COMP 210, Spring 2001

Lecture 23: A Case Study in How to Use Mutation

Reminders

Review

We explained how to simulate structures (including structure mutation) in functional Scheme + **`set!`**. It involved hiding variables with a **`local`** that returns a **`lambda`** expression.

An Example for `set-structure!` and `set!`

Consider implementing an online phone book that maps names to phone numbers. Such a phone book must support two features—we must be able to insert new entries and we need to be able to look up a name and get back the phone number. We can represent a phone book entry as a simple structure that has two fields.

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

; phone-book : list of entry
; keep track of the current phone book entries
(define phone-book empty)
```

Now we need two functions

```
; lookup-number : symbol phone-book → (number or false)
;   (lookup-number name) returns the phone number associated with name
;   or false if name is not found

; add-to-phone-book : symbol number → (void)
;   (add-to-phone-book s n) returns (void)
;   Effect: adds the entry (s,n) to the phone book
```

Can we write down the test data for these programs? It is somehow more complex than the cases that we have seen in the past. If we try

```
      (lookup-number ‘Todd)
```

the answer depends on what has happened since we last clicked the execute button in DrScheme. If we have already executed the expression

```
      (add-to-phone-book ‘Todd  7135551212)
```

then the call to lookup-number should return **`7135551212`**. If we have never added **`‘Todd`** to the phone book, then it should return **`false`**.

To write down something that has definite results, we need a sequence of calls. For example, we can state that the sequence

```
(add-to-phone-book ‘Corky  7133486042)
(lookup-number ‘Corky)
```

should always have the same results. The call on **`add-to-phone-book`** returns **`(void)`**, and the call to lookup-number returns **`7133486042`**. For programs that have persistent internal state, we need to write more complicated test data that ensures some knowledge of what is preserved in that internal state and then uses that knowledge for testing.

Both of these programs are pretty straight forward:

```
; lookup-number : symbol → (number or false)
;   (lookup-number s) returns the phone number for symbol s, or false
;   if s is not found
(define (lookup-number name)
  (local [(define matches
            (filter (lambda (an-entry)
                      (symbol=? name (entry-name an-entry)))
                    phone-book))]
     (cond [(empty? matches) false]
           [else (entry-number (first matches))])))
```

Leave room for Effect comment

```
; add-to-phone-book : symbol number → (void)
;   (add-to-phone-book s n) returns (void)
;   Effect: add the entry (s,n) to the phone book
  (set! phone-book (cons (make-entry name num) phone-book)))
```

Whenever we write a program that changes the value of a variable using **`set!`** (or **`set-structure!`**), we must document what those changes will be. Thus, the purpose statement we must (i) specify the value returned by the Function and (ii) describe any effects that the program has on the program state (the bindings of the variables defined outside the function. The purpose statement for **`add-to-phone-book`** above shows our recommended format for such purpose statements.

Exercise: write a purpose statement for the function **`mystery`** that we wrote earlier? Effects can be subtle and not obviously related to the value returned by a function.

A Stylistic Digression

Did we write the best definition for **`lookup-number`**? How does the following alternate definition compare with our original definition?

```
(define (lookup-number name)
  (local [(define (lookup-number-help add-book)
            (cond [(empty? add-book) false]
                  [(symbol=? name (entry-name (first add-book)))
                   (entry-number (first matches))]))]
     (lookup-number-help phone-book))
```

Both of these definitions are good ones. One is not clearly better than the other. The second definition is more efficient on successful searches because it does not scan the entire list. The first definition is arguably easier to understand.

Improving Our Phone Book

That happens when someone moves? How do we update the phone book? We need a function **`update`** that takes a name and a number and changes the phone number for that name. How could you write this?

The classic approach, from the days when we thought primarily about structural recursion, would be to rebuild the phone book around a new entry for the person who moved. This would require searching through the phone book for the entry corresponding to name and rebuilding the list on the way back out of the recursion.

```
; update-phone: symbol number → void
;   (update-phone s n) returns (void)
;   Effect: destructively modifies the entry for s to contain
;   the number n
(define (update-phone name num)
  (local [(define updated-book
            (map (lambda (entry)
                   (cond [(symbol=? (entry-name entry) name)
                          (make-entry name num)]
                         [else entry]))
                 phone-book))]
    (set! phone-book updated-book)))
```

There is, however, reason that you might not want to do it that way: ***efficiency*.** If your phone book approaches the size of the White Pages™ for Houston, you might want to avoid building and rebuilding it every time some customer moves. How can we do this? By exploiting the structure mutators

`set-entry-name!: entry symbol → void`

`set-entry-number!: entry number → void`

that Scheme creates for every **`define-struct`** data declaration. Using structure mutators, how can we write update-phone?

```
; update-phone: symbol number → void
;   (update-phone s n) returns (void)
;   Effect: destructively modifies the entry for s to contain
;   the number n

(define (update-phone-book! name new-num)
  (local [(define (helper! a-book)
            (cond
             [(empty? helper) void]
              [else
               (cond
                  [(symbol=? name (entry-name (first a-book)))
                   (set-entry-phone! (first a-book) new-num)]
                  [else (helper! (rest a-book))])]))]
      (helper! phone-book)))
```

Notice that this version of update-phone does not use set! at all. It does not need to change the global variable phone-book, because it changed one of the entries inside phone-book directly. (The earlier version built a whole new phone book to incorporate the change.)