Data Structures

Section 3.3.2, 3.3.3, ...

- Thinking about data structures: stacks
- Queues
- Tables

A common data structure: Stacks

A stack is a data structure that behaves like a stack of papers

- You can make one
- You can put a new thing on top
- You can look at the top thing on the stack
- You can remove the top thing on the stack
- You can tell if a stack is empty

Stacks: operations and types

Make a stack

```
(make-stack) :  nothing -> stack of X
```

Put a new thing on top

```
(push <thing> <stack>) :  X, stack of X -> stack of X
```

Look at the thing on top

```
(top <stack>) :  stack of X -> X
```

Remove the thing on top

```
(pop <stack>) :  stack of X -> stack of X
```

Tell if stack is empty

```
(empty?  <stack>) :  stack of X -> boolean
```


Contracts: there is more data than types

Stack routines obey following contract:

```
(top (push thing stack)) = thing
(pop (push thing stack)) = stack
(empty? (make-stack)) = #t
(empty? (push thing stack)) = #f
```

Anything not in the contract is an error.
Errors can be checked (fail-stop) or unchecked (garbage-in-garbage-out).

Read contract as rewrite rules: is error to do `top` or `pop` if there are as many `pop`s as `push`es.

Implementing a stack as a list

With checked error...

Nothing much to do with mutation...

```
(define (make-stack) '())
(define empty? null?)
(define (top s)
  (if (empty? s)
      (error 'top-of-empty-stack)
      (car s)))
(define (pop s)
  (if (empty? s)
      (error 'pop-from-empty-stack)
      (cdr s)))
```

Our own error messages instead of

```
cdr: expects argument of type <pair>; given ()
```

Our errors should also be part of contract...

Queues

A queue has the similar operations as a stack, but they do slightly different things:

- It's like a line of people at a cafeteria
- You can join at the back end of the line,
- But you leave at the front end.
- FIFO – first in first out (maintains order)

Queue operations

Create an empty queue

```
(make-queue) :  nothing -> queue of X
```

Put a thing at the tail end of the queue

```
(insert <thing> <queue>):  X, queue of X -> queue of X
```

Delete the front of the queue

```
(delete <queue>) :  queue of X -> queue of X
```

Return the front of the queue

```
(front <queue>) :  queue of X -> X
```

Test whether queue is empty

```
(empty?  <queue>) :  queue of X -> boolean
```


Contract for Queues

(delete (insert x_n (insert x_{n-1} (... insert(x_0 (make-queue) ...))))) =

if n = 0 then error "Empty"

if n ≥ 1 then (insert x_n,(insert x_{n-1},(... (insert x_1 (make-queue)) ...))

(front (delete D times (insert I times (make-queue))))

= x_D if I $>$ D

error otherwise.

(empty? (delete D times (insert I times (make-queue))))

= #t if I $=$ D

false otherwise

delete takes result of oldest insertion out of queue.

front returns oldest element of queue that was not deleted.

empty? checks if every item that was inserted has since been deleted.

Implementing queues with mutation

A queue is a `cons` cell. The `car` points to the head of a list of items in the queue, if any. The `cdr` points to the tail of the queue. The queue elements are kept in a list.

```
(define (front-ptr queue) (car queue))
(define (rear-ptr queue) (cdr queue))
(define (set-front-ptr! queue item) (set-car! queue item))
(define (set-rear-ptr! queue item) (set-cdr! queue item))

(define (empty-queue? queue) (null? (front-ptr queue)))
(define (make-queue) (cons '() '()))
```


Implementing queues with mutation

```
(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))

(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))
```


Implementing queues with mutation

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         queue)))
```

Note: Scheme printer shows internal structure, not just what is in queue.

Working through queue operations

Show box and pointer diagrams, and what Scheme prints for

```
> (define q1 (make-queue))
> q1

> (insert-queue! q1 'a)

> (insert-queue! q1 'b)

> (delete-queue! q1)

> (delete-queue! q1)
```

Not production code!

What if instead of `'b` we inserted a very large object?

```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         (if (empty-queue? queue)        ;; clean up empty queue
             (set-rear-ptr! queue '()))
         queue)))
```


Mutable tables (dictionaries)

Operations

Make a table

```
(make-table) : nothing -> table of (X, Y)
```

Insert into a table

```
(insert! <key> <value> <table>) :
               X, Y, table of (X, Y) -> symbol
```

Look up in a table

```
(lookup <key> <table>) :
               X, table of (X, Y) -> Y or boolean
```

Contract: `lookup` gets result of most recent `insert` for key or `#f`.

Table implementation

```
(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        #f)))

(define (assoc key records)
  (cond ((null? records) #f)
        ((equal? key (caar records)) (car records))
        (else (assoc key (cdr records)))))
```


Table implementation

```
(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)))))
  'ok)

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

Table implementation

Try:

```
> (define table (make-table))

> (insert! 'c 3 table)

> (insert! 'b 2 table)

> (insert! 'a 0 table)

> (insert! 'a 1 table)
```


Table implementation

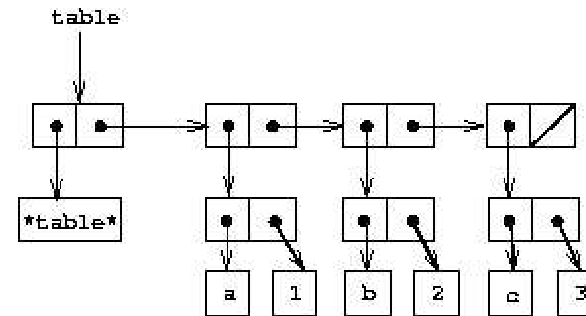


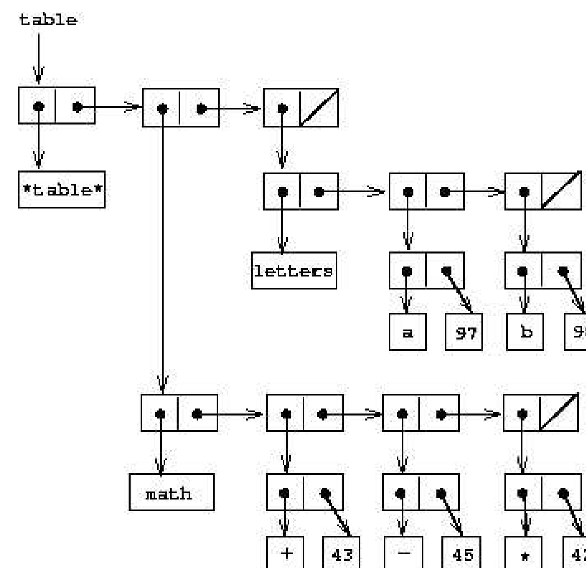
Table implementation problem: boolean values

Problem is in design, not implementation...

```
(define (make-table bogus-value)
  (list (cons '*table* bogus-value))

(define (lookup key table)
  (let ((record (assoc key (cdr table))))
    (if record
        (cdr record)
        (cdar table))))    ; user specified bogus value
```


2-d tables: Tables whose values are tables



2-d table implemetation

```
(define (lookup key-1 key-2 table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (cdr record)
              #f))
        #f)))
```


2-d table implemetation

```
(define (insert! key-1 key-2 value table)
  (let ((subtable (assoc key-1 (cdr table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value)
              (set-cdr! subtable
                        (cons (cons key-2 value)
                              (cdr subtable)))))
        (set-cdr! table
                  (cons (list key-1
                              (cons key-2 value))
                        (cdr table)))))
  'ok)
```


Don’t pass around the table

```
(define (make-table)
  (let ((local-table (list '*table*)))
    (define (lookup key-1 key-2)
      (let ((subtable (assoc key-1 (cdr local-table))))
        (if subtable
            (let ((record (assoc key-2 (cdr subtable))))
              (if record
                  (cdr record)
                  #f))
            #f)))
```


Still in scope of make-table

```
(define (insert! key-1 key-2 value)
  (let ((subtable (assoc key-1 (cdr local-table))))
    (if subtable
        (let ((record (assoc key-2 (cdr subtable))))
          (if record
              (set-cdr! record value)
              (set-cdr! subtable
                        (cons (cons key-2 value)
                              (cdr subtable)))))
        (set-cdr! local-table
                  (cons (list key-1
                              (cons key-2 value))
                        (cdr local-table)))))
  'ok)
```

Multiple procedures using local-table

```
(define (dispatch m)
  (cond ((eq? m 'lookup-proc) lookup)
        ((eq? m 'insert-proc!) insert!)
        (else (error "Unknown operation -- TABLE" m))))
dispatch))    ; end of make-table
```

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


Using a (1-d) table for memoization

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```


Using a (1-d) table for memoization

```
(define (memoize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previously-computed-result (lookup x table)))
        (or previously-computed-result
            (let ((result (f x)))
              (insert! x result table)
              result))))))
```


Using a (1-d) table for memoization

```
(define memo-fib
  (memoize (lambda (n)
             (cond ((= n 0) 0)
                   ((= n 1) 1)
                   (else (+ (memo-fib (- n 1))
                            (memo-fib (- n 2))))))))
```

Now imagine using an efficient implementation of dictionaries...

Th-th-th-that’s all folks...