Class 6: Efficiency in Scheme

SI 413 - Programming Languages and Implementation

Dr. Daniel S. Roche

United States Naval Academy

Fall 2011

Objects in Scheme

We can use closures and mutation to do OOP in Scheme!

Recall the counter example from last class:

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ 1 count))
      (display count)
      (newline))))
```

To add methods, the (first) argument to the returned lambda will be the *name* of the method we want to apply.

Objects in Scheme

We can use closures and mutation to do OOP in Scheme!

More sophisticated counter:

```
(define (make-counter-obj)
  (let ((count 0))
    (lambda (command)
      (cond [(symbol=? command 'get) count]
            [(symbol=? command 'inc)
             (set! count (+ 1 count))]
            [(symbol=? command 'reset)
             (set! count 0)]))))
```

The object now has three methods: `get`, `inc`, and `reset`.

Built-in Data Structures

Scheme has some useful built-in data structures:

- Arrays (called "vectors").

```
(define A (make-vector 5))
(vector-set! A 3 'something)
(vector-ref A 3) ; produces 'something
(vector-ref A 5) ; error: out of bounds
```

- Hash tables

```
(define H (make-hash))
(hash-set! H 2 'something)
(hash-set! H (list 20 #f) 'crazy!)
(hash-ref H '(20 #f)) ; produces 'crazy!
(hash-ref H '(bad)) ; error: no key (bad)
```


Efficiency in Scheme: Fibonacci numbers

Recall the problem of computing Fibonacci numbers from lab 1.

```
(define (fib n)
  (if (<= n 1)
      1
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

Why is this function so slow?

Memoization in Scheme

Memoization is remembering the results of previous function calls.

Why is functional programming *perfect* for memoization?

In Scheme we can use vectors or hashes to provide this functionality.

Some languages have built-in memoization.

Memoizing Fibonacci

Here's how we might memoize the Fibonacci function:

```
(define fib-hash (make-hash))

(define (fib-memo n)
  (let ((saved (hash-ref fib-hash n 'unset)))
    (if (equal? saved 'unset)
        (let ((res (if (<= n 1)
                       1
                       (+ (fib-memo (- n 1))
                          (fib-memo (- n 2))))))
          (hash-set! fib-hash n res)
          res)
        saved)))
```


Stack space in recursive calls

Recursive calls can use a lot of memory, even when the results are puny.

```
;; Sum of squares from 1 to n
(define (ssq n)
    (if (= n 0)
        0
        (+ (sqr n) (ssq (- n 1)))))
```

Why does `(ssq 4000000)` run out of memory?

Stack space in recursive calls

This function does the same thing, but takes an *extra argument* that serves as an accumulator.

```
;; Sum of squares using tail recursion
(define (ssq-better n accum)
  (if (= n 0)
      accum
      (ssq-better (- n 1)
                  (+ (sqr n) accum))))
```

Now `(ssq-better 4000000 0)` actually works!

Tail recursion

The second version worked because there was no need to make a stack of recursive calls.

A function is *tail recursive* if its output expression in every recursive case is only the recursive call.

In Scheme, this means the recursive call is **outermost** in the returned expression.

`ssq-better` is better because it is tail recursive!

Tail recursion for Fibonacci

To implement tail recursion we usually make a helper function:

```
(define (fib-tail-helper n i fib-of-i fib-of-i+1)
  (if (= i n)
      fib-of-i
      (fib-tail-helper
       n
       (+ i 1)
       fib-of-i+1
       (+ fib-of-i
          fib-of-i+1))))
```

The main function then becomes:

```
(define (fib-tail n) (fib-tail-helper n 0 1 1))
```