

CSC 160
Computer Programming
for Non-Majors
Day 17 (June 21, 2005)

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

Review

- Modularization and information-hiding; interface vs. implementation
- Generalizing functions w/extra parameters
- Generalizing functions w/*function* parameters
 - remove-if
 - do-to-each
 - combine
- Anonymous functions with lambda

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.

Now you try it

```
(define-struct employee [name id salary])
```

```
; print-employee: employee -->nothing, but prints out the  
  employee's information, nicely formatted
```

```
"Examples of print-employee:"
```

```
(print-employee (make-employee "Joe" 7 54000))
```

```
"should print"
```

```
"Joe, employee #7, earns $54000/year."
```

My solution

```
(define-struct employee [name id salary])
```

```
; print-employee: employee -->nothing, but prints out the employee's  
information, nicely formatted
```

```
(define (print-employee emp)
```

```
  (begin
```

```
    (display (employee-name emp))
```

```
    (display-with-label ", employee #" (employee-id emp))
```

```
    (display-with-label ", earns $" (employee-salary emp))
```

```
    (display "/year.")))
```

Changing the format

Now we want output in the form

Joe

Employee #7

\$54000/year

Changing the format

Now we want output in the form

Joe

Employee #7

\$54000/year

; newline : nothing -> nothing, but advances
display output to next line

My solution

```
(define-struct employee [name num salary])
```

```
; print-employee: employee -->nothing, but prints out the employee's  
  information, nicely formatted
```

```
(define (print-employee emp)
```

```
  (begin
```

```
    (display (employee-name emp))
```

```
    (newline)
```

```
    (display-with-label "Employee #" (employee-num emp))
```

```
    (newline)
```

```
    (display (employee-salary emp))
```

```
    (display "/year."))
```

```
  (newline)))
```

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)

Example

- ; repeat-input: nothing -> nothing
- ; Waits for user to type something, then displays it twice on separate lines.

Example

- ; repeat-input: nothing -> nothing
- ; Waits for user to type something, then displays it twice on separate lines.

```
(define (repeat-input)
  (local [(define input (read))])
  (begin
    (display input)
    (newline)
    (display input))))
```

Making this friendlier...

Wouldn't repeat-input be friendlier if it *asked* "What do you want to repeat?"

```
; ask : string -> object
```

```
; prints the string, waits for input, and returns it
```

```
(define (ask question)
```

```
  (begin
```

```
    (display question)
```

```
    (read)))
```

Example

; repeat-input: nothing -> nothing

; Prompts for user to type something, then displays it twice on separate lines.

```
(define (repeat-input)
```

```
  (local [(define input (ask "What do you want to repeat?"))]
```

```
    (begin
```

```
      (display input)
```

```
      (newline)
```

```
      (display input))))
```

Now you try it

; ask-posn : nothing -> posn

; Prompts "x coordinate?" and "y coordinate?" and puts inputs together into a posn.

"Example of ask-posn:"

(ask-posn)

; should print "x coordinate?" You type 3

; should print "y coordinate?" You type 4

; should return (make-posn 3 4)

; Hint: you can use **local**, but you don't need to.

Changing variable values

```
(define toys empty)
```

```
(cons "ball" toys) "should be" (list "ball")
```

```
toys "is still" empty
```

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)
```

Summary of today

- Text output: display, newline
- Text input: read
- Doing multiple things in sequence: begin
- Changing value of a variable: set!
- Changing a field of a struct:
set-structname-fieldname!