MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1999

Recitation – Wednesday, February 24

1. Abstracting Common Patterns

```
(define (map proc seq)
  (if (null? seq)
      nil
      (cons (proc (car seq))
            (map proc (cdr seq)))))

(define (accumulate op init seq)
  (if (null? seq)
      init
      (op (car seq)
        (accumulate op init (cdr seq)))))
```

```
(define (filter pred seq)
  (if (null? seq)
      nil
      (let ((rest (filter pred (cdr seq))))
         (if (pred (car seq))
             (cons (car seq) rest)
             rest))))
```

2. List Warm-Up

Write the following procedures using map, filter, and accumulate (no recursion!).

Description	Example	Definition

`square-list`: squares all the elements in a list	`(square-list (list 1 2 3))`
	`==> (1 4 9)`
		`(define (square-list x)`
		`   (map square x)`
		`)`

`apply-all`: takes a list of procs and an item, and returns a list of applying each procedure to the item.	`(apply-all`
	`  (list sqrt square fact) 4)`
	`    ==> (2 16 24)`
		`(define (apply-all ops x)`
		`  (map (lambda (op) (op x)) ops)`
		`)`

`length`: length of a list	`(length (list 1 2 3))`
	`==> 3`
		`(define (length x)`
		`  (accumulate`
		`    (lambda (a b) (+ 1 b))`
		`    0 x)`
		`)`

`mean`: mean of a list	`(mean (list 1 2 3 4 5))`
	`==> 3`
		`(define (mean x)`
		`  (/ (accumulate + 0 x)`
		`      length x))`
		`)`

`range`: range of a list (max elt minus min elt)	`(range (list 1 2 3 4 5))`
	`==> 4`
		`(define (range x)`
		`  (- (accumulate max (car x) x)`
		`     (accumulate min (car x) x))`
		`)`

`element?`: returns true if elt is an element of x	`(element? (list 1 2 3 4) 3)`
	`==> #t`
		`(define (element? x elt)`
		`  (not (null?`
		`        (filter (lambda (y) (= elt y))`
		`                x)))`
		`)`

`map`: map a function over a list of elements	`(map double (list 1 2 3))`
	`==> (2 4 6)`
		`(define (map proc x)`
		`  (accumulate`
		`    (lambda (a b) (cons (proc a) b))`
		`    nil`
		`   x)`
		`)`

3. Depth of a Tree

Write a function `depth` that takes a tree and returns the maximum depth of the tree. Note that this is equivalent to the maximum number of parenthesis open at any given time when scheme prints the tree. For example, `(depth (list 1 (list (list 2) 3) (list 4))) ==> 3`

```
(define (depth tree)
  (cond ((pair? tree)
         (max
          (+ 1 (depth (car tree)))
          (depth (cdr tree))))
        (else 0))
)
```

Now write `depth` using map and accumulate...

```
(define (depth tree)
  (if (not (pair? tree))
      0
      (+ 1 (accumulate max
           0
           (map depth tree))))
)
```

4. Deep Reverse

So far, we've been working on lists, while we've ignored the elements of the list. What does the following return?
`(reverse (list 1 (list 2 3) (list 4 5 6)))`

Write a function `deep-reverse` that when called on the above tree will reverse all the elements.
`(deep-reverse (list 1 (list 2 3) (list 4 5 6))) ==> ((6 5 4) (3 2) 1)`

```
(define (deep-reverse x)
  (define (aux x ans)
    (cond ((null? x) ans)
   ((not (pair? x)) x)
           (else
    (aux (cdr x) (cons (deep-reverse (car x)) ans)))))
  (aux x nil)
)
```

Now write `deep-reverse` using map...

```
(define (deep-reverse x)
  (if (not (pair? x))
      x
      (map deep-reverse (reverse x)))
)
```

6. Sieve of Eratosthenes

Let's walk through the process of the sieve of Eratosthenes one more time..

	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30

```
(define (sieve lst)
  (if (null? lst)
      nil
      (cons (car lst)
            (sieve (filter (lambda (x) (not (divisible? x (car lst))))
                           (cdr lst))))))
```