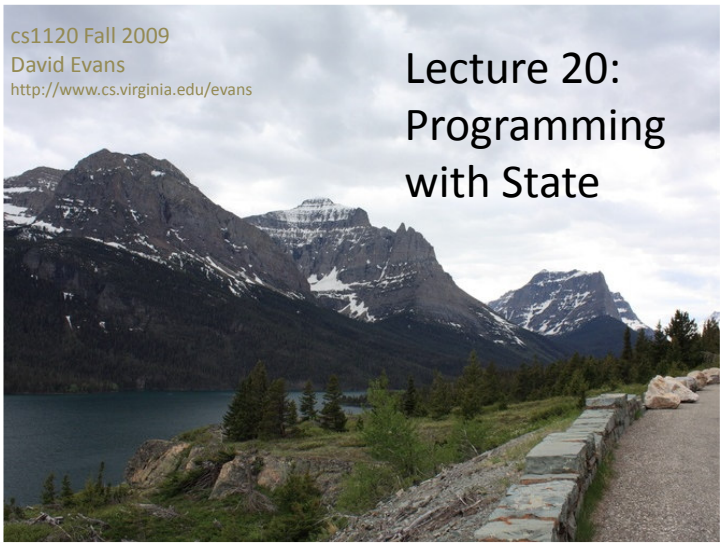
cs1120 Fall 2009
David Evans
http://www.cs.virginia.edu/evans

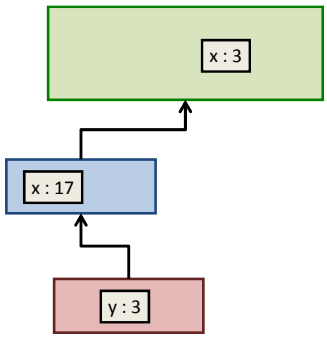
Lecture 20: Programming with State



From Friday's class:

What would create this environment?

```
> (define x 3)
> ((lambda (x) … ) 17)
```



Stateful Application Rule

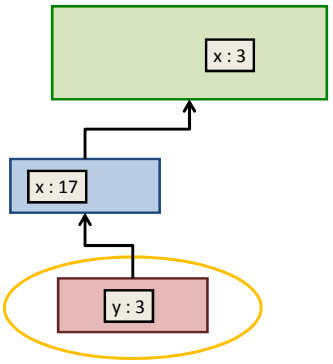
To apply a constructed procedure:

1. **Construct a new environment**, whose parent is the environment of the applied procedure.
2. **For each procedure parameter, create a place** in the frame of the new environment with the name of the parameter. Evaluate each operand expression in the environment or the application and initialize the value in each place to the value of the corresponding operand expression.
3. **Evaluate** the body of the procedure **in the newly created environment.** The resulting value is the value of the application.

To apply a constructed procedure:

1. **Construct a new environment**, whose parent is the environment of the applied procedure.
2. **For each procedure parameter, create a place** in the frame of the new environment with the name of the parameter. Evaluate each operand expression in the environment or the application and initialize the value in each place to the value of the corresponding operand expression.
3. **Evaluate** the body of the procedure **in the newly created environment.** The resulting value is the value of the application.

To create this environment, a procedure whose environment is the Blue environment was applied.




```
> (define x 3)
> ((lambda (x)
     ((lambda (y) x) 3)
  17)
```


Programming Styles

- Functional Programming
  - Program by composing functions
  - Substitution model applies
  - Problem Sets 1-4
- Imperative Programming
  - Programming by changing state
  - Requires stateful model of evaluation
  - Problem Set 5 and beyond

Most programs **combine aspects of both styles**: even imperative-style programs still involve composing procedures, and very few programs are completely functional.

Mapping

Functional Solution: A procedure that takes a procedure of one argument and a list, and returns a list of the results produced by applying the procedure to each element in the list.

```
(define (list-map f p)
  (if (null? p) null
     (cons (f (car p))
           (list-map f (cdr p)))))
```

Imperative Solution

```
(define (list-map f p)
  (if (null? p) null
      (cons (f (car p))
            (list-map f (cdr p)))))
```

A procedure that takes a procedure and a mutable list as arguments, and ***replaces*** each element in the list with the value of the procedure applied to that element. It produces no output.

```
(define (mlist-map! f p)
  (if (null? p) (void)
      (begin
        (set-mcar! p (f (mcar p)))
        (mlist-map! f (mcdr p)))))
```

Programming with Mutation

Imperative

```
> (mlist-map! square (mlist 1 2 3 4))
> (define i4 (mlist 1 2 3 4))
> (mlist-map! square i4)
> i4
(1 4 9 16)
```

Functional

```
> (define i4 (intsto 4))
> (map square i4)
(1 4 9 16)
> i4
(1 2 3 4)
```

Comparing Cost

Functional

```
(define (list-map f p)
  (if (null? p) null
      (cons (f (car p))
            (list-map f (cdr p)))))
```

Assuming *f* has constant running time, running time is in **θ(*N*)** where *N* is the number of elements in *p*.

Memory use is in **θ(*N*)** where *N* is the number of elements: it requires construction *N* new cons cells.

Imperative

```
(define (mlist-map! f p)
  (if (null? p) (void)
      (begin
        (set-mcar! p (f (mcar p)))
        (mlist-map! f (mcdr p)))))
```

Also has running time in **θ(*N*)**:
N recursive calls,
constant work each time.

Memory use is in **O(1)** : no new cons cells are created!
(Aside: because it is tail recursive, no evaluation stack either.)

Appending

```
(define (list-append p q)
  (if (null? p) q
      (cons (car p)
            (list-append (cdr p) q))))
```

Running time in $\theta(N_p)$, N_p is number of elements in *p*
Number of new cons cells: $\theta(N_p)$

```
(define (mlist-append! p q)
  (if (null? p) (error "Cannot append to empty list!")
      (if (null? (mcdr p))
          (set-mcdr! p q)
          (mlist-append! (mcdr p) q))))
```

Running time in $\theta(N_p)$, number of elements in *p*
Number of new cons cells: 0

Does it matter?

```
> (define r1 (random-list 100000))
> (define r2 (random-list 100000))
> (time (begin (list-append r1 r2) true))
cpu time: 110 real time: 122 gc time: 78
#t
> (define m1 (random-mlist 100000))
> (define m2 (random-mlist 100000))
> (time (begin (mlist-append! m1 m2) true))
cpu time: 15 real time: 22 gc time: 0
#t
> (mlength m1)
200000
> (time (begin (mlist-append! m1 m2) true))
cpu time: 47 real time: 45 gc time: 0
#t
> (mlength m1)
```

```
(define (random-list n)
  (if (= n 0) null
      (cons (random 1000) (random-list (- n 1)))))
```

```
(define (random-mlist n)
  (if (= n 0) null
      (mcons (random 1000) (random-mlist (- n 1)))))
```


Charge

- Reading (finish by next Monday): *Science's Endless Golden Age* by Neil DeGrasse Tyson
- PS5 due one week from today