Announcements

- Homework 2 due Wednesday (start of class)
  - in scheme
  - can have additional office hours Tuesday evening in ARC lab (send me email if you need this)

Scheme

- Conditionals
- Examples of recursive functions
  - shallow vs. deep
- Equality testing
- Higher Order Functions
  - apply and eval
  - map, foldleft, foldright

Conditional Execution

```
(if <test> <consequent> <alternative>)
```

- evaluate `<test>`
- if result is a "true value" (i.e., anything but `#f`), then evaluate and return `<consequent>`
- otherwise, evaluate and return `<alternative>`

```
(cond (<test1> <expr1>)
      (<test2> <expr2>)
       ...
     (else <expr-else>))
```

- evaluates `<test1>`, `<test2>`, `...` in order, and returns corresponding expression for the first test that is true
- if all are false, the expression in else clause is returned

Trace of Evaluation

(define (atom? object) (not (pair? object)))

(atom? '(a))

-obtain function value corresponding to *atom?*

-evaluate '(a) obtaining (a)

-evaluate (not (pair? object))

  -obtain function value corresponding to *not*

  -evaluate (pair? object)

    -obtain function value corresponding to *pair?*

    -evaluate object obtaining (a)

    -return value #*t*

  -return #*f*

-return #*f*

Recursive Functions

(define (len x) (cond ((null? x) 0) (else (+ 1 (len (cdr x))))))

(len ‘(1 2)) should yield 2.

len is a shallow recursive function

Trace: (len ‘(1 2)) --top level call

```
          x = (1  2)
                    (len ‘(2)) --recursive call 1
                    x = (2)
                              (len ‘( ) ) -- recursive call 2
                              x = ( )
                              returns 0 --return for call 2
                    returns (+  1  0) =1 --return for call 1
          returns (+ 1  1) = 2 --return for top level call
```

(len ‘((a) b (c d))) returns 3

5

List Append

```
(define (app  x  y)
   (cond  ((null?  x)  y)
            ((null?  y)  x)
            (else  (cons (car x)  (app (cdr x)  y)))))
```

(app ‘( ) ‘( )) yields ( )

(app ‘( ) ‘(1 4 5)) yields (1 4 5)

(app ‘(5 9) ‘(a (4) 6)) yields (5 9 a (4) 6)

This is another shallow recursive function

6

Atomcount Function

atomcount is a deep recursive function

```
(define  (atomcount  x)
   (cond  ((null?  x)  0)
             ((atom? x)  1)
             (else  (+ (atomcount  (car x))  (atomcount  (cdr x)))) ))
```

(atomcount ‘(1)) yields 1

(atomcount ‘(1 (2 (3)) (5)))yields 4

Trace: (atomcount ‘(1 (2 (3)))

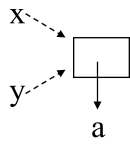
```
1> (+   (atomcount 1) (atomcount ‘( (2 (3)) ) ))
   2> (+ (atomcount ‘(2 (3)) ) (atomcount ‘( ) ) )
        3> (+  (atomcount 2) (atomcount ‘((3)) ) etc.
                4> (+ (atomcount ‘(3)) (atomcount ‘( )) )
   1                  5> (+ (atomcount 3) (atomcount ‘( )))
```


Equality

```
> (define (f x y) (list x y))
> (f ′a ′a)
```

What happens?

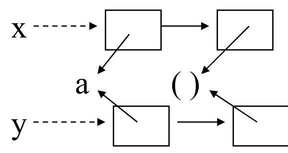
the atom: a




```
> (f ′(a) ′(a))
```

the arguments: (a)

What happens?



Equality

eq?

- checks atoms for equal values
- but doesn't work correctly on lists

eql?

- equality function for lists

```
(define (eql? x y)
  (or (and (atom? x) (atom? y) (eq? x y))
      (and (pair? x) (pair? y)
           (eql? (car x) (car y))
           (eql? (cdr x) (cdr y)))))
```


Equality

```
(eql? '(a) '(a))     → #t
(eql? 'a 'b)         → #f
(eql? 'b 'b)         → #t
(eql? '((a)) '(a))   → #f

(eq? 'a 'a)          → #t
(eq? '(a) '(a))      → #f
```

Functions: First-Class Objects

Functions as arguments:

```
(define (f g x) (g (car x)))
(f number? '(0 a)) → #t
(f length '((2 3) (4))) → 2
(f (lambda (x) (* 2 x)) '(3)) → 6
```

Functions as return values:

```
(define (incr) (lambda (n) (+ 1 n)))
((incr) 4) → 5
```


Higher Order Functions

(apply <function> <arguments>)

– **apply <function> to list of <arguments>**

```
(apply + '(1 2 3)) → 6
(apply zero? '(2)) → #f
(apply (lambda (n) (+ 1 n)) '(3)) → 4
```

(eval <expression>)

– **evaluate <expression> as a function application**

```
(eval '(+ 3 4)) → 7
(eval (list '+ 3 4)) → 7
```