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 Plait, 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 Plait 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 Plait 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 Plait 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 Plait 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 Plait 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))
```

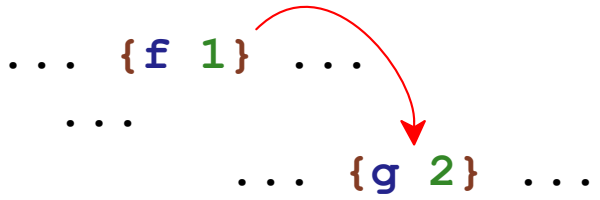
```
(define today (list 4 3 7 1))
(define tomorrow (feed today))
(compare today tomorrow)
```

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

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

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

Box b = new Box(0);

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

Boxes

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {- <Exp> <Exp>}
         |  <Symbol>
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {box <Exp>}                    NEW
         |  {unbox <Exp>}                  NEW
         |  {set-box! <Exp> <Exp>}         NEW
         |  {begin <Exp> <Exp>}            NEW
```

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

Implementing Boxes

```
(define-type Exp
  ...
  (boxE [arg : Exp])
  (unboxE [arg : Exp])
  (setboxE [bx : Exp]
           [val : Exp])
  (beginE [l : Exp]
          [r : Exp]))
```

Part 3

Implementing Boxes with Boxes

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

Implementing Boxes with Boxes

```
{box 0}
```

Implementing Boxes with Boxes

```
{box 0}
```

⇒ ... a box containing `(numV 0)` ...

Implementing Boxes with Boxes

```
(define-type Value
  (numV [n : Number])
  (closV [arg : Symbol]
         [body : Exp]
         [env : Env])
  (boxV [b : (Boxof Value)]))
```

Implementing Boxes with Boxes

```
(define (interp [a : Exp] [env : Env]) : Value
  (type-case Expr a
    ...
    [(boxE a)
     (boxV (box (interp a env)))]
    [(unboxE a)
     (type-case Value (interp a env)
       [(boxV b) (unbox b)]
       [else (error 'interp "not a box")])]
    [(setboxE bx val)
     (type-case Value (interp bx env)
       [(boxV b) (let ([v (interp val env)])
                   (begin (set-box! b v)
                          v))]
       [else (error 'interp "not a box")])]
    [(beginE 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

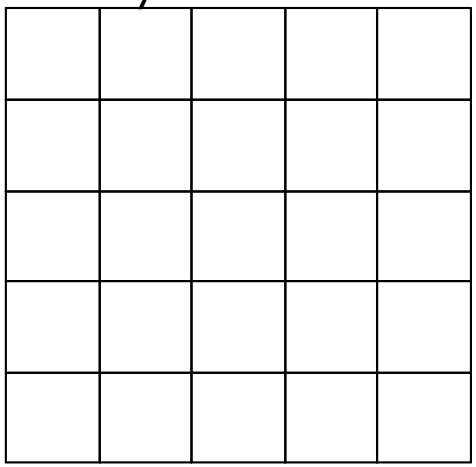
- We control all the channels of communication
- Communicate the current values of boxes explicitly

Boxes and Memory

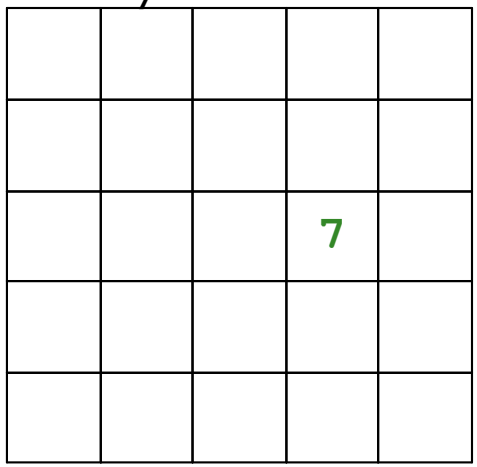
```
{let {[b {box 7}]}
  ...}
```

⇒ `...`

Memory:



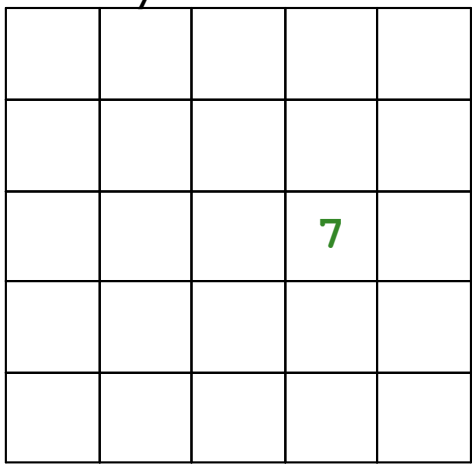
Memory:



Boxes and Memory

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

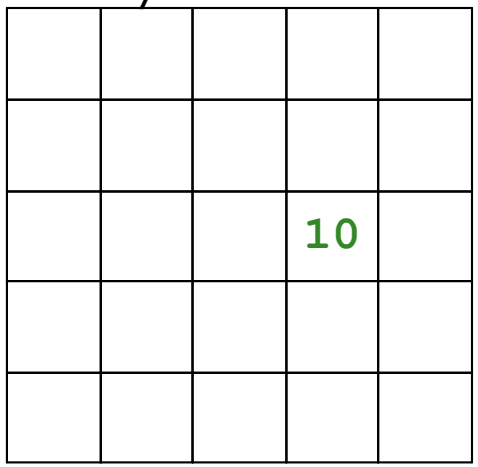
Memory:



⇒

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

Memory:

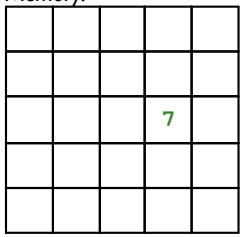


Communicating Memory

```
(interp ....  ) ⇒ ...
```

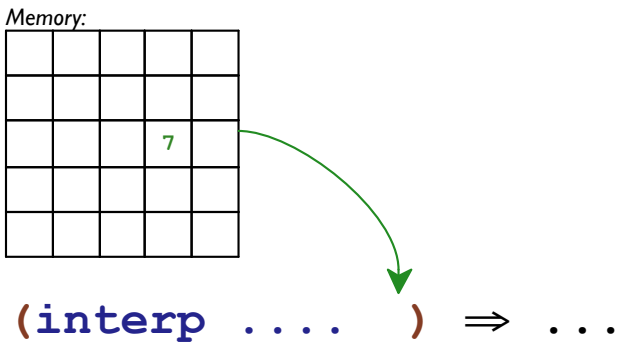
Communicating Memory

Memory:

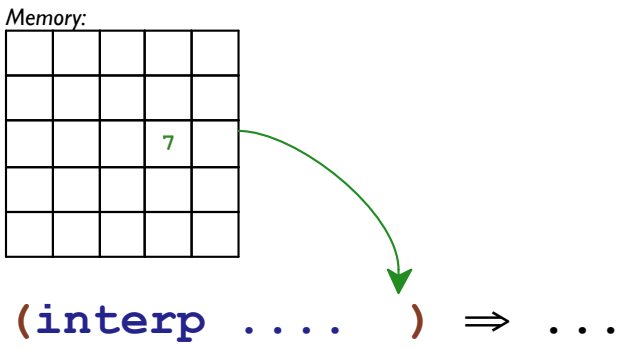



```
(interp ....  ) ⇒ ...
```

Communicating Memory

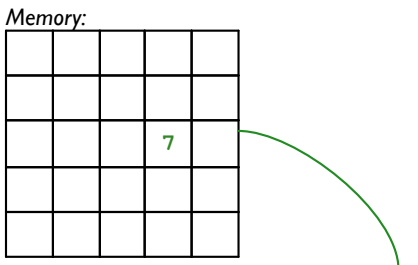


Communicating Memory




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

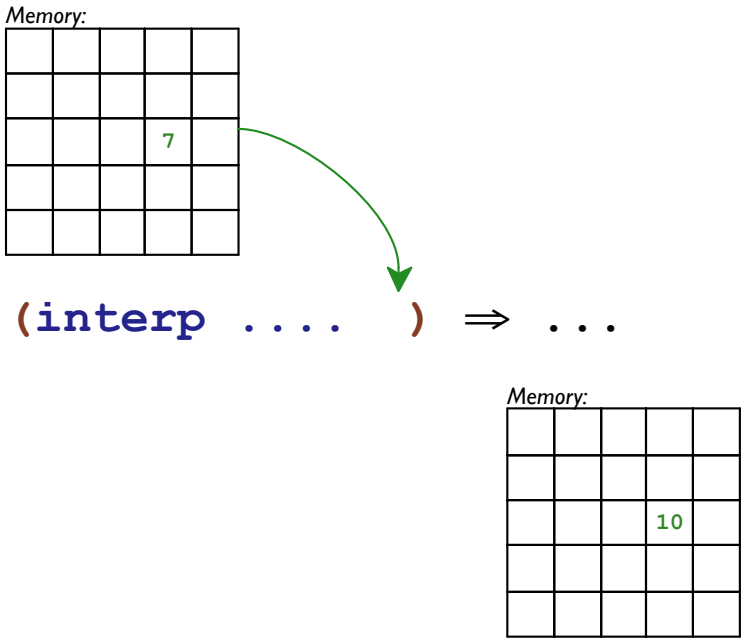
Communicating Memory




```
(interp ....  ) ⇒ ...
```

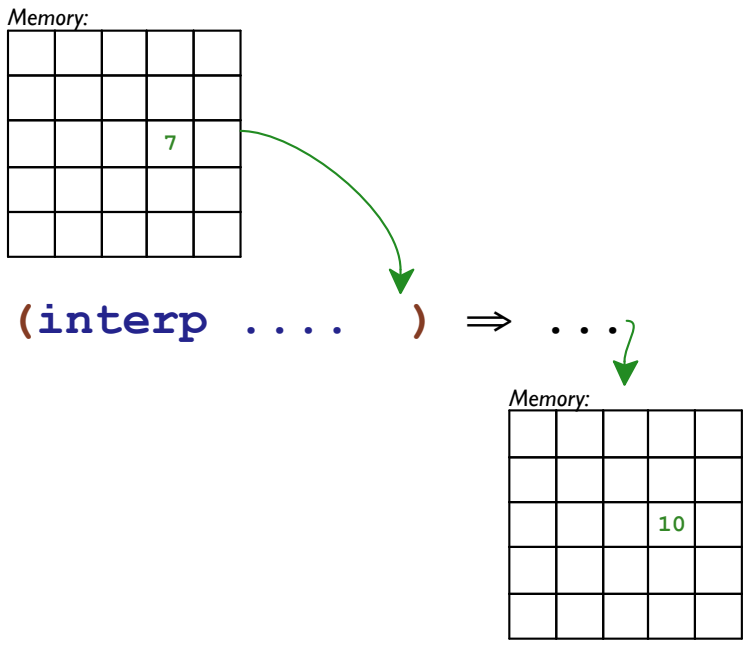
```
interp : (Expr Env Store -> Value)
```

Communicating Memory



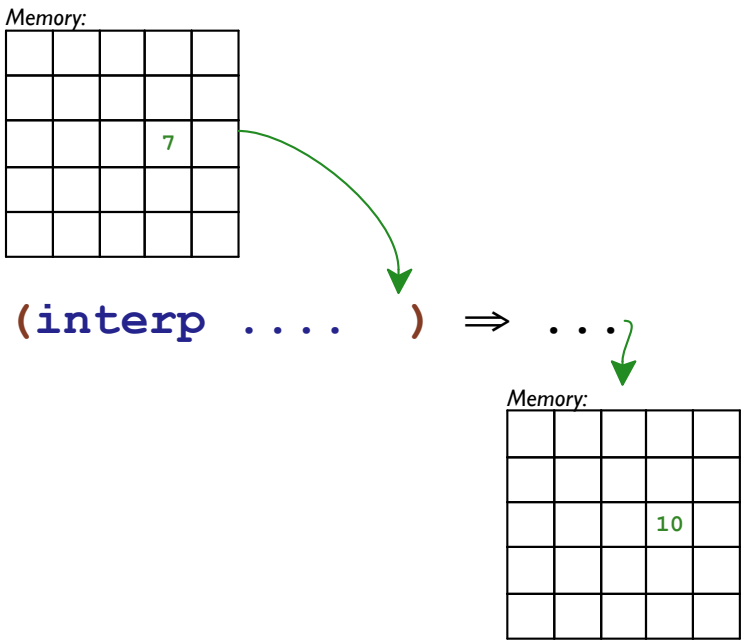

```
interp : (Expr Env Store -> Value)
```

Communicating Memory



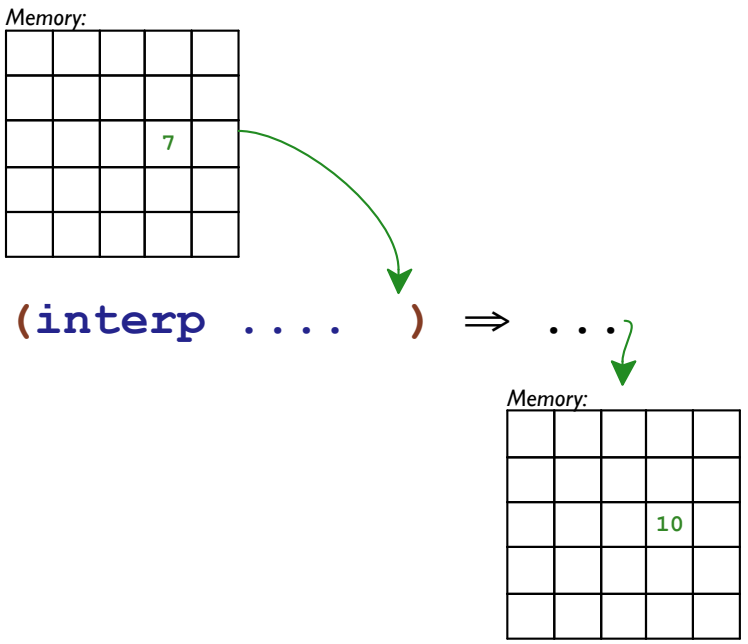

```
interp : (Expr Env Store -> Value)
```

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))
```

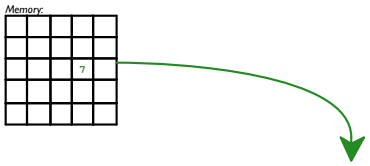
Communicating the Store

```
(interp l env)
```

```
(interp r env)
```

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

Communicating the Store




```
(interp l env)
```

```
(interp r env)
```

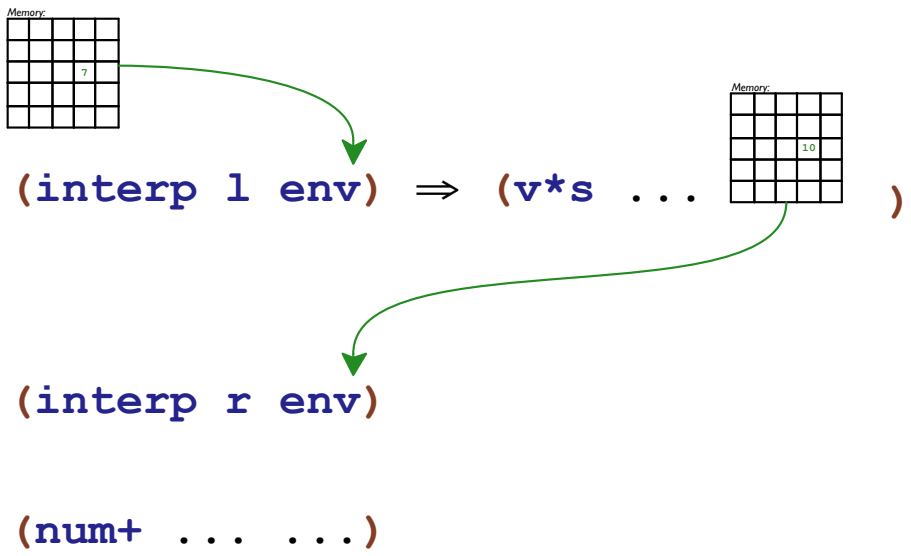
```
(num+ ... ...)
```

Communicating the Store

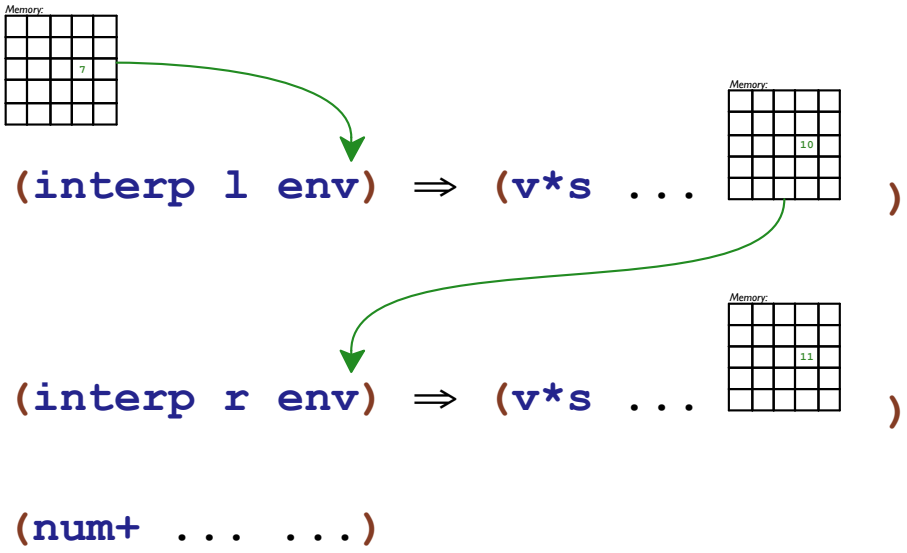

```
(interp r env)
```

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

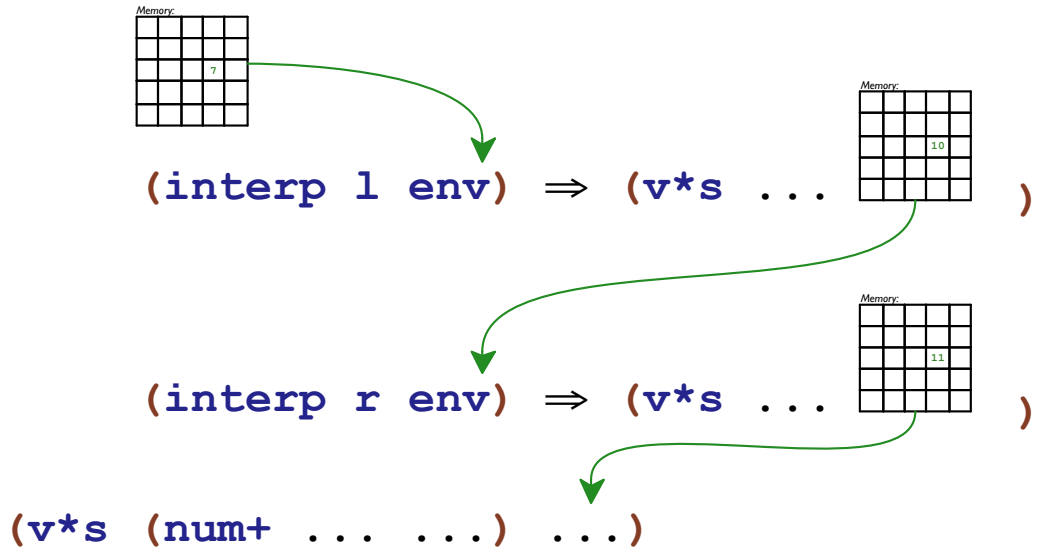
Communicating the Store



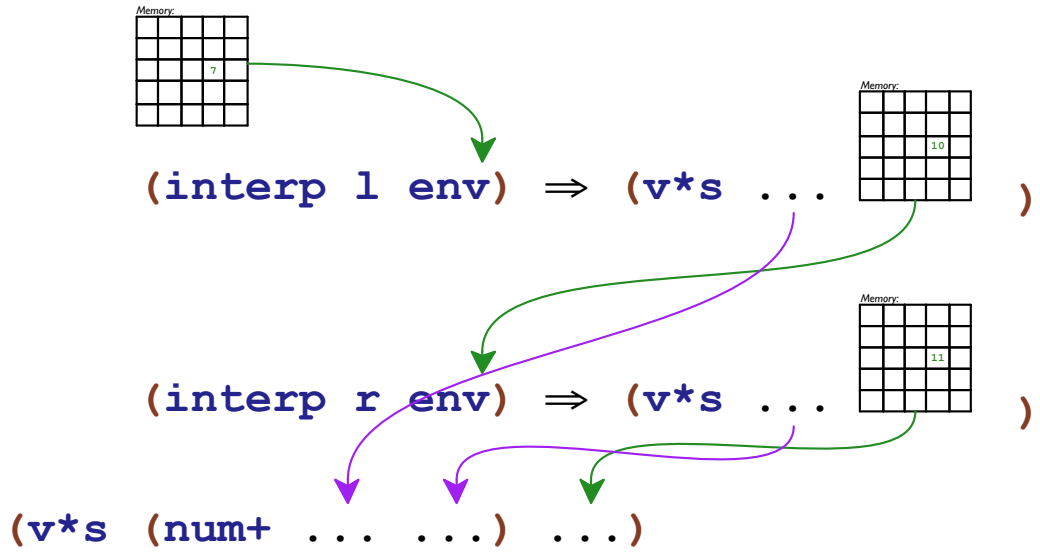
Communicating the Store



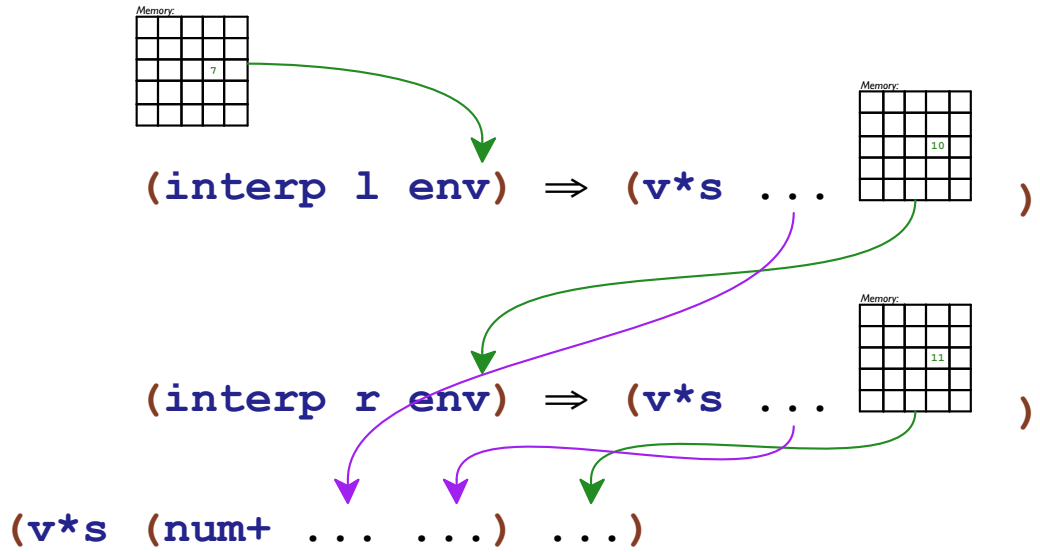
Communicating the Store



Communicating the Store



Communicating the Store

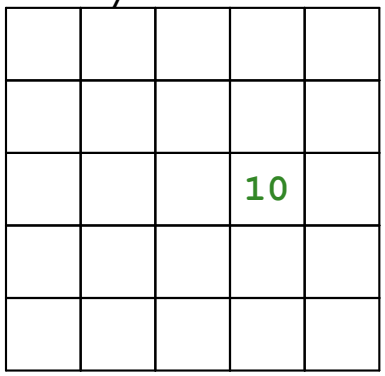



```
(type-case Result (interp l env sto)
  [(v*s v-l sto-l)
   (type-case Result (interp r env sto-l)
     [(v*s v-r sto-r)
      (v*s (num+ v-l v-r) sto-r)])])
```

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 : (Exp Env -> Value)

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

Store Examples

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

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

Store Examples

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

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

Store Examples

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

(test (interp (boxE (numE 5)) mt-env mt-store)
      ...)
```

Store Examples

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

(test (interp (boxE (numE 5)) mt-env mt-store)
      (v*s ...
           ...))
```

Store Examples

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

(test (interp (boxE (numE 5)) mt-env mt-store)
      (v*s (boxV ...)
           ...))
```

Store Examples

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

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

Store Examples

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

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

Store Examples

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

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

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

Store Examples

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

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

Store Examples

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

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

Store Examples

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

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

Store Examples

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

`interp` with a Store

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

`interp` with a Store

```
(define interp : (Exp Env Store -> Result)
  (lambda (a env sto)
    ...
    [(plusE 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 : (Exp Env Store -> Result)
  (lambda (a env sto)
    ...
    [(boxE 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
  (type-case (Listof Storage) sto
    [empty 0]
    [(cons c rst-sto) (max (cell-location c)
                           (max-address rst-sto))]))
```

`interp` with a Store

```
(define interp : (Exp Env Store -> Result)
  (lambda (a env sto)
    ...
    [(unboxE 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 : (Exp Env Store -> Result)
  (lambda (a env sto)
    ...
    [(setboxE 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 : (Exp Env Store -> Result)
  (lambda (a env sto)
    ...
    [(beginE 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 v-r 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)))
```