State

Purely 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: much easier to test and debug!
- But that can be limiting: some problems naturally need to keep track of changes over time
    - Can still always be expressed purely functionally!
    - But can get really awkward...

Purely Functional Graph Search

Graph search to check for path between two nodes. A 214 classic.

Important bit: need to keep track of the nodes we've seen.

```
(define (search graph src dest seen)
  (cond
    [(= src dest)       (cons #t seen)] ; accumulator goes up
    [(member src seen) (cons #f seen)]
    [else
     (foldl (lambda (neighbor acc) ; accumulator goes sideways
              (if (car acc)
                  acc
                  (search graph neighbor dest
                          (cdr acc)))) ; accumulator goes down
            (cons #f (cons src seen))
            (neighbors graph src))]))
```

- Need to carry the **seen** set both up **and** down the call tree
- Means returning two values: the actual result, and the seen set
- This is really awkward

Graph Search, with State

Boom. Much better.

No more funny accumulator plumbing.

Only ever need to return the actual result.

```
(define (search graph src dest)
  (define seen '()) ; mutable seen set
  (define (helper cur)
    (cond [(= cur dest)       #t]
          [(member cur seen) #f]
          ; imperatively update state
          [else (set! seen
                      (cons cur seen))
                (ormap helper
                       (neighbors graph cur))]))
  (helper src))
```

Graph Search, with Boxes

Boxes: simplest form of state, stores one value that can change.

Think of them as a mutable, single-element array.

Can initialize with contents, change contents, read contents.

```
(define (search graph src dest)
  (define seen (box '())) ; initialize
  (define (helper cur)
    (cond [(= cur dest)                #t]
          [(member cur (unbox seen)) #f] ; read
          ; write
          [else (set-box! seen
                          (cons cur (unbox seen)))
                (ormap helper
                       (neighbors graph cur))]))
  (helper src))
```

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!