State

Functional Programs

So far, our object languages have been purely ***functional***

- A function produces the same result every time for the same arguments
- That’s nice in some ways
- But that’s kind of limiting
- Sometimes we just need to keep track of changes

Non-Functional Procedure

```
(define counter 0)
(define (f x)
  (set! counter (+ x counter))
  counter)
```

- Using mutable variables to keep track of *state*

Non-Functional Procedure, now with boxes!

```
(define counter (box 0))
(define (f x)
  (set-box! counter (+ x (unbox counter)))
  (unbox counter))
```

- Alternatively, can use mutable data structures
- Box ≈ single-element mutable array

BFAE = FAE + Boxes

```
<BFAE> ::= <num>
        |  {+ <BFAE> <BFAE>}
        |  {- <BFAE> <BFAE>}
        |  <id>
        |  {fun {<id>} <BFAE>}
        |  {<BFAE> <BFAE>}
        |  {newbox <BFAE>}                NEW
        |  {setbox <BFAE> <BFAE>}         NEW
        |  {openbox <BFAE>}               NEW
        |  {seqn <BFAE> <BFAE>}           NEW
```

```
{with {b {newbox 0}}
  {seqn
   {setbox b 10}
   {openbox b}}}        ⇒  10
```

Implementing Boxes with Boxes

```
(define-type BFAE-Value
  [numV (n number?)]
  [closureV (param-name symbol?)
            (body BFAE?)
            (ds DefSub?)]
  [boxV (container (box/c BFAE-Value?))])
```

Implementing Boxes with Boxes

```
; interp : BFAE? DefSub? -> BFAE-Value?
(define (interp a-bfae ds)
  (type-case BFAE a-bfae
    ...
    [newbox (val-expr)
            (boxV (box (interp val-expr ds)))]
    [setbox (box-expr val-expr)
            (set-box! (boxV-container
                       (interp box-expr ds))
                      (interp val-expr ds))]
    [openbox (box-expr)
             (unbox (boxV-container
                     (interp box-expr ds)))]))
```

Nice parlor trick.
But we haven’t learned anything about how boxes work!