COMP 210, FALL 2000
Lecture 30: Memo Functions

Reminders:

1. Missionaries & Cannibals alert: Due 12 April 2000.

Review

We looked at accumulators on binary trees, and saw that a straight-forward formulation of the accumulator version of sumtree would leave behind pending context. To eliminate this, we formulated a version that (essentially) threaded its way through the tree. To do this, it passed the result of summing the right subtree as the accumulator to the computation that summed the left subtree. This has the properties that (1) it works, (2) it avoids the pending context.

Motivation *[WARNING: the material covered in this lecture requires the ADVANVED language level in DrScheme.]*

Let's go back to find-flights—the program that we wrote for JetSet Airlines to find routes between cities in their *route map*. Here is a cleaned-up version of the program (to slide)

```
;; find-flights : city city → list of city or false
;; creates a path of flights from start to finish
(define (find-flights start finish)
 (local [(define rm ...)
    (define (find-helper start finish visited)
      (cond [(symbol=? start finish) (list start)]
            [(memq start visited) false]
            [else (local
                    [(define possible-route
                       (find-flights/list (direct-cities start)
                                          finish
                                          (cons start visited)))]
                    (cond [(boolean? possible-route) false]
                          [else (cons start possible-route)]))])]))

    ;; direct-cities : city  route-map → list of city
    ;; return list of all cities in route map with direct flights from given city
    (define (direct-cities from-city)
      (local [(define from-city-info
                (filter (lambda (c) (symbol=? (city-info-name c) from-city))
                        rm))]
        (cond [(empty? from-city-info) empty]
              [else (city-info-fly-to (first from-city-info))])))
    ;; find-flights/list:  list of city  city  list of city  → list of city or false
    ;; finds a flight route from some city in the input list to the
    ;; destination, or returns false if no such route exists
    (define (find-flights/list loc finish visited)
      (cond [(empty? loc) false]
                  [else (local [(define possible-route
                                            (find-helper (first loc) finish visited))]
                                (cond [(boolean? possible-route)
                                            (find-flights/list (rest loc) finish visited)]
                                           [else possible-route]))]))]
   (find-helper start finish empty)))
```

Unfortunately, as JetSet Airlines has grown, they've discovered that this program takes an inordinate amount of time. It rediscovers a route each time that the question is asked; thus, if three customers want to fly from Houston to Nashville, the program repeats the path-finding operation three times. Since the route-map rarely changes, they have asked the simple question—can't we make find-flights return those pre-computed routes? (They've become much more sophisticated users since they became dedicated Schemers.)

This new version of find-flights should "remember" any pre-computed routes. If not route has been computed, it should operate as before and search through the route map to discover if one exists. [In some sense, this program ``learns'' about the route map and its structure over time. If this were a course in artificial intelligence, we would focus on the program's ability to become "smarter" —measured as "faster" in this context—as it is used. But, machine learning is a subject for another day …]

On first thought, this sounds like a job for accumulators. However, that won't work, because the accumulator maintains a record (or memory) of the results of a computation over the duration of that computation. If we type (find-flights 'Houston 'Nashville), the program builds up an accumulator that records its visits to the records for 'Dallas, 'NewOrleans, 'LittleRock, 'Memphis, and (finally) 'Nashville. When it returns the route that it found, however, it discards that accumulator. Furthermore, the accumulator only records the fact that we visited those cities, without keeping around the original query. If we preserved the accumulator (somehow) across multiple invocations of the program, find-flights would never find a second route that ran through 'Memphis. Its search would be cut off as soon as it hit a city that had been searched in some previous query.

What we need is a mechanism that lets find-flights record its behavior (and results) over a longer period of time—across multiple calls to the main program.

Find-flights is a big, complex example. Modifying it at the board would take a significant amount of time and would focus on the details of find-flights rather than the critical principles that we're exploring. So, let's work with an abstraction of the problem. Consider a program **f** that consumes a number and produces a number.

```
(define (f x)
  (g (* x x)))
```

Here, **g** is another Scheme program. It doesn't matter what **g** does, but let's assume that whatever it does is expensive. Can we build a version of **f** that remembers the calls to **g** and their results? First, we need a way to represent the information passed to g and returned by g.

```
;; a result is a structure
;;   (make-result  x-arg  answer)
;; where x-arg and answer are both numbers
(define-struct result (x-arg ans))
```

Next, we need a way to refer to the list of previously computed results. We can introduce a variable called **table** and set it to the initial list of previously computed results.

```
(define table empty)
```

Now, we need to rewrite **f** to check **table** for a previously computed result before invoking **g**.

```
;; f : number → number
;; Purpose:  uses the mystery function g on (* x x)
(define (f x)
  (local [(define prev-result (lookup x table))]
         (cond [(number? prev-result) prev-result]
               [else (local [(define x-result (g (* x x)))]
                            ;; store result in table
                            result)])))

;; lookup : number  list of result → number or false
;; Purpose returns answer stored in table for arg, or false if no value stored
(define (lookup arg table)
  (local [(define prev-ans
             (filter (lambda (res) (= arg (result-x-arg res))) table))]
          (cond [(empty? prev-ans) false]
                [else (result-ans (first prev-ans))])))
```

There's one major problem with this program, in the middle of **f**. It has this comment in about storing the result in table. To accomplish this, we need a new Scheme operator (or keyword).

> ***set!*** takes two arguments, a variable name and an expression.
> It changes the definition of the variable to refer to the new expression.

That is, ***set!*** has the effect of redefining the variable so that it contains the result of evaluating the expression. Thus, we can replace the comment in **f** with the line

```
(set! table  (cons (make-result x x-result) table))
```

which has the effect of adding the new result to the head of the list. It evalutes the expression, so it performs the make-result to create a new result structure and the cons that makes the old version of the table into the tail of that list. Then, it sets the value of the variable table to this new list.

[There's one additional problem inside **f** –- we have the **set!** expression followed immediately by the expression **result**. So far in class, we haven't written anything like this. However, in the advanced language setting, this will work and have the expected behavior. The two expressions will be evaluated, in order, and the resulting value will be the value of the expression evaluated last –- the one that occurs last in textual order. Thus, the result of evaluating the first expression is not returned. Until set!, we had no reason to do this, since we couldn't "see" the effect of the first evaluation!]

There's one additional problem with this version of **f**. Since **f** is the only program that uses **table**, we should hide table inside **f** (standard information hiding, or need-to-know, doctrine). Our COMP 210 experience tells us that this is a job for local.

```
;; f : number → number
;; Purpose:  uses the mystery function g on (* x x)
(define (f x)
  (local [(define prev-result (lookup x table))]
         (cond [(number? prev-result) prev-result]
               [else (local [(define table empty)
                             (define x-result (g (* x x)))]
                       (set! table  (cons (make-result x x-result) table))
                       result)])))
```

Unfortunately, this doesn't work as we would hope. When we call (f 3), it creates an instance of table, creates the result of (g (* 3 3)), uses cons to prepend it to the empty table. Then, it loses the name for that particular instance of table (when it finishes with the local expression. When we call (f 4), it will create its own instance of table, and repeat the process.

We need a way to hide the value of table inside **f** before consuming the value of any particular x. We can do this with lambda.

```
(define f                  ←———————————
  (local [(define table empty)]
         (lambda (x)
             (local [(define prev-result (lookup x table))]
                    (cond [(number? prev-result) prev-result]
                          [else (local [(define x-result (g (* x x)))]
                                   (set! table  (cons (make-result x x-result)
                                                      table))
                                   result)]))) ))
```

This says define f, not define (f x)

How does this work? DrScheme creates an instance of the variable table and rewrites the lambda expression with that instance inside the lambda expression. The lambda expression traps that name. Since f refers to the lambda expression every time, it gets the same instance of table, every time.

This is a complicated aspect of lambda expressions. They can, in effect, capture the values of variables defined outside of their scope. Thus, we call these lambda expressions *closures* because the close (in the set theoretic sense) over the values of variables deifned outside of their scope.

Summary

1. Sometimes, it is useful to have a program that can remember values computed in earlier invocations. We call this process memoization. It makes sense when the computation is substantially more expensive than theprocess required to go back and look up the value in the record of earlier computations.
2. To remember values, we need a mechanism to change the value associated with a variable. The Scheme keyword **set!** accomplishes this task.
3. We can use lambda to create a variable that will persist over multiple calls to a function. We call this use of lambda *creating a closure*.