Administrative Notes

Last exam

- Hand out on 19th, due 24th (Wednesday)
- Will cover material since last exam
- Take home, three hours
- Closed notes, closed books

Last Homework

- Available this afternoon
- Due next Wednesday

Set-structure!

We've only seen trivial examples, so far

- Develop an online address book
- Simple interface — two functions
  - → Insert new addresses — <name, address> pairs
  - → Lookup a name and get back a phone number

```
;; an entry is a structure
;;   (make-entry  name number)
;; where name is a symbol and number is a number
(define-struct entry (name number))

;; address-book is a list of entry
(define address-book empty)    ;; initial condition
```

Address book

And the two functions in the interface

```
;; lookup-number : symbol  -> number or false
;; Purpose:  returns the phone number for symbol, or
;;                false if no entry for symbol is in address-book
(define (lookup-number who) …)

;; add-to-address-book:  symbol  number -> true
;; Purpose: adds an entry to the address book
(define (add-to-address-book who phone) …)
```

Address book

What about test data?

```
(lookup-number ‘John)
```

What’s the expected answer?

That depends on the past

```
(add-to-address-book ‘John 7135551212)
(lookup-number ‘John)
⇒ 7135551212
```

With state, test data needs a robust history (or context)

Address book

```
;; lookup-number : symbol  -> number or false
;; Purpose:  returns the phone number for symbol, or
;;               false if no entry for symbol is in address-book
(define (lookup-number who)
   (local [(define   matches
                       (filter (lambda(x) (symbol=? who (entry-name x)))
                               address-book))]
           (cond [(empty? matches)   false]
                  [else   (entry-number (first matches))] ) ) )


;; add-to-address-book:  symbol  number -> true
;; Purpose: adds an entry to the address book
;; Effect: changes the value of address-book by adding a new entry
(define (add-to-address-book who phone)
   (begin
       (set!  address-book (cons (make-entry who phone) address-book))
       true) )
```

Address book

What happens when someone moves?

- Need to change their phone number
- How should we accomplish this?

Two classic schemes

- Create a new entry that supercedes old entry
  - → Adds to length (& cost of filter operation in lookup)
- Rebuild the list, replacing old entry with new entry
  - → Does not lengthen the list

Address book

Changing an entry

```
;; change-number1:  symbol  number -> true
;; Purpose: changes an existing phone number in the address book
;; Effect: redefines "address-book" with a new list that contains old list
(define (change-number1 who phone)
   ;; strategy 1: add to front of the list
   (begin
     (set!  address-book (cons (make-entry who phone)
                                           address-book))
      true) )
```

$\left.\right\}$ *This should be very fast*

Unintended consequences

- Changing a non-existent entry is same as adding it
- Either a <u>bug</u> or a <u>feature</u>

Address book

Changing an entry

```
;; change-number2:  symbol  number -> true
;; Purpose: changes an existing phone number in the address book
;; Effect: redefines "address-book" with a new list
(define (change-number2 who phone)
   ;; strategy 2: replace existing entry
   (begin
     (set!  address-book
            (cons (make-entry who phone)
                  (filter (lambda(x)(not (symbol=? who (entry-name x)) ))
                          address-book)))
      true) )
```

This version

- Does not lengthen address-book
- Filter re-builds entire address book, minus matching entries

Address book

Look at number of cons operations used

- Strategy 1 performs a single cons operation
  - → But it grows the list over time
- Str ss2
  - → E ok) - 1 cons

Imagine updating Southwestern Bell's telephone book for Houston

Several million entries, several hundred changes per day, …

That's a lot of cons operations and a lot of garbage to recycle

Price o

- The
- Following the filter with the set! to redefine address-book adds insult to the injury
  - → Creates lots of garbage for DrScheme to recycle

Address book

More efficient update

- Would like to move the set! Down into the list
  - → Find the entry that must change
  - → Use a set!-like effect to change its number field
- Avoid rebuilding the list, doing all those cons operations, & creating all that garbage

Enter "set-structure!"

- Define-struct creates some more functions
- For "entry": set-entry-name! and set-entry-number!

Address book

More efficient update

```
;; change-number3:  symbol  number -> boolean
;; Purpose: changes an existing phone number in the address book
;; Effect: modifies entry's phone number in place
(define (change-number3 who phone)
   local [(define aloe (filter (lambda(x)(symbol=? who (entry-name x)))
                                 address-book))]
          (cond [(empty? aloe)  false]
                  [(cons?  aloe)
                   (begin
                      (set-entry-number! (first aloe)  phone)
                       true)] )))
```

Interface changed, too

- For no extra cost, we can return false on failure
- Does not add new entries

Address book

The roommate problem

- Roommate wants to use your software
- Types (define address-book empty) to begin
    - → Oops. There went your address book!

Malicious person can have same effect with set!

- → Change phone numbers
- → Delete money from checkbook program
- → Change password in operating system
- → And so on, …

How can we design to avoid such abuses?

Hiding data

Possible solutions

- Hide address-book in a local inside the program
  - → Where? What programs need access to it?
  - → Kernel of a good thought here
  - → Should only use set! on local objects

- Hide functions together inside a local defining address-book
  - → Gives them all access to address-book
  - → Gives chance to initialize address-book
  - → How do we invoke the various programs?

Hiding data

Try something like

```
(define address-interface
   (local [ (define address-book empty)
             (define (lookup-number who)
                 ( … ))
             (define (add-to-address-book who phone)
                 ( … ))
             (define (change-number who phone)
                 ( … )) ]
           … what should this program return? …
   ))
```

((first (rest address-interface)) 'Tim 7133485185)

Hiding data

Options for address-interface

1. List of functions
    - (list lookup-number add-to-address-book change-number)
    - Does not scale
        - Works at 3 functions, not at 20
        - User must remember ordinal position
    - Terrible, counter-intuitive interface
        - What do you type for change-number?
    - No good rationalization for it
        - Function that returns a list of functions?
        - This does not sound like COMP 210

Should return one function

Hiding data

Options for address-interface

2. Return one program
    - It should map symbol -> program

```
(lambda(x)
  (cond
     [(symbol=? 'lookup x)   lookup-name]
     [(symbol=? 'add      x)  add-to-address-book]
     [(symbol=? 'change x) change-number]
  ))
```

- Now, we can instantiate address-interface and use it
- Creates private, hidden address book
- Returns a function that can be used to define accessors

Hiding data

Using it

```
(define mybook
   (local [ (define address-book empty)
            (define (lookup-number who)
                ( … ))
            (define (add-to-address-book who phone)
                ( … ))
            (define (change-number who phone)
                ( … )) ]
            (lambda(x)
              (cond  [(symbol=? ‘lookup x)   lookup-name]
                     [(symbol=? ‘add     x)  add-to-address-book]
                     [(symbol=? ‘change x) change-number] )) ))
```

Hiding data

Using it

```
((mybook ‘add)  ‘Keith  7136656325)
((mybook ‘lookup) ‘Tim)
```

Kind of awkward

```
(define lookup (mybook ‘lookup))
(define add     (mybook ‘add))
(define change (mybook ‘change))
(add ‘Keith 7136656325)
(lookup ‘Tim)
```