CSE 341
Lecture 20

Mutation; memoization

slides created by Marty Stepp
http://www.cs.washington.edu/341/
Mutation and mutability
Mutating variables

(set! name expression)

• Unlike ML, in Scheme all top-level bindings are mutable!

> (define x 3) ; int x = 3;
> (set! x 5) ; x = 5;

– Legal, but changing bound values is generally discouraged.
– Convention: Any procedure that mutates ends with !
Mutations and environment

• What does the following code do to the environment?

```
(define x 3)
(define (f n) (+ n x))
(set! x 5)
(define x 17)
(define (g k) (* k x))
(set! x 8)
```

```
<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>(to be set on call)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>symbol</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>(to be set on call)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

System libraries...

Global environment
define vs. set!

• What is the difference between these two procedures?

(\text{define } x 3)
(\text{define } (f \ k) \quad \text{(define } (g \ k) \quad (\text{define } x 5) \quad (\text{set! } x 5) \quad (* x k)) \quad (* x k))

• both return the same thing for any given call, but...
  • \(f\) defines a local \(x\) and uses it; global \(x\) is unchanged
  • \(g\) mutates the global \(x\) and uses its new value
• Use let to create a private mutable variable:

```scheme
(define incr null) ; stub for procedure
(define get null)  ; stub for procedure
(let ((n 0))
  (let ((n 0))
    (set! incr (lambda (i) ; replace stubs
        (set! n (+ n i)))))) ; n += i
    (set! get (lambda () n)))) ; return n
```

```scheme
> (get)               > (incr 8)
0                    > (get)
> (incr 3)            11
> (get)               > n
3
```

* reference to undefined ...
Lists and equality

(define L1 '(2 4 7))
(define L2 '(2 4 7))
(define L3 L2)

- Scheme lists are linked structures, as in ML
  - two lists declared with the same value are separate lists
  - one list declared to be another list will be a reference to that same list object in memory (shared)

(We didn't care much about this distinction in ML... why?)
Recall: Testing for equality

\[
\begin{align*}
\text{(define L1 '(2 4 7))} \\
\text{(define L2 '(2 4 7))} \\
\text{(define L3 L2)}
\end{align*}
\]

- (eq? expr1 expr2) ; reference/ptr comparison
- (eqv? expr1 expr2) ; compares values/numbers
- (= expr1 expr2) ; like eqv; numbers only
- (equal? expr1 expr2) ; deep equality test

- Which are true for L1 and L2? L3?
(define L4 (cons 8 L1))
(define L5 (append '(5 6) L2))

• Which of the following are true?

(eq? L4 '(8 2 4 7))
(equal? L5 '(5 6 2 4 7))
(equal? L1 (cdr L4))
(eq? L1 (cdr L4))
(equal? L2 (cddr L5))
(eq? L2 (cddr L5))
(eq? L3 (cddr L5))
Mutating lists

(set-car! list expr)
(set-cdr! list expr)

• these procedures mutate the contents of lists (!)
• any reference to that list will see the change

(set-car! L1 9)
(set-cdr! L2 '(6 5 1))

(set-car! and set-cdr! are disabled in "Pretty Big" Scheme)
(mcons expr mutableList)
(mcar mutableList)
(mcdr mutableList)
(set-mcar! mutableList expr)
(set-mcdr! mutableList expr)

• Scheme has a separate mutable list type that you can use to explicitly create a list that can be modified
  ▪ can build up a list by calling mcons with null
  ▪ mutable lists display on the interpreter with { ... }

(mutable lists are not allowed on our homework)
Memoization
Exercise

• Define a procedure `count-factors` that accepts an integer parameter and returns how many factors it has.

  ▪ Possible solution:
    
    ```scheme
    (define (count-factors n)
      (length (filter (lambda (x) (= 0 (modulo n x)))
                   (range 1 n)))
    ```

  ▪ Problem: slow for large values; "forgets" after each call
    
    ```scheme
    > (count-factors 999990)
    48 ; takes 4-5 seconds
    > (count-factors 999990)
    48 ; takes 4-5 seconds, AGAIN!
    ```
Memoization

• **memoization**: Optimization technique of avoiding re-calculating results for previously-processed function calls.
  - often uses a **cache** of previously computed values

• General algorithmic pattern:

```python
function compute(param):
    if I have never computed the result for this value of param before:
        compute the result for param.
        store (param, result) into cache data structure.
        return result.
    else  // I have computed this result before; don't re-compute
        look up (param, result) in cache data structure.
        return result.
```
Memoization w/ association lists

- a natural structure to cache prior calls is a *map*
  - recall: Scheme implements maps as *association lists*

> (define phonebook (list '(Marty 6852181) '(Stuart 6859138) '(Jenny 8675309)))

> (assoc 'Stuart phonebook)
(Stuart 6859138)

> (cdr (assoc 'Jenny phonebook)) ; get value 8675309

- we'll remember results of past calls to count-factors by storing them in a (mutating) association list
defining count-factors code

; cache of past calls as (n . count) pairs; initially empty
(define cache null)

(define (count-factors n)
  (define (divides? x) (= 0 (modulo n x)))
  ; look up n in the cache (see if we computed it before)
  (let ((memory (assoc n cache)))
    (if memory  ; if n is in cache, return cached value.
      (cdr memory)
      ; else, count the factors...
      (let ((count (length (filter divides? (range 1 n)))))
        ; store them into the cache...
        (set! cache (cons (cons n count) cache))
        ; and return the result.
        count))))
Problem: undesired global cache

- the cache is a global variable
  - can be seen (or modified!) by other code

- solution: define it locally
  - to do this properly, we must define count-factors using an inner helper and local inner cache
  - count-factors is set equal to its own helper
    - bizarre, but ensures proper closure over the local cache
Improved count-factors code

(define count-factors
  (let ((cache null)) ; Local cache; initially empty
    ; inner helper that has access to the local cache
    (define (helper n)
      (define (divides? x) (= 0 (modulo n x)))
      ; look up n in the cache (see if we computed it before)
      (let ((memory (assoc n cache)))
        (if memory ; if n is in cache, return cached value.
            (cdr memory)
            ; else, count the factors...
            (let ((count (length (filter divides? (range 1 n)))))
              ; store them into the cache...
              (set! cache (cons (cons n count) cache))
              ; and return the result.
              count))))
    helper)) ; return helper; sets count-factors equal to helper