Topic 19
Mutable Data Objects

Section 3.3

Mutable data

```
Basic operations like cons, car, cdr can
  construct list structure and select its
  parts, but cannot modify.

(define x (cons 'a 'b))
x --> (a . b)

Primitive mutators for pairs: set-car!
  And set-cdr! CHANGE the list structure
  itself.

(set-car! x 1)
x --> (1 . b)
(set-cdr! x 2)
x --> (1 . 2)
```

Mutators

- **Procedures `set-car!` and `set-cdr!` are examples of mutators**
- **Mutators are procedures that change the contents of data structures**

Some comparisons

```
(define x '((a) b))
(define y '(c d))
(define z (cons y (cdr x)))
z --> ((c d) b)
x --> ((a) b)
(set-car! x y)
x --> ((c d) b)
(set-cdr! x y)
x --> ((c d) c d)
(set-car! (cdr x) '(a))
x --> (((a) d) (a) d)
```

Circular lists

```
(define x '(2))
(define y (cons 1 x))
(set-cdr! x y)
y --> (1 2 1 2 1 2 ...
```

[DrScheme is smart enough not to print the whole list]

```
(list-ref y 100) --> 1
(list-ref y 101) --> 2
```

Mutation of shared lists

```
(define x '(a b c))
(define y (cons x x))
(define z (cons x '(a b c)))
y --> ((a b c) a b c)
z --> ((a b c) a b c)
(set-cdr! (cdr x) ())
y --> ((a b) a b)
z --> ((a b) a b c)
```

Appending (with mutation)

```
; Exercise 3.12
; append! is a destructive version of append
; note x must not be null!
(define (append! x y)
  (set-cdr! (last-pair x) y)
  x)

; takes a non-null list and returns its last pair
(define (last-pair x)
  (if (null? (cdr x))
      x
      (last-pair (cdr x))))
```

Append! Examples

```
(define x (list 'a 'b))
(define y (list 'c 'd))
(define z (append x y))
> z
(a b c d)
> x
(a b)
> (define w (append! x y))
> w
(a b c d)
> x
(a b c d)
```

Copy-List

```
; takes a list and creates a copy (sort-of)
(define (copy-list x)
  (if (null? x)
      ()
      (cons (car x) (copy-list (cdr x)))))

(define x9 '((a b) c))
(define y9 (copy-list x9))
```

Copy-List Examples

```
> x9
((a b) c)
> y9
((a b) c)
> (set-car! (car x9) 'wow)
> x9
((wow b) c)
> y9
((wow b) c)
```

Deep-Copy-List

```
; does a deep-copy instead of just top-level
(define (deep-copy-list x)
  (cond ((null? x) ())
        ((pair? (car x))
         (cons (deep-copy-list (car x))
               (deep-copy-list (cdr x))))
        (else
           (cons (car x) (copy-list (cdr x))))))

(define x9-2 '((a b) c))
(define y9-2 (deep-copy-list x9-2))
```

Deep-copy-list Example

```
> x9-2
((a b) c)
> y9-2
((a b) c)
> (set-car! (car x9-2) 'wow)
> x9-2
((wow b) c)
> y9-2
((a b) c)
```

Message Passing – cons/car/cdr

```
(define (scons x y)
 (define (dispatch m)
  (cond ((eq? m 'scar) x)
       ((eq? m 'scdr) y)
       (else (error "Undefined operation -- SCONS"
  m))))
        dispatch)

(define (scar z) (z 'scar))
(define (scdr z) (z 'scdr))
```

Testing – Message Passing

```
(define t1 (scons 'a '(b)))

(check-expect (scar t1) 'a)

(check-expect (scdr t1) '(b))
```

Adding set-car! and set-cdr!?

```
(define (mcons x y)
 (define (set-x! v) (set! x v))
 (define (set-y! v) (set! y v))
 (define (dispatch m)
  (cond ((eq? m 'mcar) x)
       ((eq? m 'mcdr) y)
       ((eq? m 'mset-car!) set-x!)
      ((eq? m 'mset-cdr!) set-y!)
      (else
       (error "Undefined operation -- MCONS" m))))
 dispatch)
```

```
(define (mcar z) (z 'mcar))
(define (mcdr z) (z 'mcdr))

(define (mset-car! z new-value)
 ((z 'mset-car!) new-value)
 z)

(define (mset-cdr! z new-value)
 ((z 'mset-cdr!) new-value)
 z)
```

Drawing Environment Model?

```
> (define x (cons 1 2))
> (define z (cons x x))
> (define x (mcons 1 2))
> (define z (mcons x x))
> (mset-car! (mcdr z) 17)
#<procedure:dispatch>

> (mcar x)
17
```

Queues

```
(define q (make-queue))
(insert-queue! q 'a)        a
(insert-queue! q 'b)        a b
(delete-queue! q)           b
(insert-queue! q 'c)        b c
(insert-queue! q 'd)        b c d
(delete-queue! q)           c d
```

Queues

- Constructor: `(make-queue)`
- **Predicate:** `(empty-queue? <queue>)`
- **Selector:** `(front-queue <queue>)`
- **Mutators:**

```
(insert-queue! <queue> <item>)
(delete-queue! <queue>)
```

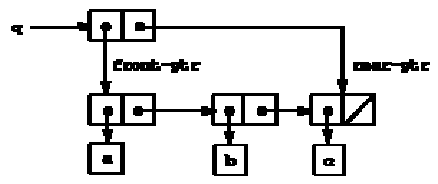
Queue Implementaton

- Straightforward queue representation: simple list structure
- Front-queue?
- Insert-queue?
- Problem?

Representation of queues

- Representation of queue is a list and a pair of pointers into that list
- Car of pair points to front of queue, where items are removed (pointer to list)
- Cdr of pair points to the end of the queue, where new items are added (pointer to last pair in list)

Queue Implementation



Implementation of queue

```
; makes an empty queue.  A queue is a pair with a front
; pointer and a rear pointer
(define (make-queue) (cons () ()))

; takes a queue and is #t if the queue is empty
(define (empty-queue? q) (null? (car q)))

; takes a queue and returns the front element
; or error if the queue is empty
(define (front-queue q)
  (if (empty-queue? q)
     (error "FRONT on empty queue" q)
     (caar q)))
```

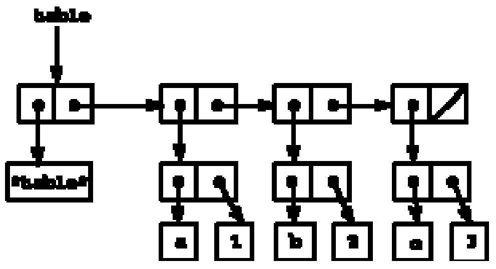
insert

```
; takes a queue and an item and changes the queue
; so as to add the new item (to the end)
(define (insert-queue! q item)
  (let ((new-pair (cons item ())))
    (cond ((empty-queue? q)
           (set-car! q new-pair)
           (set-cdr! q new-pair)
           q)
          (else
           (set-cdr! (cdr q) new-pair)
           (set-cdr! q new-pair)
           q))))
```

Delete

```
; takes a queue and changes it so as to delete the
; front element or creates an error if the queue
; is empty
(define (delete-queue! q)
  (cond ((empty-queue? q)
         (error "DELETE! on empty queue" q))
        (else
         (set-car! q (cdar q)))))
```

Association Table (one dimensional)



Association lists

```
((key1 . value1) (key2 . value2) ...)

; takes a key and an association list and
  returns
; the pair including the key in the list
(define (assoc key assoc-list)
  (cond ((null? assoc-list) #f)
        ((equal? key (caar assoc-list))
         (car assoc-list))
        (else
         (assoc key (cdr assoc-list)))))
```

One-dimensoinal tables

Use tagged association lists

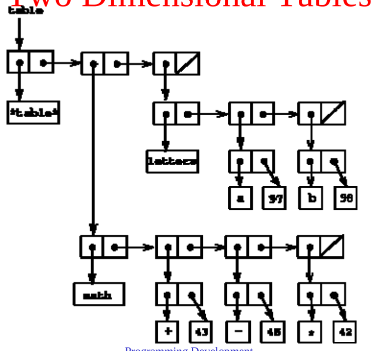
```
; makes an empty table
(define (make-table) (list '*table*))

; finds an entry in the table and returns
  the value
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        #f)))
```

Insert into table

```
; takes a key with a value and inserts
; it into the table table, value returned
  is
; the symbol done
(define (insert! key value table)
  (let ((record (assoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value)
                        (cdr table)))))
  'done)
```

Two Dimensional Tables



Two-dimensional tables

Use a table of tables, with tags as keys identifying each subtable

```
(define (make-table2) (list '*table*))

; takes two keys and a two dimensional table
; looks up the value associated with those
; two keys in the table
(define (lookup2 key1 key2 table)
  (let ((subtable (assoc key1 (cdr table))))
    (if subtable
        (let ((record (assoc key2
                             (cdr subtable))))
          (if record (cdr record) #f))
        #f)))
```

insert

```
; insert a new value into a table with two keys
; first check keys already exist
(define (insert2! key1 key2 val table)
  (let ((subtbl (assoc key1 (cdr table))))
    (if subtbl
        (let ((record (assoc key2
                             (cdr subtbl))))
          (if record
              (set-cdr! record val)
              (set-cdr! subtbl
                        (cons (cons key2 val)
                              (cdr subtbl)))))
```

continued

```
        (set-cdr! table
                  (cons (list key1
                              (cons key2 val))
                        (cdr table)))))
  'done)
```

Multidimensional tables (discrimination nets)

- Multidimensional table is actually a tree
- Each node of tree looks like
  (key value table table ...)
- **The value part of node is the value associated with the list of keys starting from the root of the tree (excluding '*table*) to this node**
- **If value = `#f`, there is no stored value for the list of keys**

Implementation

```
(define (make-table) (list '*table* #f))
(define (lookup key-list table)
  (if (null? key-list)
      (cadr table)
      (let ((subtbl (assoc (car key-list)
                           (cddr table))))
        (if subtbl
            (lookup (cdr key-list) subtbl)
            #f))))
```

insert

```
(define (insert! key-list value table)
  (if (null? key-list)
      (set-car! (cdr table) value)
      (let ((subtable (assoc (car key-list)
                             (cddr table))))
        (if subtable
            (insert! (cdr key-list)
                     value
                     subtable)
```

continued

```
(let ((subtable
       (list (car key-list) #f)))
  (set-cdr! (cdr table)
            (cons subtable
                  (cddr table)))
  (insert! (cdr key-list)
           value
           subtable)))))
```

Local tables

- Use message-passing technique
- Makes it possible for each table to have its own lookup and insert procedures

make-table

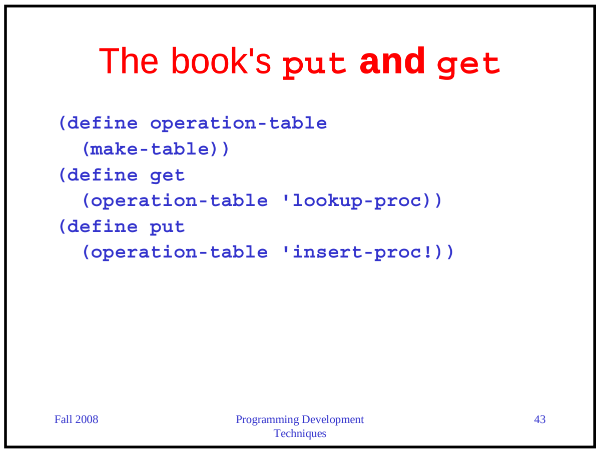
```
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key1 key2)
      (let ((subtbl
             (assoc key1 (cdr local-table))))
        (if subtbl
            (let ((record
                   (assoc key2 (cdr subtbl))))
              (if record (cdr record) #f))
            #f)))
```

local insert

```
(define (insert! key1 key2 val)
  (let ((subtbl
         (assoc key1 (cdr local-table))))
    if subtbl
       (let ((record
              (assoc key2 (cdr subtbl))))
         (if record
             (set-cdr! record val)
             (set-cdr! subtbl
                       (cons (cons key2 val)
                             (cdr subtbl)))))
```

continued

```
       (set-cdr! local-table
                 (cons (list key1
                             (cons key2
                                   val))
                       (cdr local-table)))))
  'done)
 (define (dispatch m)
   (cond ((eq? m 'lookup-proc) lookup)
         ((eq? m 'insert-proc!) insert!)
         (else (error "not TABLE op" m))))
 dispatch))
```



The book's `put` and `get`

```
(define operation-table
  (make-table))
(define get
  (operation-table 'lookup-proc))
(define put
  (operation-table 'insert-proc!))
```