

COMP 210, Fall 2000

Lecture 34 – Finishing up Data Abstraction & Objects

Announcements

1. The next homework will be available this afternoon.

Review

We did some more work with the addressing book example. We came to the conclusion that *Only variables hidden inside a local should be **set!**'ed* — not for deep methodological reasons, but for security reasons. If a program is going to create persistent state and rely on it, the program needs some reassurance that random or unintended actions by other programs (or programmers) do not harm that state.

Moving on ...

We build an address-book interface function, and talked about the options for what it could return. Here is where we ended up...

```
(define address-interface
  (local [
    (define address-book empty)
    (define (lookup-number name)
      (local [(define matches (filter (lambda(an-entry)
                                         (symbol=? name (entry-name an-entry)))
                                         address-book))]
        (cond [(empty? matches) false]
              [else (entry-number (first matches))])))
    (define (add-to-address-book name num)
      (begin
        (set! address-book
              (cons (make-entry name num) address-book))
        true)) ]
    (lambda(service)
      (cond [(symbol=? service 'lookup) lookup-number]
            [symbol=? service 'add) add-to-address-book]))
  ))
```

And, we defined some names to hold these functions

```
(define lookup-an-address (address-interface 'lookup))
```

```

(define add-an-address (address-interface 'add))

(lookup-an-address 'Keith)
➤ false
(add-an-address 'Keith 7133486013)
➤ true
(lookup-an-address 'Keith)
➤ 7133486013

```

We also looked at using the functions directly:

```

((address-interface 'lookup) 'Keith)
➤ false
((address-interface 'add) 'Keith 7133486013)
➤ true
((address-interface 'lookup) 'Keith)
➤ 7133486013

```

This looks awful, but works just fine. There are times when you might want to use this kind of interface.

One Final Extension To This Example

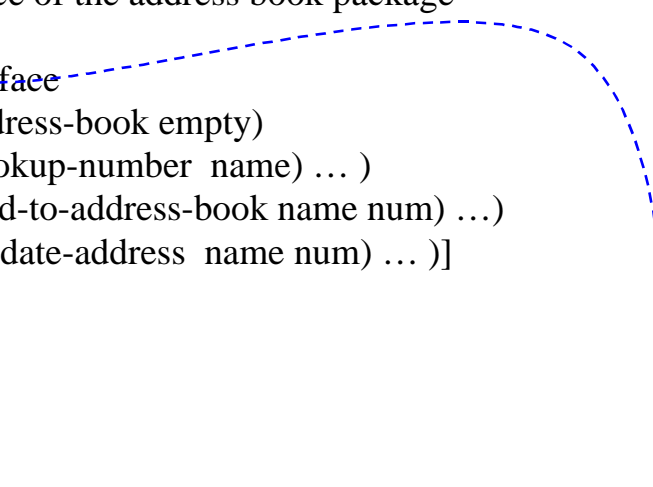
Of course, now that you've built an marvelous tool like your address book, you would like to share it with your friends, your relatives, and (maybe) your customers. (After all, that's how companies get started.) With its hidden state, how can we share an address book? <ask for solutions>

One way to share these tools is to take this process of abstraction one step further. We've already built this wrapper function, or interface function, that encapsulates the **local** holding the data and the functions that operate on it. What if we wrap that function in one more layer of encapsulation and build a meta-wrapper function that returns copies of our address interface function? It could look like

```

;; create-address-book: void → (symbol → address-book-service)
;; Purpose: return a new instance of the address book package
(define create-address-book
  (local [define-address-interface
    (local [(define address-book empty)
      (define (lookup-number name) ...)
      (define (add-to-address-book name num) ...)
      (define (update-address name num) ...)]
      (lambda(e)
        (cond

```



```

[(symbol=? 'lookup e) lookup-number]
[(symbol=? 'add e) add-to-address-book]
[(symbol=? 'update e) update-address]
)])

```

```

(lambda( )
  address-interface)
))

```

Now, we can do things such as

```

➤ (define Keith-book (create-address-book))
➤ (define Tim-book (create-address-book))
➤ ((Keith-book 'add) 'Linda 5177)
true
➤ ((Tim-book 'lookup) 'Linda)
false
➤ ((Keith-book 'lookup) 'Linda)
5177

```

So, what does this function **create-address-book** really do? It creates a new instance of our **address-interface** function and returns it. That has the effect of creating a unique copy of the address-book local variable, and unique copies of the three functions, with the appropriate rewriting to create a unique, unknowable name that is embedded in the functions.

A call to (create-address-book) creates a new instance of an address book. That instance has its own private data. The user can only access the private data by going through the functions that a particular instance of **address-interface** provides. Tim's address book cannot see Keith's data, nor can Keith's address book see Tim's data. In C++ or Java, we would think of these **address-interface** functions as objects. We access them by sending them a “message” – in the form of the symbols 'add, 'lookup, or 'update—and some arguments for the message. They perform their actions on the hidden data and return some sort of value.

Why have we done all of this? What you've learned this week are the basic concepts that underlie the design of objects (as in object-oriented programming). An object is just one of these interface objects—a function wrapped up with its private data. (We sometimes call this a closure.) Each function, or method, is hidden inside the interface and responds to a specific message. Thus, you have learned two of the fundamental concepts behind object-oriented programming—namely data-driven processing (*i.e.*, the

whole methodology built around templates) and encapsulated services, as in the address book.

To carry the analogy further, **create-address-book** plays the role of a class in C++ or Java. All you need to have a full-blown object-oriented system is the ability for one class to derive functions by default from other classes—called *inheritance* (don't worry if you haven't heard of inheritance in this way; you will in COMP 212). You could add inheritance to this kind of programming without too much trouble.

The Big Picture

Where have we been? And why? In the beginning, we taught you Scheme. Some people accepted the notion that Scheme was the obvious language for this course; others were less trusting. By focusing our attention on programming, rather than the language, we have made a huge amount of progress in teaching you to design programs. (See *Missionaries and Cannibals*, for example, or quicksort, or mergesort, ...)

We taught you a bottom-up, data-driven methodology for writing small programs. Large systems are built of many small programs. Understanding how to write those small programs so that they work, so that they are easy to read and easy to modify, and so that you have a high-degree of confidence in their correctness is the first step toward building large systems.

We introduced **local** for what seem like an endless array of reasons. We used local to speed up **max**—essentially, saving a value so that we could use it twice. We used it to hide helper functions—something that matters if you are trying to manage the name space of a large program. We used it to define functions with hidden persistent state—and state modified using **set!** and **set-structure!** We tried to give you some intuition about when it makes sense to hide details behind a **local**. Finally, we used **local** and **lambda** to create closures and build simple, prototype objects.

At times, the whole exercise with **local** seemed pointless. Once we got around to objects, it might be cool, but what about all of that stuff that preceded objects? Scheme's **local** construct creates *lexical scopes*. Almost every programming language that you learn will have lexical scopes. The ways that they provide scopes, and the ways that scopes can be used may be subtly different from language to language, but all of them can be modeled using the **local** construct that you've learned in COMP 210. What we've done is to give you the conceptual tools to understand those minor variations on scoping rules.

