

## Computer Science I

Professor Tom Ellman  
Lecture 6

## Multiply a Number by Itself N Times

(define power (lambda (base exponent) ...?...))

Welcome to DrScheme

```
> (power 2 0)
1
> (power 2 1)
2
> (power 2 2)
4
> (power 2 3)
8
```

## Constructing Solution to Larger Problem from Solution to Smaller One

$$b^n = b * \underbrace{(b * b * b \dots b)}_{b^{n-1}}$$

$$b^n = b * b^{n-1}$$

## Recursive Definition of POWER

```
(define power
  (lambda (base exponent)
    (if (= exponent 0)
        1
        (* base (power base (- exponent 1))))))
```

## Make a List with N Copies of Something

(define replicate (lambda (item n) ...?...))

Welcome to DrScheme

```
> (replicate 'foo 0)
()
> (replicate 'foo 1)
(foo)
> (replicate 'foo 2)
(foo foo)
> (replicate 'foo 3)
(foo foo foo)
```

## Constructing Solution to Larger Problem from Solution to Smaller One

$$\underbrace{\begin{array}{c} \text{n Copies} \\ (a \ a \ a \ a \ a \ \dots \ a) \\ (a \ a \ a \ a \ \dots \ a) \end{array}}_{\text{n-1 Copies}}$$

## Recursive Definition of REPLICATE

```
(define replicate
  (lambda (item n)
    (if (<= n 0)
        '()
        (cons item (replicate item (- n 1))))))
```

## Remove the Negative Numbers from a List

```
(define remove-negatives (lambda (lst) ...?...))
```

```
Welcome to DrScheme
> (remove-negatives '())
()
> (remove-negatives '(-7))
()
> (remove-negatives '(7))
(7)
> (remove-negatives '(-3 2 4 -5 6))
(2 4 6)
```

## Constructing Solution to Larger Problem from Solution to Smaller One

Case 1: Car = p (positive)  
Answer for Cdr = (p1 p2 p3 p4 ... pN)  
Return: (p p1 p2 p3 p4 ... pN)

Case 2: Car = n (negative)  
Answer for Cdr = (p1 p2 p3 p4 ... pN)  
Return: (p1 p2 p3 p4 ... pN)

## Recursive Definition of REMOVE-NEGATIVES

```
(define remove-negatives
  (lambda (lst)
    (cond ((null? lst) '())
          ((negative? (car lst)) (remove-negatives (cdr lst)))
          (else (cons (car lst) (remove-negatives (cdr lst)))))))
```

## Remove all Occurrences of a Symbol from a List of Symbols

```
(define remove-all (lambda (item lst) ...?...))
```

```
Welcome to DrScheme
> (remove-all 'a '())
()
> (remove-all 'a '(z a z))
(z z)
> (remove-all 'a '(a z a))
(z)
> (remove-all 'a '(a a a))
()
```

## Constructing Solution to Larger Problem from Solution to Smaller One

Case 1: lst = (x0 x1 x2 ... xN) where x0 = item  
Answer for Cdr = (y1 y2 ... yN)  
Return: (y1 y2 ... yN)

Case 2: lst = (x0 x1 x2 ... xN) where x0 ≠ item  
Answer for Cdr = (y1 y2 ... yN)  
Return: (x0 y1 y2 ... yN)

## Recursive Definition of REMOVE-ALL

```
(define remove-all
  (lambda (item lst)
    (cond ((null? lst) '())
          ((equal? item (car lst)) (remove-all item (cdr lst)))
          (else (cons (car lst) (remove-all item (cdr lst)))))))
```

## Take Every Second Element of a List

```
(define every-second-element (lambda (lst) ...?...))
```

```
Welcome to DrScheme
> (every-second-element '())
()
> (every-second-element '(a))
()
> (every-second-element '(a b))
(b)
> (every-second-element '(1 2 3 4 5 6 7))
'(2 4 6)
```

## Constructing Solution to Larger Problem from Solution to Smaller One

```
Argument = (a1 b1 a2 b2 a3 b3 ... aN bN)
Car = a1
Cadr = b1
Cddr = (a2 b2 a3 b3 ... aN bN)
Answer for Cddr = (b2 b3 ... bN)
Return: (b1 b2 b3 ... bN)
```

## Recursive Definition of EVERY-SECOND-ELEMENT

```
(define every-second-element
  (lambda (lst)
    (cond ((null? lst) '())
          ((null? (cdr lst)) '())
          (else (cons (car (cdr lst))
                      (every-second-element (cdr (cdr lst))))))))
```

## Checking Whether a List of Numbers is Increasing

```
(define increasing? (lambda (lst) ...?...))
```

```
Welcome to DrScheme
> (increasing? '())
#t
> (increasing? '(7))
#t
> (increasing? '(5 8 11 54))
#t
> (increasing? '(1 2 3 2 1))
#f
> (increasing? '(1 2 3 3 4 5))
#f
```

## Constructing Solution to Larger Problem from Solution to Smaller One

```
Argument = (x y1 y2 y3 y4 ... yN)
```

Case 1:  $x < y1$

Return: #t if  $(y1 y2 y3 y4 \dots yN)$  is increasing, otherwise return #f.

Case 2:  $x \geq y1$

Return: #f

## Recursive Definition of Increasing?

```
(define increasing?  
  (lambda (lst)  
    (cond ((null? lst) #t)  
          ((null? (cdr lst)) #t)  
          ((< (car lst) (car (cdr lst))) (increasing? (cdr lst)))  
          (else #f))))
```

## Recursive Definition of Increasing?

```
(define increasing?  
  (lambda (lst)  
    (or (null? lst)  
        (null? (cdr lst))  
        (and (< (car lst) (cadr lst))  
              (increasing? (cdr lst))))))
```

## Checking Whether a Symbol is a Member of a List of Symbols

```
(define member? (lambda (item lst) ...?...))
```

Welcome to DrScheme

```
> (member? 'a '())  
#f  
> (member? 'a '(z a z))  
#t  
> (member? 'a '(a z a))  
#t  
> (member? 'a '(x y z))  
#f
```

## Recursive Definition of MEMBER?

```
(define member?  
  (lambda (item lst)  
    (cond ((null? lst) #f)  
          ((equal? item (car lst)) #t)  
          (else (member? item (cdr lst))))))
```

## Recursive Definition of MEMBER?

```
(define member?  
  (lambda (item lst)  
    (and (not (null? lst))  
         (or (equal? item (car lst))  
             (member? item (cdr lst))))))
```