

Louis XIV

AKA: State (Part II)

Obscurity rating: 7/10

Boxes and Memory

`{with {b {newbox 7}}}` $\Rightarrow$ `...`
`...}`

Memory:

Memory:

			7	

Boxes and Memory

... {setbox b 10}

...

⇒

...

...

Memory:

			7	

Memory:

			10	

The Store

We represent memory with a **store**:

```
(define-type Store
  [mtSto]
  [aSto (address integer?)
        (value BFAE-Value?)
        (rest Store?)])
```

Memory:

			10	

```
(aSto 13
      (numV 10)
      (mtSto))
```

Implementing Boxes without State

```
; interp : BFAE? DefSub? Store? -> Value*Store?
```

```
(define-type BFAE-Value  
  [numV (n number?)]  
  [closureV (param-name symbol?)  
             (body BFAE?)  
             (ds DefSub?)]  
  [boxV (address integer?)])
```

```
(define-type Value*Store  
  [v*s (value BFAE-Value?)  
       (store Store?)])
```

Implementing Boxes without State

```
; interp : BFAE? DefSub? Store? -> Value*Store?
(define (interp a-bfae ds st)
  ...
  [newbox (val-expr)
    (type-case Value*Store (interp val-expr ds st)
      [v*s (val st)
        (define a (malloc st))
        (v*s (boxV a)
              (aSto a val st))]]])
  ...)

; malloc : Store? -> integer?
```

Implementing Boxes without State

```
; malloc : Store? -> integer?  
(define (malloc st)  
  (+ 1 (max-address st)))
```

```
; max-address : Store? -> integer?  
(define (max-address st)  
  (type-case Store st  
    [mtSto () 0]  
    [aSto (n v st)  
          (max n (max-address st))]))
```

Implementing Boxes without State

```
; interp : BFAE? DefSub? Store? -> Value*Store?
(define (interp a-bfae ds st)
  ...
  [openbox (box-expr)
    (type-case Value*Store (interp box-expr ds st)
      [v*s (box-val st)
        (v*s (store-lookup (boxV-address box-val)
                           st)
              st)]]])
  ...)
```


Implementing Boxes without State

```
; interp : BFAE? DefSub? Store? -> Value*Store?
(define (interp a-bfae ds st)
  ...
  [setbox (box-expr val-expr)
    ... (interp box-expr ds st) ...
    ... (interp val-expr ds st)
    ...]
  ...)
```

```
{with {b {newbox 10}}
  {setbox {seqn {setbox b 12} b}
    {openbox b}}}
```

should put 12 in b, not 10

Implementing Boxes without State

```
; interp : BFAE? DefSub? Store? -> Value*Store?
(define (interp a-bfae ds st)
  ...
  [setbox (box-expr val-expr)
    (type-case Value*Store (interp box-expr ds st)
      [v*s (box-val st2)
        (type-case Value*Store (interp val-expr ds st2)
          [v*s (val st3)
            (v*s val
              (aSto (boxV-address box-val)
                    val
                    st3))])])])
  ...)
```

seqn, **add**, **sub**, and **app** will need the same sort of sequencing

Implementing Boxes without State

```
; interp-two : (BFAE? BFAE? DefSub? Store?
;             (Value? Value? Store? -> Value*Store?)
;             -> Value*Store?)
(define (interp-two a-bfae1 a-bfae2 ds st finish)
  (type-case Value*Store (interp a-bfae1 ds st)
    [v*s (val1 st2)
      (type-case Value*Store (interp a-bfae2 ds st2)
        [v*s (val2 st3)
          (finish val1 val2 st3)]]))])
```

finish $\approx$ callbacks in JavaScript!

A function we pass in that will do "the rest of the work"
after we're done with our part

We'll explore that in much more depth next time

Implementing Boxes without State

```
; interp : BFAE? DefSub? Store? -> Value*Store?
(define (interp a-bfae ds st)
  ...
  [add (r l) (interp-two r l ds st
                        (lambda (v1 v2 st)
                          (v*s (num+ v1 v2) st)))]
  ...
  [seqn (a b) (interp-two a b ds st
                          (lambda (v1 v2 st)
                            (v*s v2 st)))]
  ...
  [setbox (box-expr val-expr)
          (interp-two box-expr val-expr ds st
                      (lambda (box-val val st3)
                        (v*s val
                            (aSto (boxV-address box-val)
                                val
                                st3)))))]
  ...)
...)
```