Administrative Notes

Last exam

- Hand out 17th or 19th, due 24th ? (Wednesday)
- Will cover material since last exam
- Take home, three hours
- Closed notes, closed books

Review from Last Class

Introduced the Scheme function set!

- (set! Object (expression))
  - → Evaluates expression and causes Object to refer to its result
  - → A form of assignment
- set! produces no useful value (void)
  - → First Scheme expression we've seen with no value
  - → Need to use consecutive expressions, as with begin
- We used set! to build a memo-function

Memo functions

Started from a simple abstract problem

```
;; f:  number -> number
(define (f x)
   (g (*  x  x))
```

What is g?
We don't need to care

Built a version of f that remembers

- Records arguments and results
- Checks the record before calling g again

Memo functions

Need a representation for the results

```
;; a result is
;;   (make-result arg answer)
;; where arg and answer are numbers
(define-struct result (arg answer))

;; table is a list of result
;; We will use Scheme's built-in constructor for the list
(define table empty)
```

Now,

- Need a new version of f that looks in the table
  - → Returns answer from table if it is found
  - → Computes and records answer if it is not found

Memo functions

We developed a memo-function version of f

```
;; f:  number -> number
(define f
   (local [ (define table empty) ]
      (lambda(x)
         (local [(define prev (filter (lambda(y) (= x (result-arg y))) table))]
            (cond
               [(empty? prev)
                (local [(define new-result (g (* x x )))]
                      (begin
                         (set!  table
                                (cons (make-result x new-result) table))
                          new-result ))]
               [else (result-answer (first prev))] ))) ) )
```

This is simpler than the version in lecture 29

Following suggestion from class with filter …

Memo functions

Set! disrupts our model of the world

- This version of f gives the same answers as the old one
- This version computes them in a different way

```
> (f 2)
37
> (f 3)
77
> (f 2)
37
```

It did not compute (g 4) this time. It found the answer in table

Before set! the rewriting semantics was simple

- Expression evaluation did not depend on prior results
- With set!, it depends on prior results in a critical way

More on set!

- set! changes the world
  - → Evaluation suddenly depends on previous history
  - → New complexity to the rewriting rules for Scheme
- We need to get used to this new, non-functional world
  - → Most other programming languages rely on assignment
- set! introduces time into the evaluation process
  - → Subtle, yet critical, change

More on set!

Consider the sequence

```
(define x 5)
x
(set! x (add1 x))
x
```

Before set!, x always had the same value (in the same scope)

Now, the value of x depends on when we evaluate it

- *Need to know what "effects" have taken place*

sequence of evaluations in DrScheme

That trick with lambda and local

We played a little fast and loose with this one

- In slow-motion, instant replay, it works like this

```
(define (f x) (* x x x))        ⇔    (define f
                                         (lambda(x) (* x x x))

(define f                       ⇔    (define f
    (lambda(x) (* x x x))                (local [(define table empty)]
                                                (lambda(x) (* x x x)))
```

- Now, <u>f</u> is a function of one argument with hidden state
  - → We just made a more complex function of <u>f</u>
  - → Uses set! to change its hidden state (table)
  - → Uses filter to check its hidden state

That trick with lambda and local

Here is the full-blown version of <u>f</u>

```
;; f:  number -> number
(define f
  (local [ (define table empty) ]
     (lambda(x)
       (local [(define prev (filter (lambda(y) (= x (result-arg y))) table))]
             (cond
                 [(empty? prev)
                  (local [(define new-result (g (* x x )))]
                         (begin
                            (set!  table
                                   (cons (make-result x new-result) table))
                            new-result ))]
                 [else (result-answer (first prev))] ))) ) )
```

The more complex function with hidden state

A final note on our memo function, f

Consider the cost of running f

- Performs a filter on whole table every time it runs
- (length table) is number of distinct arguments f has seen
- This might grow to be large
- Cost of f can grow with history

Two lessons in f

- Only use a memo-function when the underlying computation is costly enough to justify the lookup
- Consider better techniques for the lookup
  - → Binary search tree would reduce it from N to $\log_2$ N

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

The functions are pretty simple

```
;; 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))]
          ) ) )
```


Address book

The functions are pretty simple

```
;; add-to-address-book:  symbol  number -> true
;; Purpose: adds an entry to the address book
;; Effect: …
(define (add-to-address-book who phone)
   (begin
      (set!  address-book
             (cons (make-entry who phone) address-book))
      true) )
```

This is still COMP 210. We need to document the use of set!

Why? Because it shows that you've thought about what it does.

Address book

The functions are pretty simple

```
;; 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) )
```


Lambda

How do lambda & define differ?

```
;; times3: number -> number
(define (times3 x)
   (* 3 x))
```

- *Creates a function that multiplies its input by three*
- *Associates that function with the Scheme object "times3"*

```
;; same function, no name
(lambda (x) (* 3 x))
```

- *Creates an anonymous function that multiplies its input by three*

```
;; times3: number -> number
(define times3
    (lambda (x) (* 3 x)) )
```

- *Binds the anonymous function to the Scheme object "times3"*