CSC 160
Computer Programming
for Non-Majors
Day 16 (June 20, 2005)

Dr. Stephen Bloch
sbloch@adelphi.edu
http://www.adelphi.edu/sbloch/class/160/

Review

- New syntax rule allows "local" definitions
- Can use for both variables and functions
- Common applications:
  - save recursive results to be used several times; improve *efficiency*
  - give names to intermediate results; improve *readability*
  - hide things "outside world" doesn't need to know about; improve *modularization*

Modularization

- Large project w/several programmers
- Each in charge of a "module" of the project
- For example, video game
  - one module in charge of rules of game
  - another module in charge of graphics, menus, buttons, etc.

"Open" scenario

- Jim changes one of his auxiliary functions to return a different type
- Julia, who was using Jim's auxiliary function in her module, finds her module mysteriously stop working
- Problem is hard to track down because Julia doesn't even know Jim has changed the function

Information-hiding scenario

- Each module has
  - "public interface" (like a function contract & examples)
  - "private implementation" (like a function body)
- Each programmer knows only the "public interface" of other programmers' modules
- Jim's auxiliary function isn't in his public interface, so nobody else is using it, so he can change it w/o screwing up Julia's module
- If Jim changes a publicly-known function, but it still satisfies its public interface, it won't screw up Julia's module

Interface vs. implementation

- Interface: "What you need to know in order to *use* the function(s)"
  How functions can be called & what they should return.
- Implementation: "What you need to know in order to *write* or *fix* the function(s)"
  How functions compute the result

Interface vs. implementation: Automobile analogy

- interface is steering wheel, pedals, gear shift…
- implementation is engine, carburetor, spark plugs, …
- If repair shop changes positions of pedals, you have to learn how to drive again.
- If repair shop replaces spark plugs w/better model, but steering wheel, pedals, etc. still work as before, you don't have to change anything.

Review: operating 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)
```

Review: 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))
```

Generalizing the contract

Nothing in **remove-if** or **do-to-each** actually depends on *numbers*

Real contracts are

; remove-if : (X -> boolean) list-of-X -> list-of-X

; do-to-each : (X -> X) list-of-X -> list-of-X

where X is *any* type

Writing functions using these

```
; fire-over-100K : list-of-emps -> list-of-emps
; Auxiliary function earns-over-100K? : emp -> boolean
(define (earns-over-100K? emp)
    (> (emp-salary emp) 100000))
(define (fire-over-100K emps)
    (remove-if earns-over-100K? emps))

; give-10%-raises: list-of-emps -> list-of-emps
; Auxiliary function give-10%-raise : emp -> emp
(define (give-10%-raise emp)
    (make-emp (emp-name emp)  (emp-id emp)
                (* 1.1 (emp-salary emp))))
(define (give-10%-raises emps)
    (do-to-each give-10%-raise emps))
```

Generalizing even farther

Nothing in **do-to-each** requires input and output lists to be the *same* type

Real contract is

; do-to-each : (X -> Y) list-of-X -> list-of-Y

where X and Y are *any two* types, possibly the same.

Writing functions using this

```
; extract-names : list-of-emps -> list-of-strings
(define (extract-names emps)
   (do-to-each emp-name emps))


"Example of extract-names:"
(extract-names
   (list (make-emp "Joe" 1 75000)
         (make-emp "Mary" 2 79995)
         (make-emp "Phil" 3 26000)))
   "should be" (list "Joe" "Mary" "Phil")
```

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)))
```

Could do the same thing with **earns-over-100K?** and **give-10%-raise**

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.

A trickier example

; add-up : list-of-nums -> num

; multiply-all : list-of-nums -> num

; largest : non-empty-list-of-nums -> num

; highest-paid : non-empty-list-of-emps -> emp

What these have in common is that they *combine pairs* of objects to get a third object, repeatedly until whole list has been combined

So we **generalize**. Note that in each case, we need to know what value to start with…

A trickier example

```
; combine : X (X X -> X) list-of-X -> X
(define (combine start-value combiner values)
    …)

"Examples of combine:"
(define (add-up nums)
    (combine 0 + nums))
; insert standard test cases for add-up here

(define (multiply-all nums)
    (combine 1 * nums))
; insert standard test cases for multiply-all here
```

A trickier example

```
(define (largest nums)
   (local [(define (larger num1 num2)
              (cond [(> num1 num2) num1]
                    [else num2]))]
      (combine (first nums) larger (rest nums))))
; insert standard test cases for largest here

(define (highest-paid emps)
   (local [(define (higher-paid emp1 emp2)
              (cond [(> (emp-salary emp1) (emp-salary emp2)) emp1]
                    [else emp2]))]
      (combine (first emps) higher-paid (rest emps))))
; insert standard test cases for highest-paid here
```

A trickier example

In fact, there's no rule that the types of list elements and the type of the result are the same…

; combine : Y (X Y -> Y) list-of-X -> Y

For example,

```
; add-blue-dots : list-of-posns image (background) -> image
(define (add-blue-dots posns background)
  (local [(define (add-blue-dot where background)
               (add-colored-dot where "blue" background))]
    (combine background add-blue-dot posns)))
```

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