

Computer Science I

Professor Tom Ellman
Lecture 20

Increment Each Number on a Nested List

```
(define increment-nlist (lambda (x) ...?...))
```

Welcome to Dr. Scheme.

```
> (increment-nlist '(1 2 3 4 5))
```

```
(2 3 4 5 6)
```

```
> (increment-nlist '(1 (2 3)(4 (5))))
```

```
(2 (3 4)(5 (6)))
```

```
> (increment-nlist '())
```

```
()
```

INCREMENT-NLIST

```
(define increment-nlist  
  (lambda (x)  
    (cond ((null? x) '())  
          ((number? x) (+ x 1))  
          (else (cons (increment-nlist (car x))  
                      (increment-nlist (cdr x)))))))
```

Count the Numbers on a List Lying between Two Bounds

```
(define count-between (lambda (lst low high) ...?...))
```

Welcome to Dr. Scheme.

```
> (count-between '(5 1 2 3 4) 2 4)
```

```
3
```

```
> (count-between '(1 2 3 4 5) 7 9)
```

```
0
```

```
> (count-between '() 2 4)
```

```
0
```

Count-Between

```
(define count-between  
  (lambda (lst low high) (cb-helper lst low high 0)))
```

```
(define cb-helper
```

```
(lambda (lst low high acc)
```

```
(cond ((null? lst) acc)
```

```
      ((and (>= (car lst) low) (<= (car lst) high))
```

```
        (cb-helper (cdr lst) low high (+ 1 acc)))
```

```
      (else (cb-helper (cdr lst) low high acc))))))
```

Count-Between

```
(define count-between
```

```
(lambda (lst low high)
```

```
(letrec ((cb-helper (lambda (lst acc)
```

```
                  (cond ((null? lst) acc)
```

```
                        ((and (>= (car lst) low) (<= (car lst) high))
```

```
                          (cb-helper (cdr lst) (+ 1 acc)))
```

```
                        (else (cb-helper (cdr lst) acc))))))
```

```
(cb-helper lst 0)))
```

Square Constructor Function

```
(define make-square
  (lambda (left bottom side)
    (list 'square left bottom side)))
```

Square Selector Functions

```
(define square-left (lambda (s) (cadr s)))
(define square-bottom (lambda (s) (caddr s)))
(define square-side (lambda (s) (caddr s)))
(define square-width (lambda (s) (square-side s)))
(define square-height (lambda (s) (square-side s)))
(define square-right
  (lambda (s) (+ (square-left s) (square-width s))))
(define square-top
  (lambda (s) (+ (square-bottom s) (square-height s))))
(define square-area
  (lambda (s) (* (square-side s) (square-side s))))
```

Storing Redundant Information

```
(define make-square
  (lambda (left bottom side)
    (list 'square
          left
          bottom
          side
          (+ left side)
          (+ bottom side)
          (* side side))))
```

Fast Access to Stored Redundant Information

```
(define square-left (lambda (square) (cadr square)))
(define square-bottom (lambda (square) (caddr square)))
(define square-side (lambda (square) (caddr square)))
(define square-width (lambda (square) (caddr square)))
(define square-height (lambda (square) (caddr square)))
(define square-right (lambda (square) (car (cddddr square))))
(define square-top (lambda (square) (cadr (cddddr square))))
(define square-area (lambda (square) (caddr (cddddr square))))
```

Add Up the Absolute Values of Numbers on a List

```
(define sum-absolute-values (lambda (lst) ...?...))
```

Welcome to Dr. Scheme.

```
> (sum-absolute-values '(-5 1 2 -3 4))
```

15

```
> (sum-absolute-values '(5 1 2 3 4))
```

15

```
> (sum-absolute-values '())
```

0

Sum-Absolute-Values

```
(define sum-absolute-values
  (lambda (lst) (apply + (map abs lst))))
```

```
(define abs (lambda (x) (if (< x 0) (- x) x)))
```

Do at Least N Elements of a List Satisfy a Predicate?

```
(define at-least-n? (lambda (lst n p?) ...?...))
```

Welcome to Dr. Scheme.

```
> (at-least-n? '(1 2 3 4 5) 3 even?)
```

```
#f
```

```
> (at-least-n? '(1 2 3 4 5) 3 odd?)
```

```
#t
```

```
> (at-least-n? '(1 2 3 4 5) 0 pair?)
```

```
#t
```

AT-LEAST-N?

```
(define at-least-n?  
  (lambda (lst n p?)  
    (or (<= n 0)  
        (and (not (null? lst))  
              (or (and (p? (car lst))  
                       (at-least-n? (cdr lst) (- n 1) p?))  
                  (and (not (p? (car lst)))  
                       (at-least-n? (cdr lst) n p?))))))))
```

Are at Least M Elements of a List Greater than N?

```
(define at-least-n-above-m? (lambda (lst n m) ...?...))
```

Welcome to Dr. Scheme.

```
> (at-least-n-above-m? '(1 2 3 4 5) 3 2)
```

```
#t
```

```
> (at-least-n-above-m? '(1 2 3 4 5) 2 4)
```

```
#f
```

```
> (at-least-n? '(1 2 3 4 5) 0 6)
```

```
#t
```

AT-LEAST-N-ABOVE-M?

```
(define at-least-n-above-m?  
  (lambda (lst n m)  
    (at-least-n? lst n (lambda (x) (> x m)))))
```

Given Coefficients A, B and C, Return the Quadratic Function

```
(define quadratic-function (lambda (a b c) ...?...))
```

Welcome to Dr. Scheme.

```
> (define x-squared-plus-one (quadratic-function 1 0 1))
```

```
> (map x-squared-plus-one '(1 2 3 4 5))
```

```
(2 5 10 17 26)
```

QUADRATIC-FUNCTION

```
(define quadratic-function  
  (lambda (a b c)  
    (lambda (x) (+ (* a x x) (* b x) c))))
```

Given a List of Predicates, Return a Predicate...

```
(define satisfies-one-predicate (lambda (plist) ...?...))  
Welcome to Dr. Scheme.  
> (define pe? (satisfies-one-predicate (list positive? even?)))  
> (pe? -4)  
#t  
> (pe? 3)      ((satisfies-one-predicate (list p1? ... pn?) x)  
#t             is true if at least one of (p1? x) ... (pn? x) is  
> (pe? -7)      true.  
#f
```

SATISFIES-ONE-PREDICATE

```
(define satisfies-one-predicate  
  (lambda (plist)  
    (lambda (x)  
      (there-exists? plist (lambda (p?) (p? x))))))
```