# CSC 270
# Nov. 22, 2005
# Last Day of Scheme

**Dr. Stephen Bloch**
**sbloch@adelphi.edu**
**http://www.adelphi.edu/sbloch/class/270/**

# Review

- A "list" is either empty or not.
- If it's empty, it contains no information.
- If it's non-empty, it has a "first" element (some data type) and a "rest" (another list).

# Review: built-in functions on lists

- empty? : object -> boolean
- cons? : object -> boolean
- cons : object list -> non-empty-list
- first : non-empty-list -> object
- rest : non-empty-list -> list

# Review: building lists

- empty
- (cons "a" empty)
- (cons "b" (cons "a" empty))
- (define mylist (cons "b" (cons "a" empty)))
- (cons "e" (cons "d" (cons "c" mylist)))

# Review: examining lists

- (first (cons "a" empty)) "should be a"
- (rest (cons "a" empty)) "should be empty"
- (first (rest (rest (cons "a" (cons "b" (cons "c" (cons "d" empty))))))) "should be c"

# Review: writing functions on lists

```
; how-long : list -> number
(define (how-long L)
   (cond [(empty? L) 0]
         [(cons? L) (+ 1 (how-long (rest L)))]))
"Examples of how-long:"
(how-long empty) "should be 0"
(how-long (cons 73 empty)) "should be 1"
(how-long (cons "a" (cons "b" empty)))
   "should be 2"
```

# Exercise: writing a function on lists

```
; add-up : list-of-numbers -> number
(define (add-up nums)
   (cond [(empty? nums) 0]
            [(cons? nums)
             (+ (first nums) (add-up (rest nums)))]))
"Examples of add-up:"
(add-up empty) "should be 0"
(add-up (cons 4 empty)) "should be 4"
(add-up (cons 3 (cons 2 empty))) "should be 5"
```

# Shorter notation

Note: **change languages to Beginning Student with List Abbreviations** (or higher)

; list : as many objects as you wish -> list

(list "a" "b" "c") is short for (cons "a" (cons "b" (cons "c" empty)))

(list) is equivalent to empty

The functions **empty?**, **cons?**, **cons**, **first**, **rest**, **how-long**, and **add-up** work *exactly as before*; this is just a shorter way of *displaying* a list

**Warning:**

(list "a" empty) is *not* the same thing as (cons "a" empty)!

# Another function on lists

```
; remove>10 : list-of-nums -> list-of-nums
(define (remove>10 nums)
  (cond [(empty? nums) empty]
        [(cons? nums)
         (cond [(> (first nums) 10)   (remove>10 (rest nums))]
               [else       (cons (first nums)   (remove>10 (rest nums)))])]))

"Examples of remove>10:"
(remove>10 empty) "should be" empty
(remove>10 (list 6)) "should be" (list 6)
(remove>10 (list 11)) "should be" empty
(remove>10 (list 6 11 10 -24 13 9)) "should be" (list 6 10 -24 9)
(remove>10 (list 11 10 -24 13 9)) "should be" (list 10 -24 9)
```

# Generalizing the function

; remove>5 : list-of-nums -> list-of-nums

; remove>17: list-of-nums -> list-of-nums

What these have in common is that they **remove all elements of the list greater than a fixed threshold**.

So we **generalize** the function:

```
; remove-over: num list-of-nums -> list-of-nums
(define (remove-over threshold nums)
  (cond [(empty? nums) empty]
        [(cons? nums)
         (cond [(> (first nums)threshold)   (remove-over threshold (rest nums))]
               [else   (cons (first nums) (remove-over threshold (rest nums)))])]))
```

"Examples of remove-over:"

(remove-over 6 empty) "should be" empty

…

(remove-over 3.5 (list 4 9 17 2 6 3)) "should be" (list 2 3)

# Generalizing the function *farther*

; remove<5 : list-of-nums -> list-of-nums

; remove>=4: list-of-nums -> list-of-nums

; remove-evens : list-of-nums -> list-of-nums

What all of these have in common is that they **perform a test on each element of the list, and remove the ones that pass the test.**

Generalization:

; remove-if : test list-of-nums -> list-of-nums

Q: What is a "test"?

A: a property that every number either has or doesn't have

A: a function from number to boolean

Note: **change languages to Intermediate Student**

# Writing **remove-if**

```
; remove-if : (num -> boolean) list-of-nums -> list-of-nums
(define (remove-if test? nums)
  …

   )
"Examples of remove-if:"
(remove-if even? (list 1 2 3 4 5)) "should be" (list 1 3 5)
(define (over-10? x) (> x 10))
(remove-if over-10? (list 3 12 10 5 16 -24 6)) "should be" (list 3 10 5 -24 6)
(define (under-5? x) (< x 5))
(remove-if under-5? (list 3 12 10 5 16 -24 6)) "should be" (list 12 10 5 16 6)
```

# The routine stuff

```
; remove-if : (num -> boolean) list-of-nums -> list-of-nums
(define (remove-if test? nums)
  (cond [(empty? nums) empty]
        [(cons? nums)
         (cond [...
                 (remove-if test? (rest nums))]
               [else
                (cons (first nums) (remove-if test? (rest nums)))])]))
"Examples of remove-if:"
(remove-if even? (list 1 2 3 4 5)) "should be" (list 1 3 5)
(define (over-10? x) (> x 10))
(remove-if over-10? (list 3 12 10 5 16 -24 6)) "should be" (list 3 10 5 -24 6)
(define (under-5? x) (< x 5))
(remove-if under-5? (list 3 12 10 5 16 -24 6)) "should be" (list 12 10 5 16 6)
```

# Using the test

```
; remove-if : (num -> boolean) list-of-nums -> list-of-nums
(define (remove-if test? nums)
  (cond [(empty? nums) empty]
        [(cons? nums)
         (cond [(test? (first nums))
                (remove-if test? (rest nums))]
               [else
                (cons (first nums) (remove-if test? (rest nums)))])]))
"Examples of remove-if:"
(remove-if even? (list 1 2 3 4 5)) "should be" (list 1 3 5)
(define (over-10? x) (> x 10))
(remove-if over-10? (list 3 12 10 5 16 -24 6)) "should be" (list 3 10 5 -24 6)
(define (under-5? x) (< x 5))
(remove-if under-5? (list 3 12 10 5 16 -24 6)) "should be" (list 12 10 5 16 6)
```

# Writing functions using remove-if

```
; remove<5 : list-of-nums -> list-of-nums
(define (under-5? x) (< x 5))
(define (remove<5 nums) (remove-if under-5? nums))
```

```
; remove>=7: list-of-nums -> list-of-nums
```

**You try this one.**

```
; remove-evens : list-of-nums -> list-of-nums
(define (remove-evens nums) (remove-if even? nums))
```

# Another example

```
; cube-each : list-of-nums -> list-of-nums
(define (cube-each nums)
  (cond [(empty? nums) empty]
        [(cons? nums)
         (cons (cube (first nums))
               (cube-each (rest nums)))]))

"Examples of cube-each:"
(cube-each empty) "should be" empty
(cube-each (list 2)) "should be" (list 8)
(cube-each (list 3 -2 0 5 -6)) "should be"
  (list 27 -8 0 125 -216)
```

# Similar functions

; sqrt-each : list-of-nums -> list-of-nums

; negate-each : list-of-nums -> list-of-nums

What these have in common is that they **do the same thing to each element of a list, returning a list of the results.**

So we **generalize** the functions:

; do-to-each : operation list-of-nums -> list-of-nums

What's an "operation"? In this case, a function from number to number.

; do-to-each : (num -> num) list-of-nums -> list-of-nums

# Writing do-to-each

```
; do-to-each : (num -> num) list-of-nums -> list-of-nums
(define (do-to-each op nums)
    (cond [(empty? nums) empty]
          [(cons? nums)
           (cons (op (first nums))
                 (do-to-each op (rest nums)))]))
"Examples of do-to-each:"
(do-to-each cube (list 3 5 -2)) "should be" (list 27 125 -8)
(do-to-each sqrt (list 4 25 0)) "should be" (list 2 5 0)
(do-to-each - (list 3 -2 0 7.5)) "should be" (list -3 2 0 -7.5)
```

# Writing functions using do-to-each

```
; sqrt-each : list-of-nums -> list-of-nums
(define (sqrt-each nums)
   (do-to-each sqrt nums))

; add-3-to-each : list-of-nums -> list-of-nums
(define (add3 x) (+ x 3))
(define (add-3-to-each nums)
   (do-to-each add3 nums))
```

# Dumb single-use functions

```
; add-3-to-each : list-of-nums -> list-of-nums
(define (add3 x) (+ x 3))
(define (add-3-to-each nums) (do-to-each add3 nums))
```

Better: hide **add3** inside a local definition

```
(define (add-3-to-each nums)
  (local [(define (add3 x) (+ x 3))]
    (do-to-each add3 nums)))
```

# An example where we *have* to use **local**

```
; remove-over : num list-of-nums -> list-of-nums
(define (remove-over threshold nums)
  (local [(define (over-threshold? num)
              (> num threshold))]
    (remove-if over-threshold? nums)))
```

Note: we *couldn't* have defined **over-threshold?** outside **remove-over**, because it would have depended on the threshold value.

# Defining functions without names

(+ 3 (* 4 5))

doesn't require defining a variable to hold the value of (* 4 5), and then adding 3 to it!

Why should **add-3-to-each** require defining a function to add 3 to things, and then applying **do-to-each** to it?

Note: **change languages to Intermediate Student with Lambda**

# Defining functions without names

New syntax rule:

**(lambda (param param …) expr)**

constructs a function without a name and returns it.

Example:

```
(define (add-3-to-each nums)
  (do-to-each (lambda (x) (+ x 3)) nums))
```

# Defining functions without names

- Anything you can do with **lambda** can also be done with **local**; may be more readable because things have names
- Anything you can do with **local** can also be done with **lambda**, often a little shorter

# Programs that interact with user

- Our Scheme programs so far are *called* with input, and they *return* an answer.
- Many real-world programs have to hold a *continuing dialogue* with user:
  - user says something
  - program responds
  - user responds to this
  - program responds to that
  - etc.

# Programs that interact with user

- Other programs need to produce output *piece by piece*
- (list-primes)
  - 2
  - 3
  - 5
  - 7
  - 11
  - 13
  - user break

# Text input & output
# (in Advanced Student language)

```
; display : object -> nothing, but prints the
   object on the screen
(display 3)
(display (+ 3 4))
(display "hello there")
(display 'blue)
(display (make-posn 3 4))
```

# Text input & output

; display-with-label : string obj -> nothing, but prints the string and the object

```
"Examples of display-with-label:"
(define my-age 40)
(display-with-label "Age:" my-age)
"should print Age: 40"
```

# Text input & output

```
; display-with-label : string obj -> nothing, but prints the
    string and the object
(define (display-with-label label obj)

    (display label)
    (display obj))                    <--- problem! 2 expressions!
"Examples of display-with-label:"
(define my-age 40)
(display-with-label "Age:" my-age)
"should print Age: 40"
```

# Text input & output

```
; display-with-label : string obj -> nothing, but prints the
    string and the object
(define (display-with-label label obj)
  (begin
     (display label)
     (display obj)))
"Examples of display-with-label:"
(define my-age 40)
(display-with-label "Age:" my-age)
"should print Age: 40"
```

# Sequential programming

; begin : expr expr expr … -> object

; Evaluates each expression, ignoring the results, but returns the result of the last one.

```
(begin
  (display (+ 3 4))
  (* 5 6))
```

"should display 7 and then return 30"

; Note: if last expression returns nothing (e.g. display), so does begin.

# Also want to get input from user

; read : nothing -> object

; waits for user to type an expression, and returns it

Try some examples: numbers, strings, booleans, identifiers

# Oddities about "read"

; read : nothing -> object

; waits for user to type an expression, and returns it

; Note: variable names are treated as symbols, not evaluated

; Function calls are treated as lists, with the function being the first element

; 'x is treated as the function call (quote x)

# Changing variable values

```
(define toys empty)
(cons "ball" toys) "should be" (list "ball")
toys "is still" empty
; add-toy : symbol -> nothing, but changes the value of toys
"Examples of add-toy:"
(add-toy "ball")
toys "should be" (list "ball")
(add-toy "nintendo")
toys "should be" (list "nintendo" "ball")
```

# Changing variable values

; set! : variable expression -> nothing, but changes the variable's value to be the expression

; Note: only works if the variable is already defined

; Convention: name ends in !, indicating that the function *changes* at least one of its arguments

```
"Examples of set!:"
(define toys empty)
(set! toys (list "ball"))
toys "should be" (list "ball")
(set! toys (cons "nintendo" toys))
toys "should be" (list "nintendo" "ball")
```

# Changing variable values

```
; add-toy : symbol -> nothing, but changes the value
    of toys
(define (add-toy new-toy)
    (set! toys (cons new-toy toys)))
"Examples of add-toy:"
(add-toy "ball")
toys "should be" (list "ball")
(add-toy "nintendo")
toys "should be" (list "nintendo" "ball")
```

# Now you try it

```
(define age 18)
; birthday : nothing -> nothing, changes age


"Examples of birthday:"
(birthday)
age "should be" 19
(birthday)
age "should be" 20
```

# My solution

```
(define age 18)
; birthday : nothing -> nothing, changes age
(define (birthday)
   (set! age (+ 1 age)))
"Examples of birthday:"
(birthday)
age "should be" 19
(birthday)
age "should be" 20
```

# Combining set! and begin

```
(define counter 0)
; count : nothing -> num
; returns 1 more each time you call it
```

```
"Examples of count:"
(count) "should be" 1
(count) "should be" 2
(count) "should be" 3
```

# Combining set! and begin

```
(define counter 0)
; count : nothing -> num
; returns 1 more each time you call it
(define (count)
  (begin                ; remember, returns the value of its last expression
    (set! counter (+ 1 counter))
    counter))
"Examples of count:"
(count) "should be" 1
(count) "should be" 2
(count) "should be" 3
```

# A problem with set!

(define-struct person [name age shoe-size])

(define prof (make-person "Steve" 40 10.5))

(define me prof)

(set! me (make-person "Steve" 41 10.5))

prof "is still 40 years old!"

Problem: set! changes the *variable*, not the object it refers to.

# Modifying a structure

; set-person-age! : person num -> nothing,  but changes the age of the person

(define prof (make-person "Steve" 40 10.5))

(define me prof)

(set-person-age! me 41)

prof "is now 41 years old!"

# Recall constructor, selector, and discriminator functions for a structure type

```
(define-struct person [name age shoe-size])
; make-person : string num num -> person
; person-name : person -> string
; person-age : person -> num
; person-shoe-size : person -> num
; person? : object -> boolean
```

# There are also *mutator* functions for a structure type

```
(define-struct person [name age shoe-size])
; make-person : string num num -> person
; person-name : person -> string
; person-age : person -> num
; person-shoe-size : person -> num
; person? : object -> boolean
; set-person-name! : person string -> nothing
; set-person-age! : person num -> nothing
; set-person-shoe-size! : person num -> nothing
```

# Example

```
(define-struct employee [name num salary])
; give-raise! : emp num -> nothing, but changes the employee's salary by num%
(define (give-raise! emp percent)
      …
  )

"Examples of give-raise!:"
(define joe (make-employee "Joe" 7 54000))
(give-raise! joe 10)
joe "should be" (make-employee "Joe" 7 59400)
```

# Example

```
(define-struct employee [name num salary])
; give-raise! : emp num -> nothing, but changes the employee's salary by num%
(define (give-raise! emp percent)
   (set-employee-salary! emp
    (* (employee-salary emp)
       (+ 1 (/ percent 100)))))
"Examples of give-raise!:"
(define joe (make-employee "Joe" 7 54000))
(give-raise! joe 10)
joe "should be" (make-employee "Joe" 7 59400)
```