Administrative Notes

Last Homework

- Available this afternoon
- Due next Friday

Address book example

- We defined a structure to hold names & numbers
- We built some functions
  - → Lookup-number, add-to-address-book
  - → Change-number (several versions, set-structure!)
- We looked at ways to hide the actual address-book object
  - → Ended up building an interface function
  - → Address-interface: symbol -> function

Hiding data

Schema for address-book

```
(define address-interface
   (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] )) ))
```

The actual code & data

The interface

Address-interface is defined as the interface function

Hiding data

Using it

```
((address-interface 'add)  'Keith  7136656325)
((address-interface 'lookup) 'Tim)
```

Kind of awkward

```
(define lookup (address-interface 'lookup))
(define add      (address-interface 'add))
(define change (address-interface 'change))
(add 'Keith 7136656325)
(lookup 'Tim)
```

Handling success

Of course, since this address book is revolutionary …

- Others want to use it
- It only creates one address book
  - → Accessed through the interface, but one book
- Need a way to create multiple, independent books

Handling success (and reuse)

Add one more layer …

```
(define create-address-book
  (local [(define address-interface
            (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] )) )) ]
    (lambda() address-interface)
  ))
```

Function of zero arguments that returns an address-book interface

Handling success (and reuse)

Using it

(define KeithBook (create-address-book))

((KeithBook ‘add) ‘Keith 7136656325)

((KeithBook ‘add) ‘Tim 7133485185)

((KeithBook ‘lookup) ‘Tim) ⇒ 7133485185

(define LindaBook (create-address-book))

((LindaBook ‘add) ‘Vicky 7133486041)

((KeithBook ‘lookup) ‘Vicky) ⇒ false

((LindaBook ‘lookup) ‘Tim) ⇒ false

….

Handling success (and reuse)

And, of course, we can create shortcuts

(define KeithBook (create-address-book))

(define klookup (KeithBook ‘lookup))

(define kadd (KeithBook ‘add))

(kadd ‘Tim 7133485185)

(klookup ‘Tim) ⇒ 7133485185

(klookup ‘Keith) ⇒ false

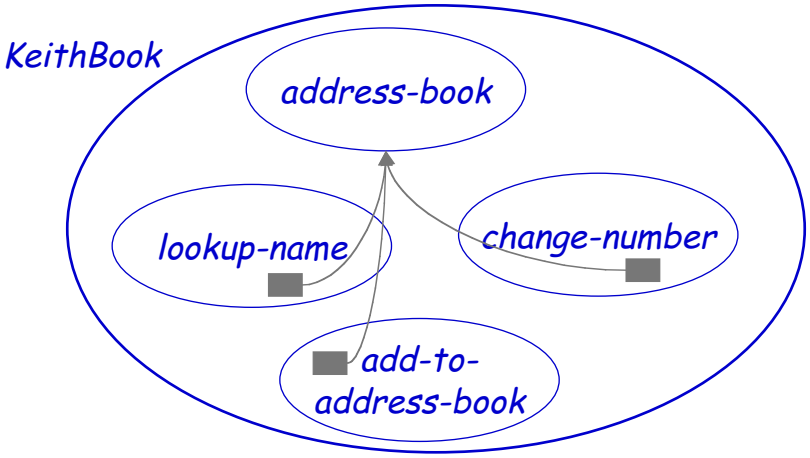
….

Handling success

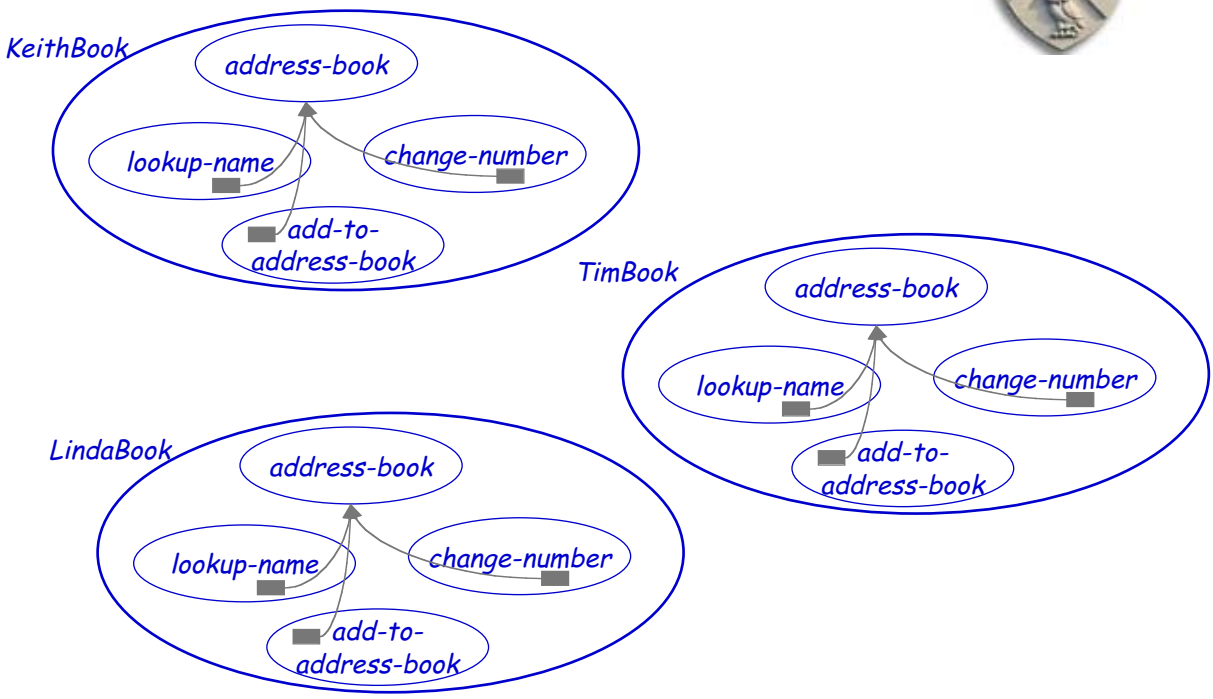
What happened?

- Create-address-book returns address-interface
  - → Every time it runs, it creates a new address-interface
  - → And a new local inside it
  - → And a new address book with the access functions
- Separate invocations of create address book
  - → Create separate copies of lookup-name, add-to-address-book, and change-number, along with address-book
  - → Rewriting for local gives them all unique names
  - → Bindings ensure separation & privacy
    - No other code can touch your address book

Handling success (and reuse)



Handling success (and reuse)



The Big Picture

Introduced Scheme

- Language has almost no syntax, but lots of power
- Used Scheme to make giant strides in programming
  - → Did some algebraic programming
  - → Learned about unbounded data structures (lists & trees)
  - → Structural recursion
  - → Generative recursion
  - → Abstract functions
  - → Think about the complexity of missionaries & cannibals
- You've all done a lot of learning and a lot of work

The Big Picture

What does this have to do with the rest of the world?

- They use C, or Java, or C++, or Fortran, or Perl, or …
- The basic concepts of programming are the same
  - → You have been biased toward functional programming
  - → Later courses will undo much of that bias
- The syntax & structure of those languages are different
  - → Problem solving & program development are similar
  - → Skills from 210 will help with low-level details, too
- Tools from 210 will help you understand all the other languages that you encounter

The Big Picture

COMP 210 Concepts

- Contract : notion of a type system & type correctness
- Structures : aggregates in almost every language
- Lists : natural interface in Scheme, used in many applications where size of the input is unknown
- Trees : critical data structure for many applications
- Functions : taught you to think of them as data
  - → Critical underpinning of higher-order languages
  - → Tail-recursion was critical to yesterday's talk by Taha
- Abstract functions : fundamental strategy for code reuse
- Assignment : important efficiency hack

The Big Picture

Local

- Introduced it for many reasons
  - → Efficiency, clarity, name-space management, …
  - → Used it to isolate effects

    "only set! an object defined in a local"
  - → Used it to create hidden state, interface functions, …

Local models lexical scoping

- Key feature of almost all programming languages
- Minor variations in its application, rules, & use
- You now have the tools to understand those variations