

## Mutable data

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

## Mutation is assignment

```
(define (cons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else (error "Huh?" m))))
  dispatch)
```

## continued

```
(define (car z) (z 'car))
(define (cdr z) (z 'cdr))
(define (set-car! z new-val)
  ((z 'set-car!) new-val)
  z)
(define (set-cdr! x new-val)
  ((z 'set-cdr!) new-val)
  z)
```

## Queues

- Constructor: (make-queue)
- Predicate: (empty-queue? <queue>)
- Selector: (front-queue <queue>)
- Mutators:
  - (insert-queue! <queue> <item>)
  - (delete-queue! <queue>)

## 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)

## Implementation of queue

```
(define (make-queue) (cons () ()))  
(define (empty-queue? q) (null? (car q)))  
(define (front-queue q)  
  (if (empty-queue? q)  
      (error "FRONT on empty queue" q)  
      (caar q)))
```

## insert

```
(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

```
(define (delete-queue! q)  
  (cond ((empty-queue? q)  
        (error "DELETE! on empty queue" q))  
        (else  
         (set-car! q (cдар q)))))
```

## Association lists

```
((key1 . value1) (key2 . value2) ...)  
(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-dimensional tables

Use tagged association lists

```
(define (make-table) (list '*table*))  
(define (lookup key table)  
  (let ((record (assoc key (cdr table))))  
    (if record  
        (cdr record)  
        #f)))
```

## Insert into table

```
(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

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

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

## insert

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

## 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 ...)
- 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)
                          (caddr 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)
                              (caddr table))))
        (if subtable
            (insert! (cdr key-list)
                     value
                     subtable)
            (let ((subtable (subtable)))
              (insert! key-list value
                      subtable)))))))
```

## continued

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

## 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)))))
      (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!))
```