CSC 270
Survey of Programming Languages
Sept. 24, 2009

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

Recall "largest" function

```
; largest : non-empty-list-of-numbers -> number
(define (largest nums)
  (cond [(empty? (rest nums)) (first nums)]
        [(cons? (rest nums))
         (cond [(>= (first nums) (largest (rest nums)))
                (first nums)]
               [else
                (largest (rest nums))])
        ]))
```

Performance issues

(largest (list 20 19 18 17 … 3 2 1)) returns 20 instantly

(largest (list 1 2 3 … 17 18 19 20)) returns 20 after 40 seconds!

To find out what's wrong, let's step through

(largest (list 1 2 3 4))

- (largest (list 1 2 3 4))
  - (largest (list 2 3 4))
    - (largest (list 3 4))
      - (largest (list 4)) = 4
      - 3 isn't >= 4, so ….
      - (largest (list 4)) = 4
      - return 4
    - 2 isn't >= 4, so…
    - (largest (list 3 4))
      - (largest (list 4)) = 4
      - 3 isn't >= 4, so ….
      - (largest (list 4)) = 4
      - return 4
    - return 4
  - 1 isn't >= 4, so…
  - (largest (list 2 3 4))
    - (largest (list 3 4))
      - (largest (list 4)) = 4 etc. etc.

We're solving the same problem over and over!

A new syntax rule

(local [*definition definition* …]
   expression

Example:

```
(local [(define x 7)]
   (+ x 5))
```

"should be" 12

x "is now undefined again"

More examples of local

```
(define bignum 1234567890)
(local [(define bignum 5)]
   (* bignum bignum bignum))
"should be" 125
bignum "should be" 1234567890 "again"
```

Using this to improve "largest"

```
; largest : non-empty-list-of-numbers -> number
(define (largest nums)
    (cond [(empty? (rest nums)) (first nums)]
          [(cons? (rest nums))
           (local [(define biggest-of-rest (largest (rest nums)))]
            (cond [(>= (first nums) biggest-of-rest)
                   (first nums)]
                  [else
                   biggest-of-rest] ))
          ]))
```

Another approach to "largest"

```
; larger : num num -> num
…
"Examples of larger:"
(check-expect (larger 5 2) 5)
(check-expect (larger 2 5) 5)
(check-expect (larger 4 4) 4)

; largest : non-empty-list-of-numbers -> number
(define (largest nums)
     (cond [(empty? (rest nums)) (first nums)]
           [(cons? (rest nums))
             (larger (first nums) (largest (rest nums)))
            ]))
```

If we didn't need "larger" anywhere else…

```
; largest : non-empty-list-of-numbers -> number
(define (largest nums)
  (local [(define (larger x y)
                (cond [(>= x y) x] [else y]))]
    (cond [(empty? (rest nums)) (first nums)]
          [(cons? (rest nums))
           (larger (first nums) (largest (rest nums)))
          ])))
```

Review

- New syntax rule allows "local" definitions
- Can use for variables, functions, even structs
- 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*

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)))])]))

(check-expect (remove>10 empty) empty)
(check-expect (remove>10 (list 6)) (list 6))
(check-expect (remove>10 (list 11)) empty)
(check-expect (remove>10 (list 6 11 10 -24 13 9)) (list 6 10 -24 9))
(check-expect (remove>10 (list 11 10 -24 13 9)) (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:"
(check-expect (remove-over 6 empty) empty)
…
(check-expect (remove-over 3.5 (list 4 9 17 2 6 3)) (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** or **PLAI**

Defining remove-if

```
; 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)))])]))
(check-expect (remove-if even? (list 1 2 3 4 5)) (list 1 3 5))
(define (over-10? x) (> x 10))
(check-expect(remove-if over-10? (list 3 12 10 5 16 -24 6)) (list 3 10 5 -24 6))
(define (under-5? x) (< x 5))
(check-expect (remove-if under-5? (list 3 12 10 5 16 -24 6)) (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-evens : list-of-nums -> list-of-nums
(define (remove-evens nums) (remove-if even? nums))
```

Actually, we don't need to write this…

There's a built-in function

filter : (X -> boolean) list-of-X -> list-of-X

that does basically the same thing, except it *keeps* the items that pass the test, rather than *removing* the items that pass the test.

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)))]))


(check-expect (cube-each empty) empty)
(check-expect (cube-each (list 2)) (list 8))
(check-expect (cube-each (list 3 -2 0 5 -6)) (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)))]))

(check-expect (do-to-each cube (list 3 5 -2)) (list 27 125 -8))
(check-expect (do-to-each sqrt (list 4 25 0)) (list 2 5 0))
(check-expect (do-to-each - (list 3 -2 0 7.5)) (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))
```

Pop quiz

- What other functions did you write on HW2 that could have been written using **do-to-each** or **remove-if**?

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:"
(check-expect (extract-names
   (list (make-emp "Joe" 1 75000)
         (make-emp "Mary" 2 79995)
         (make-emp "Phil" 3 26000)))
   (list "Joe" "Mary" "Phil"))
```

We don't need to write this…

There's a built-in function

map : (X -> Y) list-of-X -> list-of-Y

that does basically the same thing.

Actually, it works with multiple lists:

map : (X_1 X_2 X_3 -> Y) list-of-X_1 list-of-X_2 list-of-X_3 -> list-of-Y

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) (map add3 nums))
```

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

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

A trickier example

On list '(a b c d e), all of these functions compute

f(a,f(b,f(c,f(d,f(e,BASE)))))

where BASE is the answer to the empty case.

The functions differ only in "f" and "BASE".

All these functions *combine pairs* of objects to get a third object, repeatedly until whole list has been combined

So we **generalize**.

A trickier example

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


(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

```
; convert-reversed : list-of-nums -> num
(define (convert-reversed digits)
  (local ((define (add-digit d v) (+ d (* 10 v))))
    (combine add-digit 0 digits)))
; insert standard test cases for convert-reversed here
```

A trickier example

```
(define (largest nums)
    (local [(define (larger num1 num2)
              (cond [(> num1 num2) num1]
                    [else num2]))]
      (combine larger (first nums) (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 higher-paid (first emps) (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 add-blue-dot background posns)))
```

We don't need to write this…

There's a built-in function

foldr : (X Y -> Y) Y list-of-X -> Y

that does basically the same thing.

Actually, it works with multiple lists:

foldr : (X_1 X_2 X_3 Y -> Y) Y list-of-X_1 list-of-X_2 list-of-X_3 -> Y

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** or **PLAI**

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

Can also write functions that *return* functions *as values*

; make-adder : number -> (number -> number)

"Examples of make-adder:"

(make-adder 3) "should be a function that adds 3"

((make-adder 3) 5) "should be 8"

(do-to-each (make-adder -1) (list 5 2 -4 6)) "should be" (list 4 1 -5 5)

Can also write functions that *return* functions *as values*

; make-adder : number -> (number -> number)

```
(define (make-adder increment)
  (local [(define (f num)
            (+ num increment))]
    f))
```

Can also write functions that *return* functions *as values*

```
; make-adder : number -> (number -> number)
(define (make-adder increment)
  (local [(define (f num)
              (+ num increment))]
    f))
; Alternate definition:
(define (make-adder increment)
  (lambda (num) (+ num increment)))
```

Can also write functions that *return* functions *as values*

Project 1 requires you to write a function that returns a function.

HW3 will have several exercises of this kind.