Administrative Notes

Homework 9

- Six days left
- Eight subparts
- If you haven't started, you are late

- John talked about "shared"
  - → Single copy of an object with multiple references to it
  - → Could not see this in "beginner" Scheme
  - → Reads like a local, with invented names

Accumulators on trees

Version derived from the methodology

```
;; largest: bnt -> number
;; Purpose: return the largest number in the bnt, or zero if
;;        the bnt is empty
(define (largest abnt)
   (cond
     [(empty? abnt)                              0]
     [(bnt?      abnt)     (max (bnt-num abnt)
                                 (largest (bnt-left abnt))
                                 (largest (bnt-right abnt)))]
   ))
```

Accumulators on trees

Accumulator version

```
(define (largest abnt)
  (local [;; acc holds largest number seen in nodes visited so far
          (define (lhelper atree acc)
             (cond [(empty? atree)    acc ]
                   [else (lhelper (bnt-left atree)
                                  (lhelper (bnt-right atree)
                                           (max (bnt-num atree) acc) ) ) ]
             )) ]
        (lhelper abnt -1 )  ))
```

Which is faster? ⇒ Dr. Scheme !

Moving on

How did we get to this point in the course?

- Remember JetSet Air?
- Remember find-flights?

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

Moving on

How did we get to this point in the course?

- Remember JetSet Air?
- Remember find-flights?

What happens if they succeed?

- Number of queries to server grows
- Number of people flying Houston to Nashville grows
- Much time spent computing known routes

There ought to be a better way

- *Preserve the answers we have already computed*

Teaching find-flights to "remember"

Sounds like a job for an accumulator

- Accumulators build up context and pass it along
- Can we formulate this problem with an accumulator?

No.

- Accumulator only has value during one chain of calls
  - → During one query to find-flights
- We need to keep the value(s) across multiple queries

We need something new

Memo functions

Abstract the problem

- Find-flights is too big for us to rewrite it 10 times
- Let's work with a simple algebraic function

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

What is g?
We don't need to care

Build a version of f that remembers

- Record arguments and results
- Check 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

Rewriting f

```
;; f:  number -> number
;; Purpose: invoke mystery function g on x squared
(define (f x)
   (local [(define prev-result (lookup x table))]
        (cond
            [(number? prev-result)  prev-result]
            [else
              (local [(define new-result (g (* x x )))]
                   (begin
                            ;; store new-result in table
                            result ))]
         )))
```


Memo functions

Rewriting f

```
;; lookup:  number list-of-result -> number or false
;; Purpose: returns answer if it is stored in the table, or
;;        false if it is not in the table
(define (lookup arg table)
   (local [(define answers
                      (filter (lambda(try)(= arg (result-arg try)))
                               table))]
            (cond
                [(empty? answers)   false]
                [else  (result-answer (first answers))]
             )))
```

In concept, this should work, but →

Memo functions

Rewriting f

```
;; f:  number -> number
;; Purpose: invoke mystery function g on x squared
(define (f x)
  (local [(define prev-result (lookup x table))]
      (cond
         [(number? prev-result)  prev-result]
         [else
           (local [(define new-result (g (* x x )))]
                (begin
                          ;; store new-result in table
                          result ))]
       )))
```

What is this?

Memo functions

Need a way to add a result to table

- We have seen nothing in Scheme that does this
- Need a new Scheme construct

```
;;  set! takes 2 arguments, an object & an expression
;;        It changes the definition of the object to refer to the
;;        value produced by evaluating the expression
(set!  table  (cons (make-result x new-result) table))
```

- Creates a new result and puts it add the head of the list
- Makes table refer to that list

Memo functions

Now, f looks like

```
;; f:  number -> number
;; Purpose: invoke mystery function g on x squared
(define (f x)
   (local [(define prev-result (lookup x table))]
       (cond
          [(number? prev-result)  prev-result]
          [else
            (local [(define new-result (g (* x x )))]
                 (begin
                          (set!  table  (cons (make-result x new-result) table))
                          result ))]
        )))
```


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

Memo functions

Thinking about COMP 210 philosophy

- If set! makes such a momentous difference in our execution model, should we use it?
  - → Yes, but with some caution
  - → We should demarcate its use with a comment
- What's with the exclamation point
  - → It demarcates set!
- Shouldn't we hide <u>table</u> and <u>lookup?</u>
  - → Of course
- Why do all these slides keep saying "Memo functions"
  - → This technique is called a memo-function implementation

Information hiding

We should hide table & lookup in a local

- Where do we define table?

```
;; f:  number -> number
(define (f x)
  (local [(define prev-result (lookup x table))]
     (cond
        [(number? prev-result)  prev-result]
        [else (local [(define table empty)
                          (define new-result (g (* x x )))]
                  (begin
                     (set!  table  (cons (make-result x new-result) table))
                     result ))]
     )))
```

This cannot work

Information hiding

We should hide table & lookup in a local

```
;; f:  number -> number
(define (f x)
  (local [ (define table empty)
           (define prev-result (lookup x table))]
     (cond
       [(number? prev-result)  prev-result]
       [else (local [(define new-result (g (* x x )))]
               (begin
                 (set!  table  (cons (make-result x new-result) table))
                 result ))]
  )))
```

This will never work. Each call to f creates a new table.

It cannot possible remember results of earlier computations!

Information hiding

We need a local that survives across invocations

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

Defines f, not (f x)

Function can see table because of local's rewriting rules. (Nothing outside the function can see it!)

Result of the local is a function

Net result: f is a function with hidden, persistent state stored in the object table

Information hiding

We need a local that survives across invocations

```
;; f:  number -> number
(define f
   (local [ (define table empty) ]
       (lambda(x)
          (local [(d[illegible] table))]
            (cond
                [(nu[illegible]esult]
                [else (local [(define new-result (g (* x x )))]
                      (begin
                         (set!  table  (cons (make-result x new-result) table))
                         result ))]
        )))
    )
)
```

WHAT?

Information hiding

We need a local that survives across invocations

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

See lecture 22, slide 6

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"*