v}}
            {lambda {x} {set! x v}}}}}]}
  {let {[uncrate
         {lambda {b}
           {{b {lambda {x} {lambda {y} x}}} 0}}]}
    {let {[set-crate!
           {lambda {b}
             {lambda {v}
               {{b {lambda {x} {lambda {y} y}} v}}}}]}
      {let {[b {create 0}]}
        {begin
          {{set-crate! b} 5}
          {uncrate b}}}}}}
```

Boxes vs. Variables

Mutable variables and mutable structures have the same expressive power

Part 4

Mutating Variables

```
(define (swap x y)
  (let ([z y])
    (set! y x)
    (set! x z)))

(let ([a 10])
  (let ([b 20])
    (begin
      (swap a b)
      a)))
```

Result is `10`: assignment in `swap` cannot affect `a`

Mutating Variables

```
{let {[fill?! {lambda {b}
                {set! b 5}}]}
  {let {[a 0]}
    {begin
      {fill?! a}
      a}}}
```

Result is `0`...

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

Call by Value

```
{let {[fill?! {lambda {b}
                {set! b 5}}]}
  {let {[a 0]}
    {begin
      {fill?! a}
      a}}}

⇒ {let {[fill?! {lambda {b-b}
                  {set-box! b-b 5}}]}
    {let {[a {box 0}]}
      {begin
        {fill?! {box {unbox a}}}
        {unbox a}}}}
```

Call by Reference

```
{let {[fill?! {lambda {b}
                {set! b 5}}]}
  {let {[a 0]}
    {begin
      {fill?! a}
      a}}}

⇒ {let {[fill?! {lambda {b-b}
                  {set-box! b-b 5}}]}
    {let {[a {box 0}]}
      {begin
        ; {fill?! {box {unbox a}}}
        {fill?! a}
        {unbox a}}}}
```

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

Implementing Call-by-Reference

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [appC (fun arg)
          (with [(v-f sto-f) (interp fun env sto)]
            (with [(v-a sto-a) (interp arg env sto-f)]
              ....))]
    ...))
```

Implementing Call-by-Reference

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [appC (fun arg)
          (with [(v-f sto-f) (interp fun env sto)]
            (type-case ExprC arg
              [idC (s)
               ...]
              [else
               (with [(v-a sto-a) (interp arg env sto-f)]
                 ....)]))]
    ...))
```

Implementing Call-by-Reference

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [appC (fun arg)
          (with [(v-f sto-f) (interp fun env sto)]
            (type-case ExprC arg
              [idC (s)
               (type-case Value v-f
                 [closV (n body c-env)
                        (interp body
                                (extend-env
                                 (bind n (lookup s env))
                                 c-env)
                                sto-f)]
                 [else (error ...)])]
              [else
               (with [(v-a sto-a) (interp arg env sto-f)]
                 ....)]))]
    ...))
```

Implementing Call-by-Reference

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [appC (fun arg)
          (with [(v-f sto-f) (interp fun env sto)]
            (type-case ExprC arg
              [idC (s)
               (type-case Value v-f
                 [closV (n body c-env)
                        (interp body
                                (extend-env
                                 (bind n (lookup s env))
                                 c-env)
                                sto-f)]
                 [else (error ...)])]
              [else
               (with [(v-a sto-a) (interp arg env sto-f)]
                 ....)]))]
    ...))
```