# Part I

# State

Substitution relies on an identifier having a fixed value

```
{let {[x 5]}
  {let {[f {lambda {y} {+ x y}}]}
    ...
    {f 1}}}
=
{let {[f {lambda {y} {+ 5 y}}]}
  ...
  {f 1}}
```

because `x` cannot change

# State

In **plai-typed**, a variable's value *can* change

```
> (let ([x 5])
    (let ([f (lambda (y) (+ x y))])
      (begin
        (set! x 6)
        (f 1))))
- number
7
```

A variable has ***state***

Assignment to variables in **plai-typed** is strongly discouraged, but in other languages...

# Inessential State: Summing a List

The Java way:

```
int sum(List<Integer> l) {
  int t = 0;
  for (Integer n : l) {
    t = t + n;
  }
  return t;
}
```

The **plai-typed** way:

```
(define (sum [l : (listof number)]) : number
  (cond
   [(empty? l) 0]
   [else (+ (first l) (sum (rest l)))]))
```

# Inessential State: Summing a List

The Java way:

```
int sum(List<Integer> l) {
  int t = 0;
  for (Integer n : l) {
    t = t + n;
  }
  return t;
}
```

The **plai-typed** way:

```
(define (sum [l : (listof number)] [t : number])
  (cond
   [(empty? l) t]
   [else (sum (rest l) (+ (first l) t))]))
```

# Inessential State: Summing a List

The Java way:

```
int sum(List<Integer> l) {
  int t = 0;
  for (Integer n : l) {
    t = t + n;
  }
  return t;
}
```

The **plai-typed** way:

```
(define (sum [l : (listof number)]) : number
  (foldl + 0 l))
```

# Inessential State: Feeding Fish

The Java way:

```
void feed(int[] aq) {
  for (int i = 0; i < aq.length; i++) {
    aq[i]++;
  }
}
```

The **plai-typed** way:

```
(define feed : ((listof number) -> (listof number))
  (lambda (l)
    (map (lambda (x) (+ x 1)) l)))
```

# Reasons to Avoid State

```
(test (feed (list 4 3 7 1))
      (list 5 4 8 2))
```

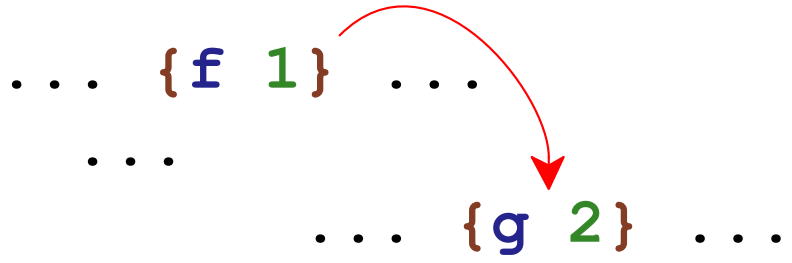
```
(define today (list 4 3 7 1))
(define tomorrow (feed today))
(compare today tomorrow)
```

# When State is Essential


```
(define weight 0)

(define total-message (make-message (to-string weight)))

(define (make-feed-button label amt)
  (make-button label
               (lambda (evt)
                 (begin
                   (set! weight (+ weight amt))
                   (draw-message total-message
                                 (to-string weight))))))

(create-window (list (list total-message)
                     (list (make-feed-button "Feed 3" 3)
                           (make-feed-button "Feed 7" 7))))
```

# State as a Side Channel

State is a ***side channel*** for parts of a program to communicate




+ Programmer can add new channels at will

− Channels of communication may not be apparent

# Part 2

# Variables vs. Boxes

```
(define weight 0)

(define (feed!) : void
  (set! weight (+ 1 weight)))

(define (get-size) : number
  weight)
```

# Variables vs. Boxes

```
(define weight (box 0))

(define (feed!) : void
  (set-box! weight (+ 1 (unbox weight))))

(define (get-size) : number
  (unbox weight))
```

```
box : ('a -> (boxof 'a))
unbox : ((boxof 'a) -> 'a)
set-box! : ((boxof 'a) 'a -> void)
```

# Boxes as Simple Objects

```
class Box<T> {
  T v;
  Box(T v) {
    this.v = v;
  }
}
```

```
(let ([b (box 0)])
  (begin
    (set-box! b 10)
    (unbox b)))
```

```
Box b = new Box(0);

b.v = 10;
return b.v;
```

# Boxes

```
<Expr>  ::=  <num>
          |  {+ <Expr> <Expr>}
          |  {- <Expr> <Expr>}
          |  <Sym>
          |  {lambda {<Sym>} <Expr>}
          |  {<Expr> <Expr>}
          |  {box <Expr>}                    NEW
          |  {unbox <Expr>}                  NEW
          |  {set-box! <Expr> <Expr>}        NEW
          |  {begin <Expr> <Expr>}           NEW
```

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

# Implementing Boxes

```
(define-type ExprC
  ...
  [boxC (arg : ExprC)]
  [unboxC (arg : ExprC)]
  [setboxC (bx : ExprC)
           (val : ExprC)]
  [beginC (l : ExprC)
          (r : ExprC)])
```

# Part 3

# Implementing Boxes with Boxes

```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol)
         (body : ExprC)
         (env : Env)]
  [boxV (b : (boxof Value))])
```

# Implementing Boxes with Boxes

```
(define (interp [a : ExprC] [env : Env]) : Value
  (type-case Expr a
    ...
    [boxC (a)
          (boxV (box (interp a env)))]
    [unboxC (a)
            (type-case Value (interp a env)
              [boxV (v) (unbox v)]
              [else (error 'interp "not a box")])]
    [setboxC (bx val)
             (type-case Value (interp bx env)
               [boxV (v) (let ([v (interp val env)])
                           (begin (set-box! b v)
                                  v))]
               [else (error 'interp "not a box")])]
    [beginC (l r) (begin
                    (interp l env)
                    (interp r env))]))
```

This doesn't explain anything about boxes!

# Part 4

# State and `interp`

We don't need state to `interp` state

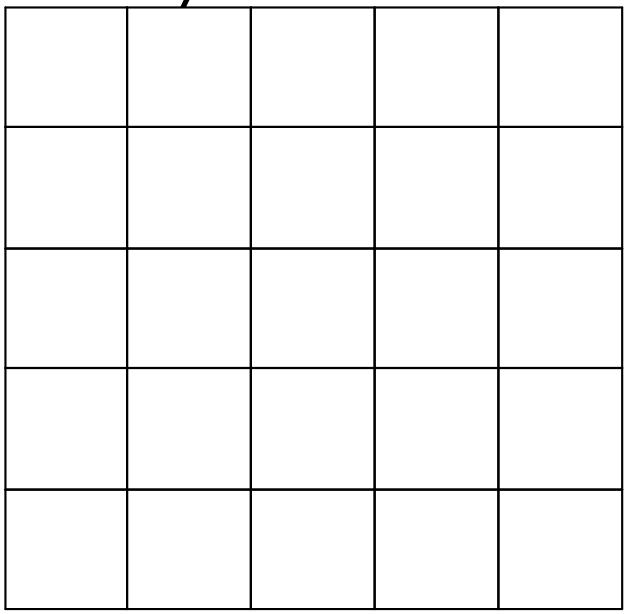
- We control all the channels of communication
- Communicate the current values of boxes explicitly

# Boxes and Memory

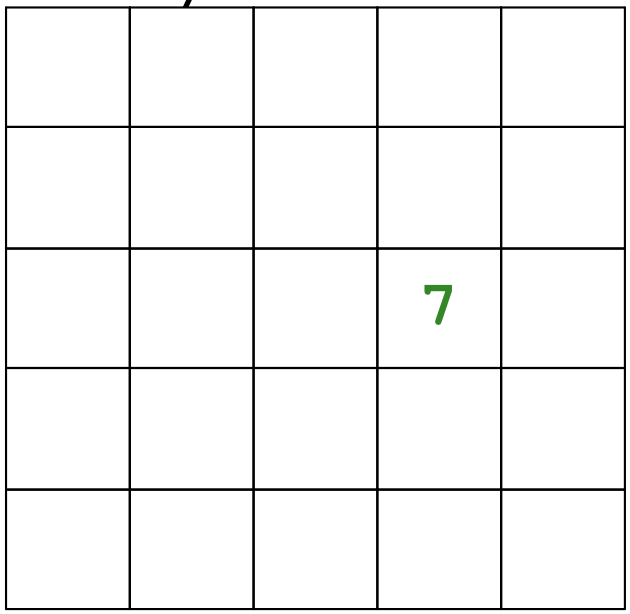
```
{let {[b {box 7}]}
  ...}
```

⇒ `...`

*Memory:*



*Memory:*



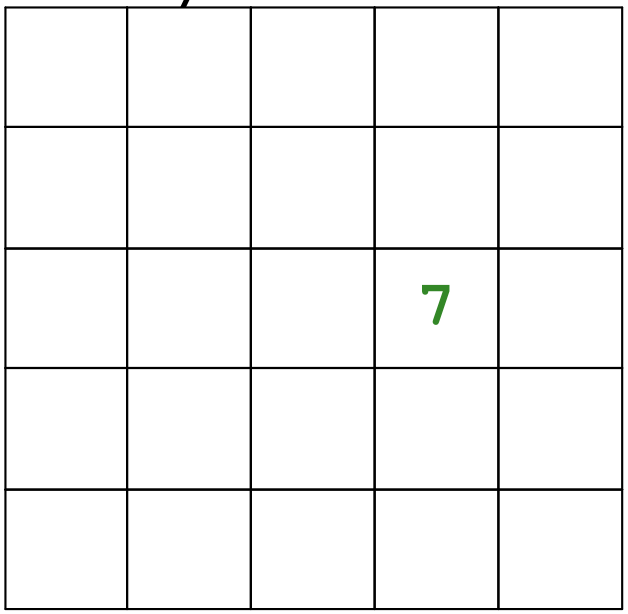
# Boxes and Memory

```
... {set-box! b 10}
...
```

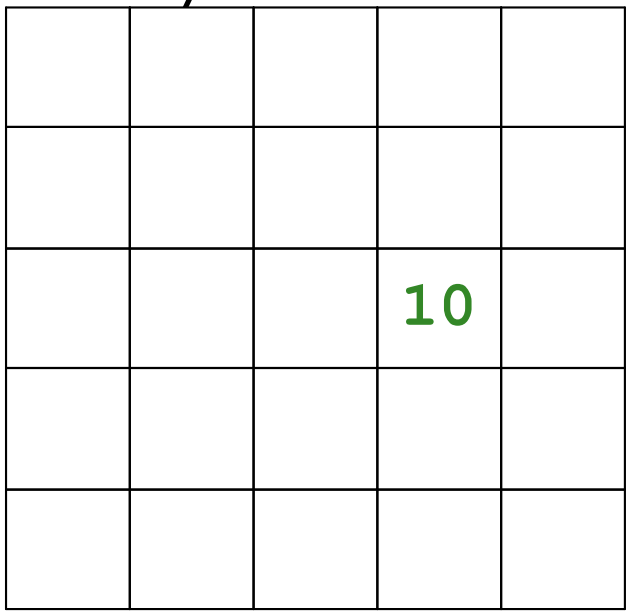
⇒

```
... {unbox b}
...
```

*Memory:*




*Memory:*

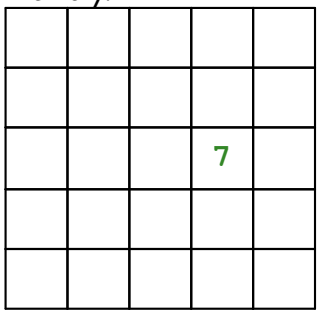


# Communicating Memory

```
(interp ....  ) ⇒ ...
```

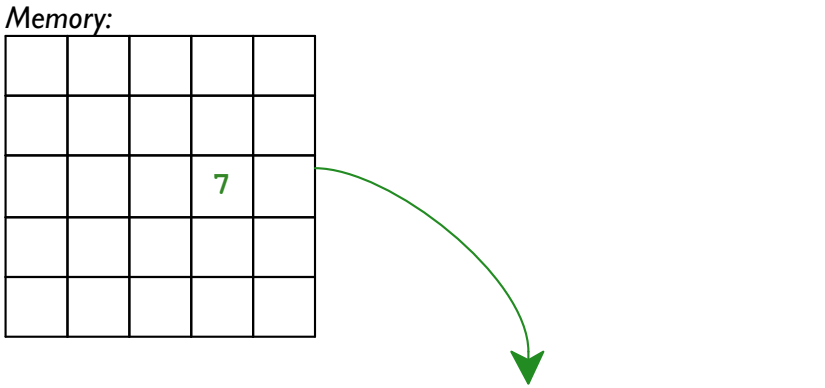
# Communicating Memory

*Memory:*

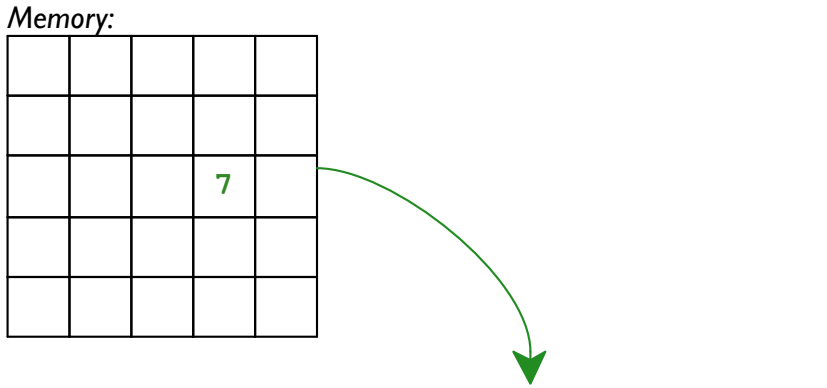



```
(interp ....  ) ⇒ ...
```

# Communicating Memory

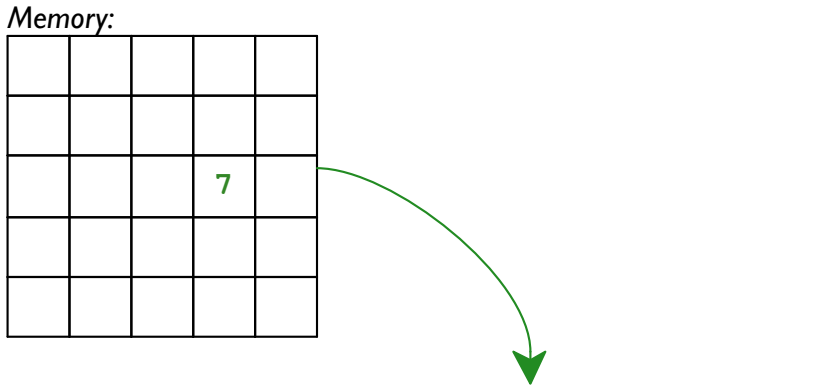


# Communicating Memory



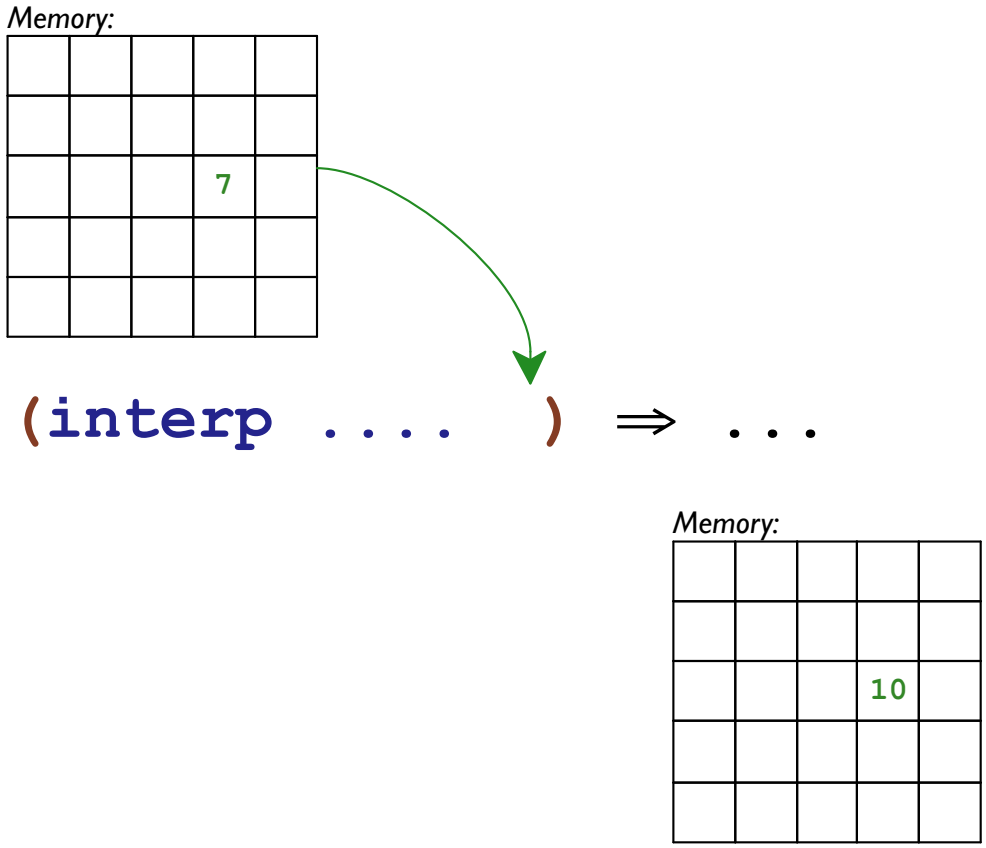

```
interp : (Expr Env -> Value)
```

# Communicating Memory



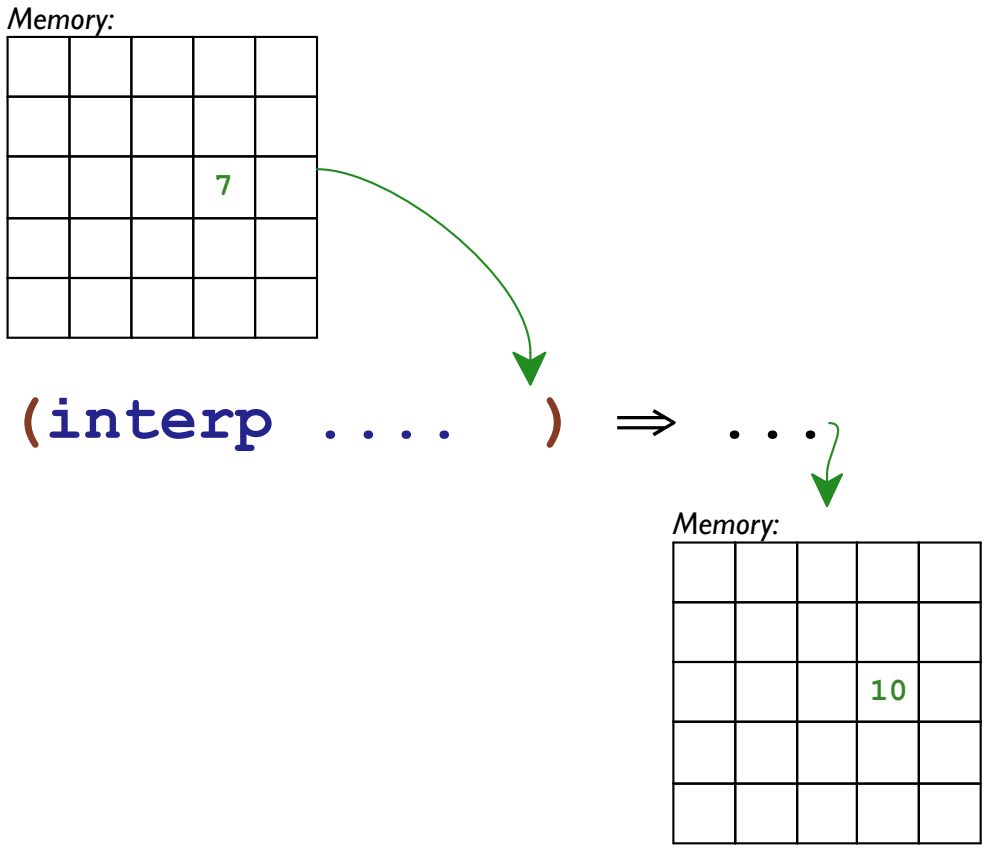

```
interp : (Expr Env Store -> Value)
```

# Communicating Memory



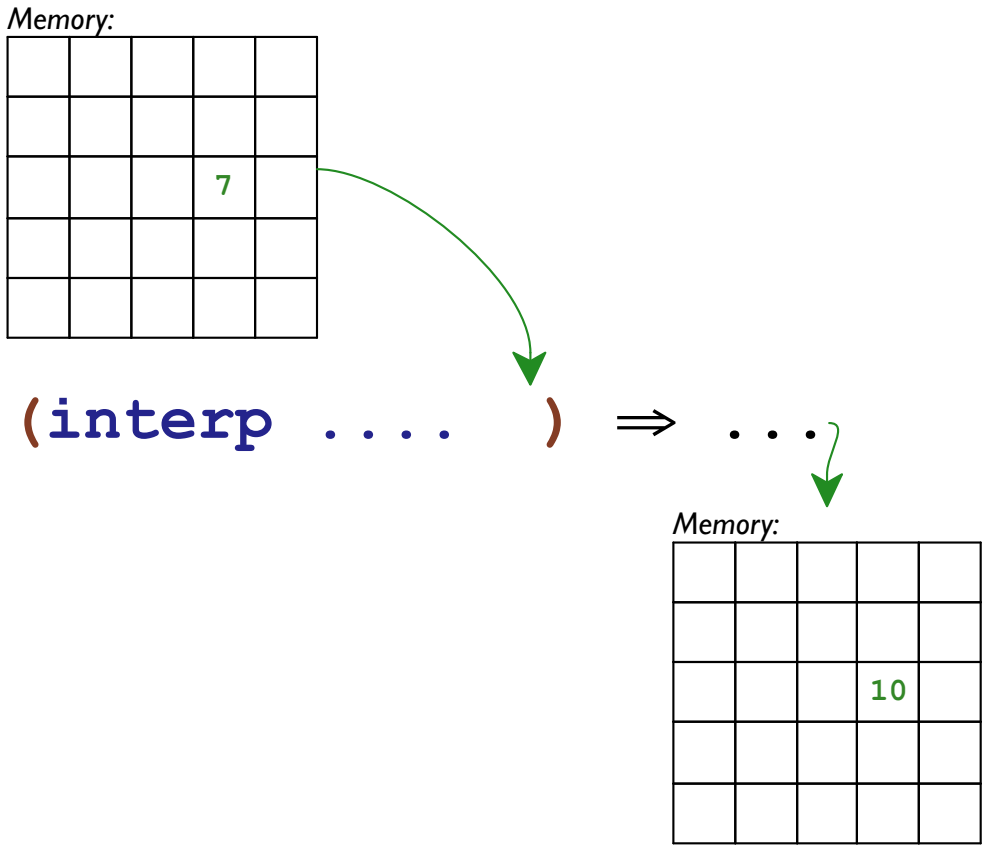

```
interp : (Expr Env Store -> Value)
```

# Communicating Memory



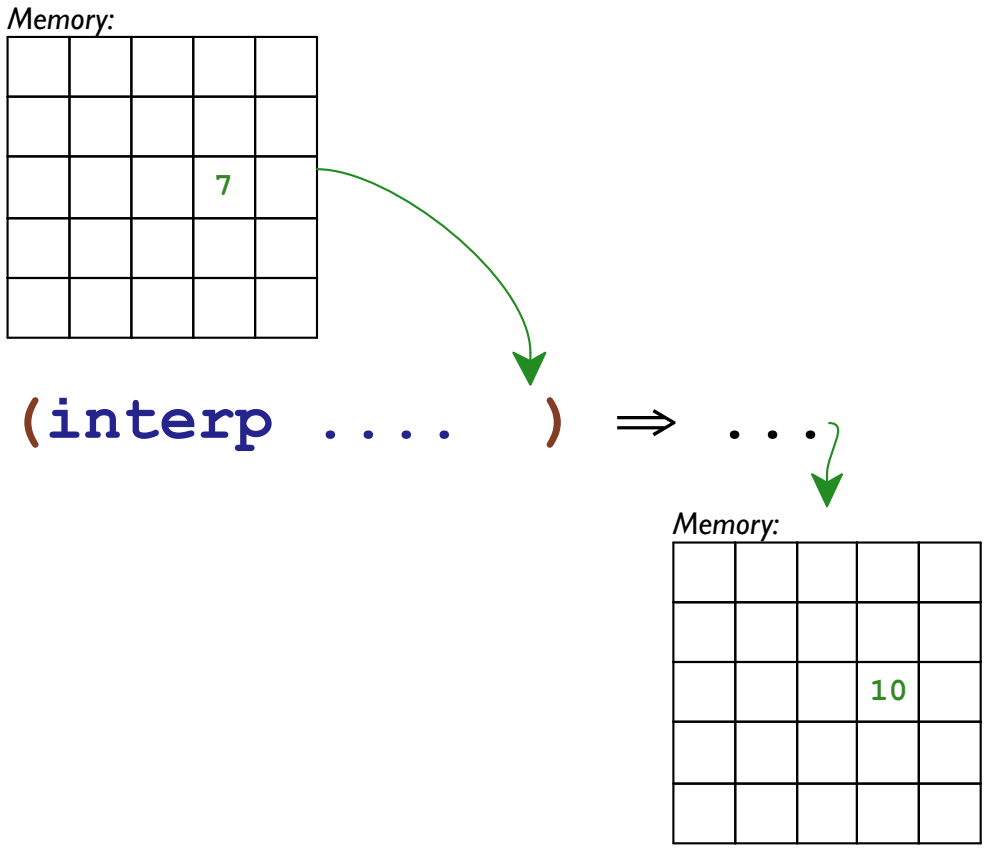

```
interp : (Expr Env Store -> Value)
```

# Communicating Memory




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

# Communicating Memory




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

(define-type Result
  [v*s (v : Value) (s : Store)])
```

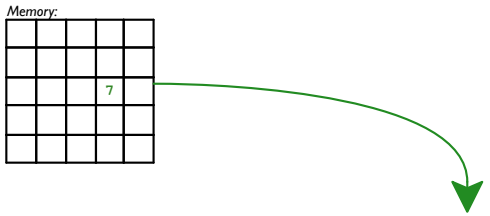
# Communicating the Store

```
(num+ (interp l env) (interp r env))
```

# Communicating the Store

```
(interp l env)

(interp r env)

(num+ ... ...)
```

# Communicating the Store

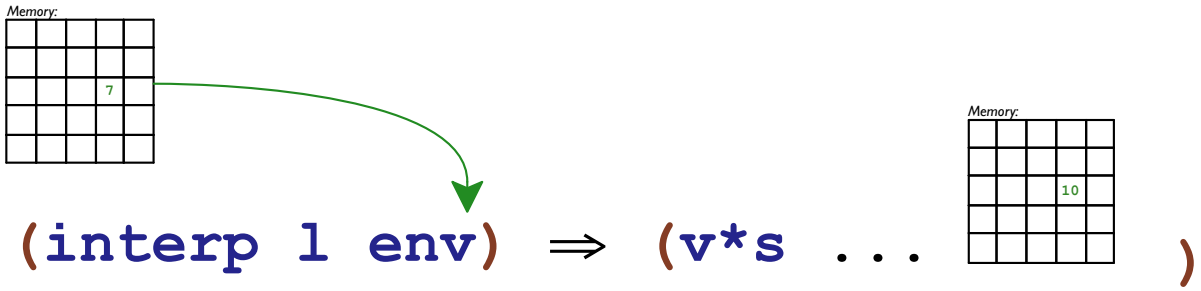



```
(interp l env)
```

```
(interp r env)
```

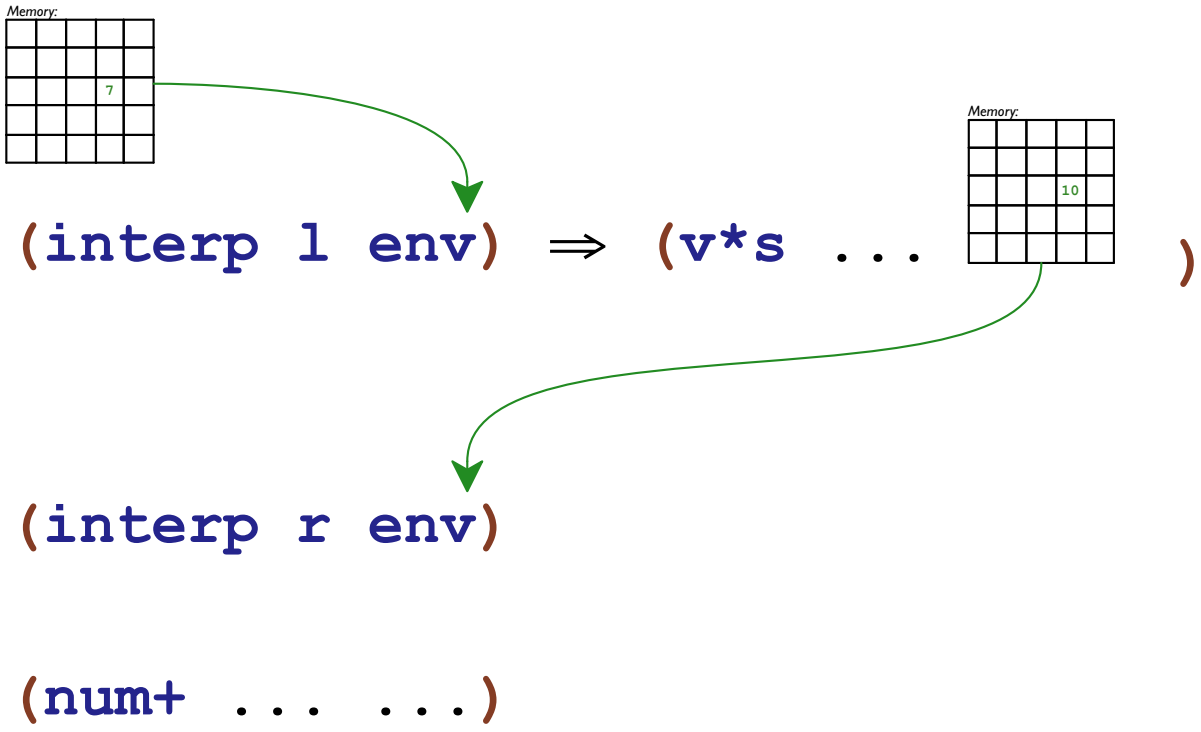
```
(num+ ... ...)
```

# Communicating the Store

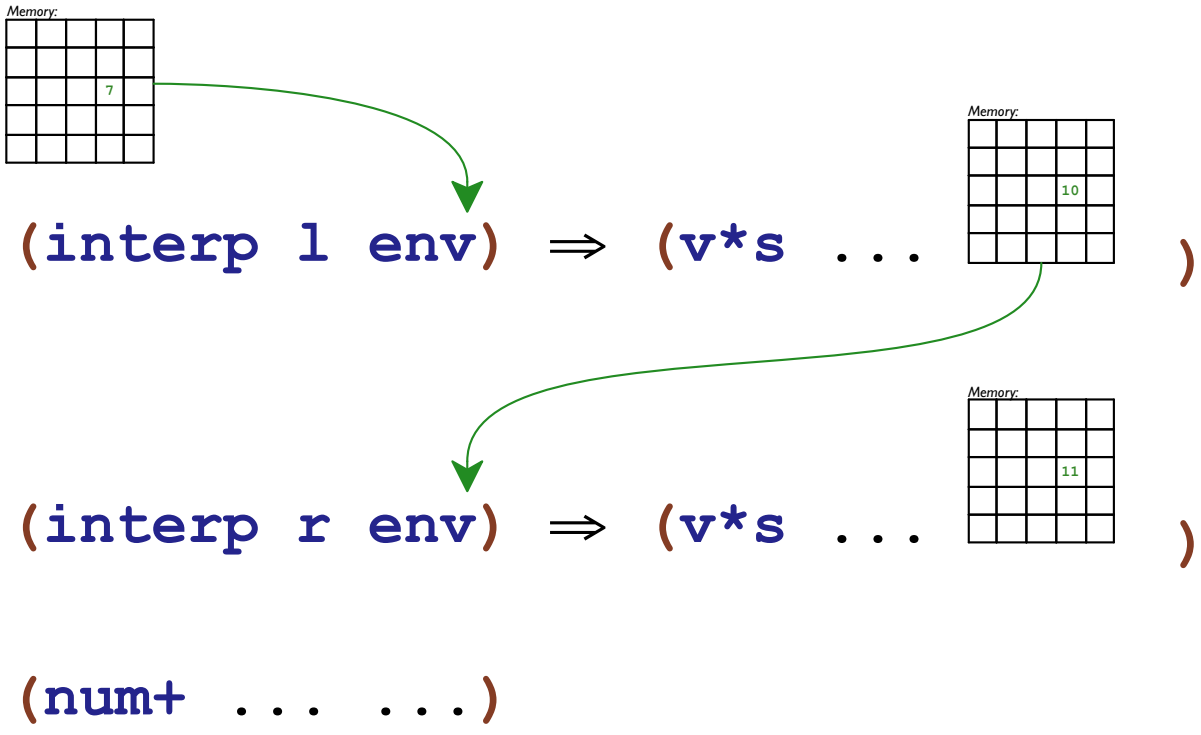



```
(interp r env)

(num+ ... ...)
```

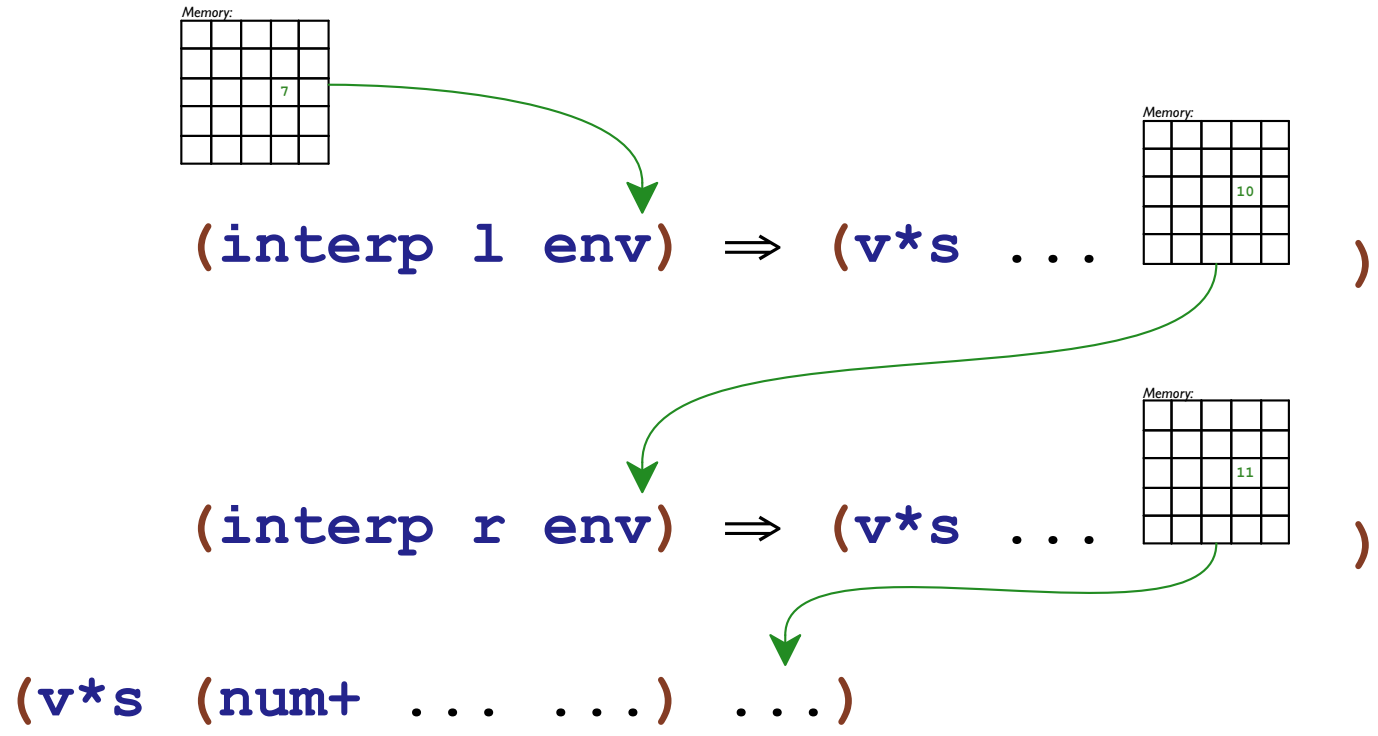
# Communicating the Store



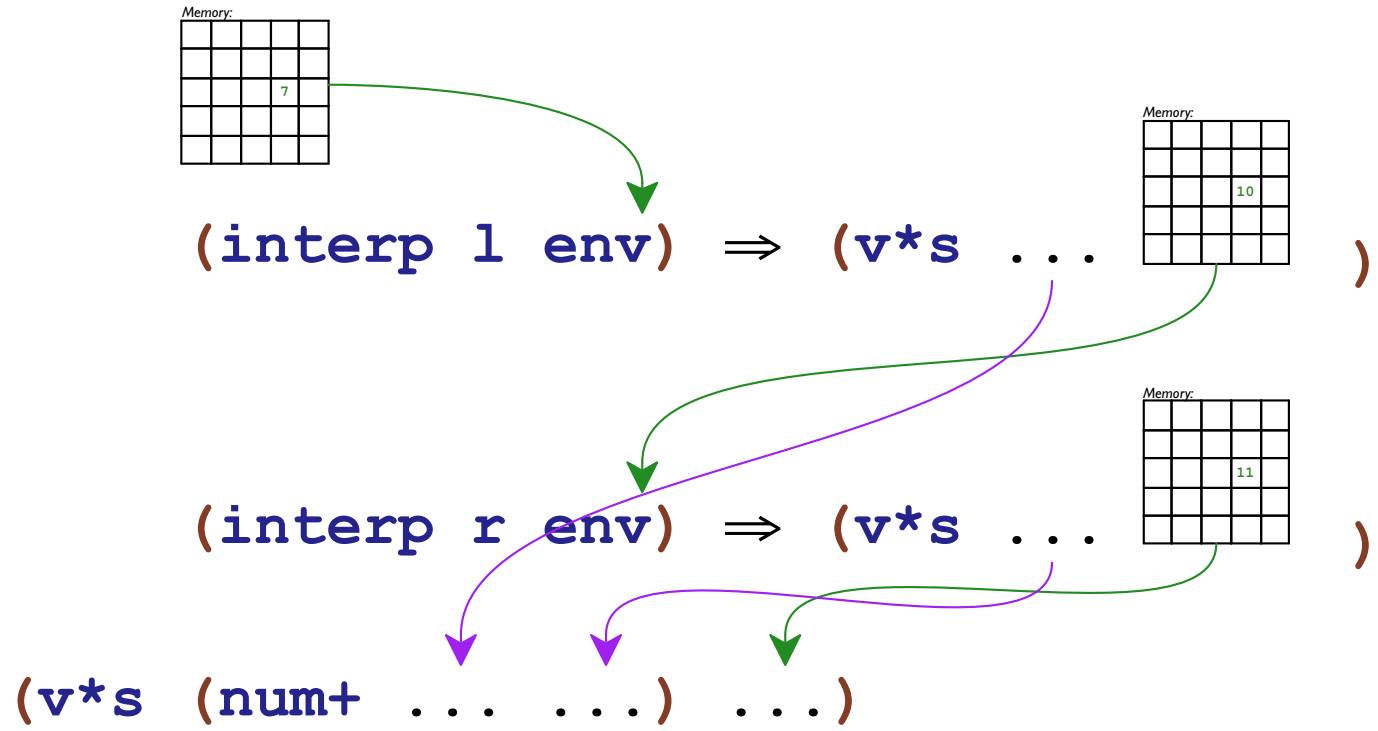
# Communicating the Store



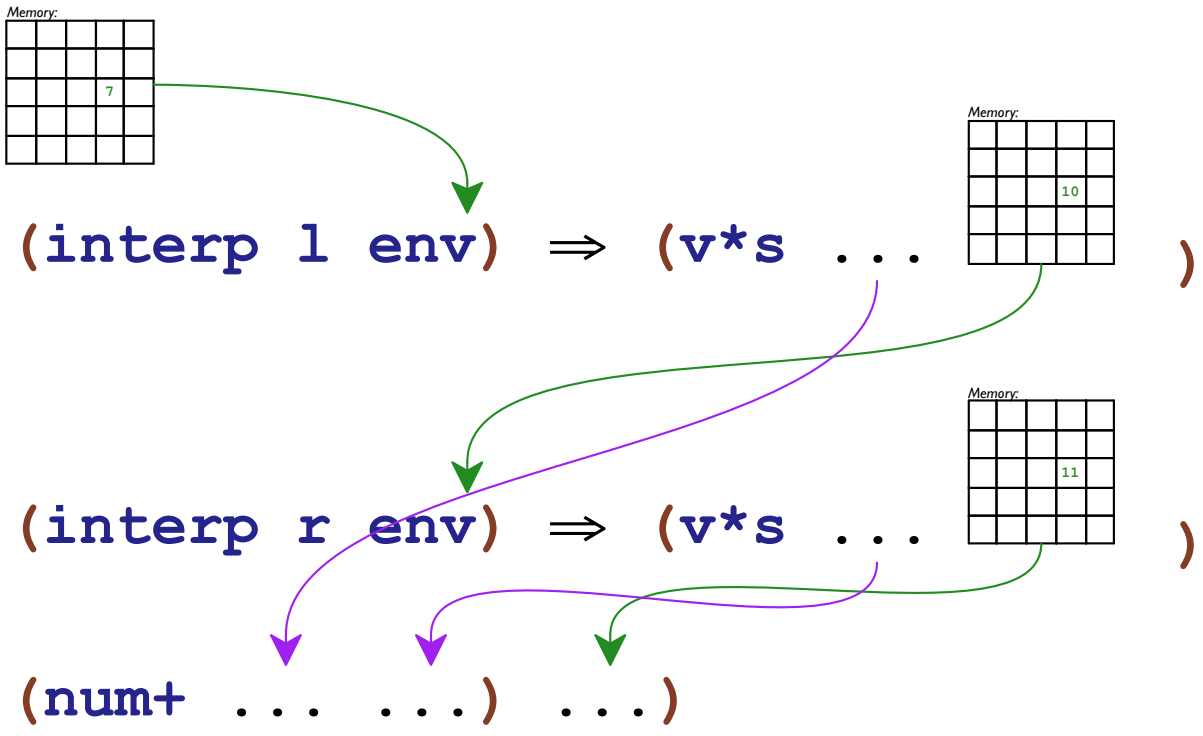
# Communicating the Store



# Communicating the Store



# Communicating the Store

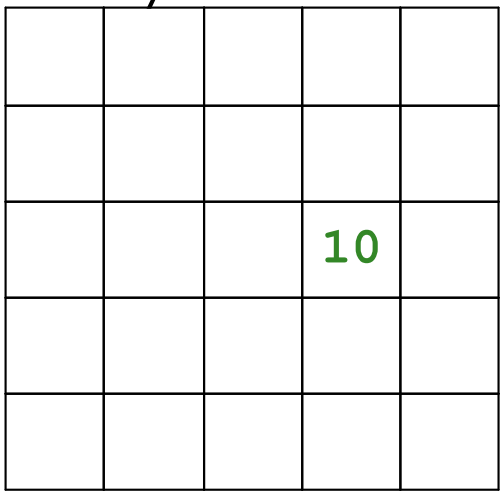



```
(type-case Result (interp l env sto)
  [v*s (v-l sto-l)
       (type-case Result (interp r env sto-l)
         [v*s (v-r sto-r)
              (v*s (num+ v-l v-r) sto-r)])])
```

# The Store

```
(define-type-alias Location number)

(define-type Storage
  [cell (location : Location) (val : Value)])

(define-type-alias Store (listof Storage))
(define mt-store empty)
(define override-store cons)
```

*Memory:*




```
(override-store (cell 13 (numV 10))
                mt-store)
```

# Part 5

# Store Examples

```
interp : (ExprC Env -> Value)

(test (interp (numC 5) mt-env)
      (numV 5))
```

# Store Examples

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

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

# Store Examples

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

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

# Store Examples

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

(test (interp (boxC (numC 5)) mt-env mt-store)
      ...)
```

# Store Examples

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

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

# Store Examples

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

(test (interp (boxC (numC 5)) mt-env mt-store)
      (v*s (boxV ...)
           ...))
```

# Store Examples

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

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

# Store Examples

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

(test (interp (boxC (numC 5)) mt-env mt-store)
      (v*s (boxV ...)
           ... (cell 1 (numV 5)) ...))
```

# Store Examples

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

(test (interp (boxC (numC 5)) mt-env mt-store)
      (v*s (boxV 1)
           ... (cell 1 (numV 5)) ...))


           (define-type Value
             [numV (n number?)]
             [closV (arg : symbol)
                    (body : ExprC)
                    (env : Env)]
             [boxV (l : Location)])
```

# Store Examples

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

(test (interp (boxC (numC 5)) mt-env mt-store)
      (v*s (boxV 1)
           (override-store
            (cell 1 (numV 5))
            mt-store)))
```

# Store Examples

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

(test (interp (parse '{set-box! {box 5} 6})
              mt-env
              mt-store)
      (v*s ...
           ...))
```

# Store Examples

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

(test (interp (parse '{set-box! {box 5} 6})
              mt-env
              mt-store)
      (v*s (numV 6)
           ...))
```

# Store Examples

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

(test (interp (parse '{set-box! {box 5} 6})
              mt-env
              mt-store)
      (v*s (numV 6)
           ...
           ...
           (override-store
            (cell 1 (numV 5))
            mt-store)
                    ...))
```

# Store Examples

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

(test (interp (parse '{set-box! {box 5} 6})
              mt-env
              mt-store)
      (v*s (numV 6)
           (override-store
            (cell 1 (numV 6))
            (override-store
             (cell 1 (numV 5))
             mt-store))))
```

# Part 6

# `interp` with a Store

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [numC (n) (v*s (numV n) sto)]
    ...))
```

# `interp` with a Store

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

# `interp` with a Store

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [plusC (l r)
     (type-case Result (interp l env sto)
       [v*s (v-l sto-l)
            (type-case Result (interp r env sto-l)
              [v*s (v-r sto-r)
                   (v*s (num+ v-l v-r) sto-r)])])]
    ...))
```

# `interp` with a Store

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [boxC (a)
          (type-case Result (interp a env sto)
            [v*s (v sto-v)
                 (let ([l (new-loc sto-v)])
                   (v*s (boxV l)
                        (override-store (cell l v)
                                        sto-v)))])]
    ...))
```

# `interp` with a Store

```
(define (new-loc [sto : Store]) : Location
  (+ 1 (max-address sto)))

(define (max-address [sto : Store]) : Location
  (cond
   [(empty? sto) 0]
   [else (max (cell-location (first sto))
              (max-address (rest sto)))]))
```

# `interp` with a Store

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [unboxC (a)
            (type-case Result (interp a env sto)
              [v*s (v sto-v)
                   (type-case Value v
                     [boxV (l) (v*s (fetch l sto-v)
                                    sto-v)]
                     [else (error 'interp
                                  "not a box")])])]
    ...))
```

# `interp` with a Store

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [setboxC (bx val)
     (type-case Result (interp bx env sto)
       [v*s (v-b sto-b)
            (type-case Result (interp val env sto-b)
              [v*s (v-v sto-v)
                   (type-case Value v-b
                     [boxV (l)
                           (v*s v-v
                                (override-store
                                 (cell l v-v)
                                 sto-v))]
                     [else (error 'interp
                                  "not a box")])])])]
    ...))
```

# `interp` with a Store

```
(define interp : (ExprC Env Store -> Result)
  (lambda (a env sto)
    ...
    [beginC (l r)
     (type-case Result (interp l env sto)
       [v*s (v-l sto-l)
            (interp r env sto-l)])]
    ...))
```

# Part 7

# Awkward Syntax

```
(type-case Result (interp l env sto)
  [v*s (v-l sto-l)
       (type-case Result (interp r env sto-l)
         [v*s (v-r sto-r)
              (v*s (num+ v-l v-r) sto-r)])])
```

```
(type-case Result call
  [v*s (v-id sto-id)
       body])
```

# Better Syntax

```
(type-case Result call
  [v*s (v-id sto-id)
       body])
```

# Better Syntax

```
(type-case Result call
  [v*s (v-id sto-id)
       body])
```

```
(with [(v-id sto-id) call]
  body)
```

# Better Syntax

```
                          (with [(v-id sto-id) call]
                            body)
(type-case Result call
  [v*s (v-id sto-id)
       body])
```

# Better Syntax

```
(define-syntax-rule (with [(v-id sto-id) call]
                          body)
  (type-case Result call
    [v*s (v-id sto-id)
         body]))
```

# Better Syntax

```
(define-syntax-rule (with [(v-id sto-id) call]
                          body)
  (type-case Result call
    [v*s (v-id sto-id)
         body]))
```

```
(with [(v-r sto-r) (interp r env sto-l)]
  (v*s (num+ v-l v-r) sto-r))
```

# Better Syntax

```
(define-syntax-rule (with [(v-id sto-id) call]
                          body)
  (type-case Result call
    [v*s (v-id sto-id)
         body]))
```

```
(with [(v-r sto-r) (interp r env sto-l)]
  (v*s (num+ v-l v-r) sto-r))
```

⇒

```
(type-case Result (interp r env sto-l)
  [v*s (r-v sto-r)
       (v*s (num+ v-l v-r) sto-r)])
```

# Better Syntax

```
(with [(v-l sto-l) (interp l env sto)]
  (with [(v-r sto-r) (interp r env sto-l)]
    (v*s (num+ v-l v-r) sto-r)))
```