Administrative Notes

Exam

- Most of them are graded
- Available tomorrow morning outside my door (DH 2065)
- Solutions will be available on web site

Homework 9 *(Ex. 32.2.1 — 32.2.8 in book)*

- Due Wednesday, April 10, 2002 in class
- 8 sub-problems
- Do them one a day and you will finish early
- Procrastinate and you will not finish

Graph Problems

Definition of a route map

→ Instance of a mathematical construct called a graph

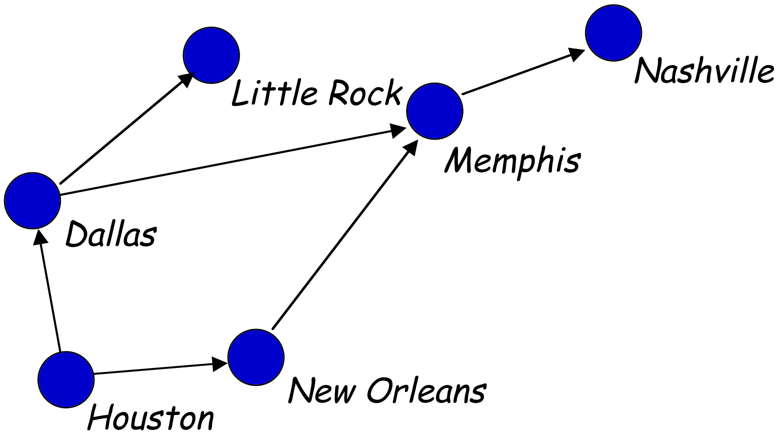
```
;; a city is a symbol

;; The information for a city is a structure
;;  (make-city-info  name  dests)
;; where name is a city and dests is a list of cities
(define-struct city-info (name dests))

;; a route-map is a list of city-info
;; We will use Scheme's built-in implementation of lists
```

Graph Problems

```
;; Example Route Map
(define routes
   (list    (make-city-info  ‘Houston (list ‘Dallas  ‘NewOrleans))
            (make-city-info  ‘Dallas (list ‘LittleRock  ‘Memphis))
            (make-city-info  ‘NewOrleans  (list ‘Memphis))
            (make-city-info  ‘Memphis  (list ‘Nashville))
    ))
```



Graph Problems

Developed a program find-flights

→ It used direct-cities to find neighbors in the route map

```
;; direct-cities : city route-map -> list of city
;; Purpose: find the cities reached by direct flights from the argument
(define (direct-cities from rm)
    (local [(define from-dests
                (filter (lambda (city)(symbol=? (city-info-name city) from)) rm))]
        (cond
            [(empty? from-dests) empty]
            [else (city-info-dests (first from-dests))]])
     ))
```

(direct-cities ‘Houston routes) ⇒ (list ‘Dallas ‘NewOrleans)

Graph Problems

Program find-flights to deal with city-info

```
;; find-flights: city city route-map -> list of city
;; Purpose: find a flight in rm from start to fini
(define (find-flights start finish rm)
    (cond
       [(symbol=? start finish) (list start)]     ;; trivial case
       [else (local [(define possible-route
                         (find-flights-for-list (direct-cities start rm) finish rm))]
                         (cond [(empty? possible-route)   empty]
                                  [else  (cons start possible-route)] ) ) ]
     ))
```

Uses find-flights-for-list to handle a list-of-city

```
(find-flights 'Houston  'LittleRock routes)
            ⇒  (list  'Houston  'Dallas  'LittleRock)
```

Graph Problems

Routine find-flights-from-list to deal with list-of-city

```
;; find-flights-for-list: list-of-city city route-map -> list of city
;; Purpose: finds a route from some city in the argument list to the
;;          city given as the singleton argument, using the route map
(define (find-flights-for-list aloc finish rm)
    (cond
      [(empty? aloc)   empty]
      [else
       (local [(define one-route (find-flights (first aloc) finish rm))]
             (cond
               [(empty? one-route) (find-flights-for-list (rest aloc) finish rm)]
               [else  one-route] )) ]
    ))
```

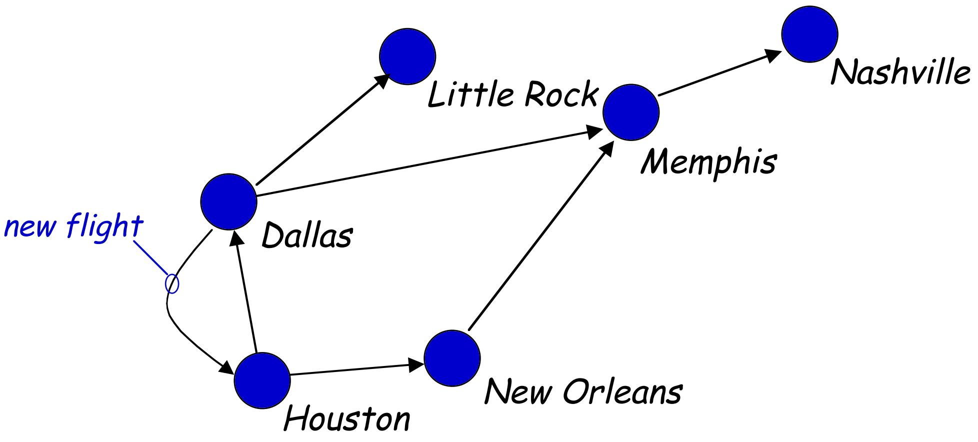
Find-flights

What happens if we add a cycle?

- Add a Dallas to Houston flight
- Now, (find-flights ‘Houston ‘Nashville new-routes) recurs indefinitely. (almost always a bad thing)

Graph Problems

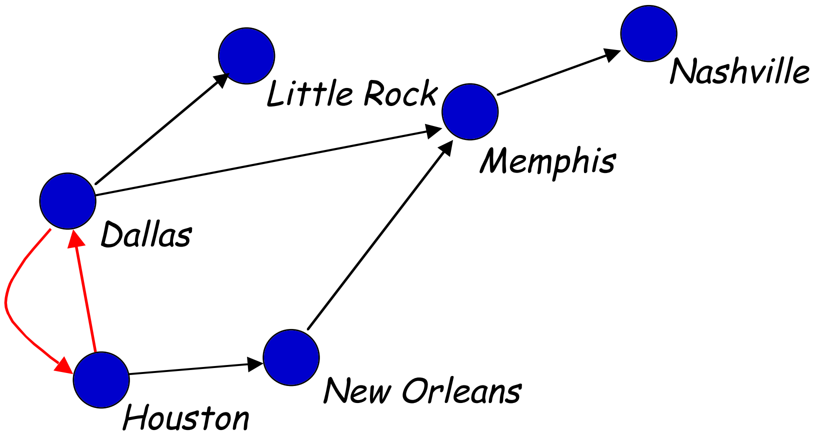
```
(define new-routes
   (list    (make-city-info  ‘Houston (list ‘Dallas  ‘NewOrleans))
            (make-city-info  ‘Dallas (list ‘Houston ‘LittleRock  ‘Memphis))
            (make-city-info  ‘NewOrleans  (list ‘Memphis))
            (make-city-info  ‘Memphis  (list ‘Nashville))
    ))
```



Graph Problems

(find-flights ‘Houston ‘LittleRock new-routes)

→ visits Dallas
→ visits Houston
→ visits Dallas
→ visits Houston, and so on …



Find-flights

What happens if we add a cycle?

- Add a Dallas to Houston flight
- Now, (find-flights ‘Houston ‘Nashville new-routes) recurs indefinitely. (almost always a bad thing)

What’s the real problem?

- Find-flights and find-flights-for-list have no history
  - → Those who ignore the past are doomed to repeat it
- Need to give them some institutional memory
  - → Add a parameter that contains cities already tested

Find-flights, take 2

```
;; find-flights: city city route-map list of city → list of city
;; Purpose: create a path of flights from start to finish or return
;;          empty
(define (find-flights start finish rm visited)
  (cond
    [(symbol=? start finish) (list start)]
    [(memq start visited)   empty]  ;; cut off this search path
    [else
          (local [(define possible-route
           (find-flights-for-list (direct-cities start rm) finish
                                rm  (cons start visited)))]
          (cond
             [(empty? possible-route)  empty]
             [else  (cons start possible-route)])) ] ))
```

Find-flights, take 2

```
;; find-flights-for-list: list-of-city city route-map list of city
;;                                → list-of-city
;; Purpose: finds a flight route from some city in the input list to the
;;          destination, or returns empty if no such route can be found.
(define (find-flights-for-list aloc finish rm visited)
  (cond
     [(empty? aloc)  empty]
     [else
(local [(define possible-route
                         (find-flights (first aloc) finish rm visited))]
                 (cond
                      [(boolean? possible-route)
                       (find-flights-for-list (rest aloc) finish rm visited)]
                     [else  possible-route]))]))
```

So, what is "visited"?

- We used "visited" to accumulate information
  - → Gathered over course of computation
  - → Used to ensure correct behavior
- We call such a parameter an *accumulator*

The Downside

- To let find-flights handle cycles, we changed its contract
- Can we avoid this? Sure …
  - → Wrap it up in a local
  - → We should hide direct-cities & find-flights-from-list, too

Find-flights —the last version

High-level overview

```
;; find-flights: city city route-map → list of city
;; Purpose: create a path of flights from start to finish or return
;;            empty
(define (find-flights start finish rm)
  (local [(define (direct-cities from rm)        ;; as before
                  … )
              (define (ffh start finish rm visited)  ;; accumulator version
                  … )
              (define (ffflh aloc finish rm visited) ;; accumulator version
                  … )]
           (ffh start finish rm empty)
    ))
```

This has original interface, guarantees right initial value to visited

Another Example

Reverse

- Simple programming problem
- Develop a program that consumes a list and produces a list containing the same elements, in reverse order

(reverse (list 1 2 3 4 5 6 7 8 9 10))

⇒ (list 10 9 8 7 6 5 4 3 2 1)

To begin, let’s write it using structural recursion

→ Start with the classic list template

Reverse

Version based on structural recursion

```
;; reverse:  list of alpha -> list of alpha
;; Purpose: returns a list containing the elements of the argument
;;            list, in reverse order
(define (reverse alist)
   (cond
      [(empty? alist)      empty ]
      [(cons?   alist)
            … (first alist) …
            … (reverse (rest alist)) … ]
    ))
```

Returns list-of-alpha suggests empty? clause returns empty

What to do with (first alist) and (reverse (rest alist)) ?

Reverse

Version based on structural recursion

```
;; reverse:  list of alpha -> list of alpha
;; Purpose: returns a list containing the elements of the argument
;;            list, in reverse order
(define (reverse alist)
   (cond
     [(empty? alist)   empty]
     [(cons? alist) (append (reverse (rest alist)) (cons (first alist) empty))]
   )
 )
```

Make (first alist) into a list for append …

Use append to paste sublists together

Reverse

What happens with (reverse (list 1 2 3))?

- Recall the rewriting rules
- Arguments evaluated before program's body
- Dives down into list and evaluates the end first

Reverse

What happens with (reverse (list 1 2 3 4))?

```
(reverse (list 1 2 3))       ;; look at the calls to add-to-end …
⇒(append (reverse (list 2 3 4)) (list 1))
    ⇒(append (append (reverse (list 3 4)) (list 2)) (list 1))
        ⇒(append (append (append (reverse (list 4)) (list 3))
            (list 2)) (list 1))
            ⇒(append (append (append (append (reverse empty)
                (list 4)) (list 3)) (list 2)) (list 1))
                ⇒(append (append (append (append empty (list 4))
                  (list 3)) (list 2)) (list 1))
            ⇒(append (append (append (list 4) (list 3)) (list 2)) (list 1))
        ⇒(append (append (list 4 3  (list 2)) (list 1))
    ⇒(append (list 4 3 2) (list 1))
⇒(list 4 3 2 1)
```

This code is "rev1" in lecture26.scm

This is a lot of work to reverse a list of three elements

Reverse

How costly is this?

- Think about what append does
  - → Walks down the list, rebuilding it
- Code invokes append for every element in the list
- N elements => N calls to append, each walking down the list
  - → First one walks whole list
  - → Next one walks list - 1
  - → Next one walks list -2

This takes time proportional to N^2 (Quadratic in length of original list)

This is a lot of work to reverse a list of three elements

Reverse

Can we improve this quadratic behavior?

- Reverse passes result of one recursive call to another recursive program — a *danger* signal for performance

```
;; reverse:  list of alpha -> list of alpha
;; Purpose: returns a list containing the elements of the argument
;;            list, in reverse order
(define (reverse alist)
   (cond
      [(empty? alist)   empty]
      [(cons? alist) (append (reverse (rest alist)) (cons (first alist) empty))]
   )
)
```

What if we used an accumulator?

Reverse

Using an accumulator

- New interface — second parameter is accumulator
- Start from list template

```
;; revacc:  list-of-alpha  list-of-alpha -> list-of-alpha
;; Purpose: …
(define (revacc alist acc)
   (cond
      [(empty? alist)           … ]
      [(cons?   alist)
           … (first alist) …
           … (revacc (rest alist) … ) ]
      ) )
```

Second parameter is acc, should add (first alist) to it

Start with cons? clause

Reverse

Using an accumulator

- New interface — second parameter is accumulator
- Start from list template

```
;; revacc:  list-of-alpha  list-of-alpha -> list-of-alpha
;; Purpose: …
(define (revacc alist acc)
   (cond
      [(empty? alist)          … ]
      [(cons?   alist)      (revacc (rest alist) (cons (first alist) acc) ) ]
      ) )
```

Now, what should empty? case return?

Answer: acc contains the reversed list

Reverse

Using an accumulator

- New interface — second parameter is accumulator
- Start from list template

```
;; revacc:  list-of-alpha  list-of-alpha -> list-of-alpha
;; Purpose: …
(define (revacc alist acc)
   (cond
      [(empty? alist)          acc ]
      [(cons?   alist)      (revacc (rest alist) (cons (first alist) acc) ) ]
      ) )
```

1. *Does it work? (to DrScheme)*
2. *How fast?   (next slide)*

Reverse

Using an accumulator

- New interface — second parameter is accumulator
- Start from list template

```
;; revacc:  list-of-alpha  list-of-alpha -> list-of-alpha
;; Purpose: …
(define (revacc alist acc)
   (cond
      [(empty? alist)           acc ]
      [(cons?   alist)      (revacc (rest alist) (cons (first alist) acc) ) ]
      ) )
```

This calls revacc once per list element

⇒ linear rather than quadratic number of calls

Much more efficient !

Reverse

The last step

- Fix the interface to ensure correct initial value to acc

```
;; reverse : list-of-alpha -> list-of-alpha
;; Purpose: …
(define (reverse alist)
  (local  [;; revacc:  list-of-alpha  list-of-alpha -> list-of-alpha
            ;; Purpose: …
            (define (revacc alist acc)
               (cond
                  [(empty? alist)           acc ]
                  [(cons?   alist) (revacc (rest alist) (cons (first alist) acc))]))]
           (revacc alist empty)
   ))
```