Problem: Make a list with \( n \) copies of something

\[
\text{(define \texttt{replicate} }
\begin{array}{l}
\text{(lambda (item n) )} \\
\text{...?...)}
\end{array}
\)

\[
> \text{(replicate 'foo 0)} \\
'(())
\]

\[
> \text{(replicate 'foo 1)} \\
'(foo)
\]

\[
> \text{(replicate 'foo 2)} \\
'(foo foo)
\]

\[
> \text{(replicate 'foo 3)} \\
'(foo foo foo)
\]
Recursive definition of replicate

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

Problem: Remove the negative numbers from a list

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

> (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: first = p (positive)
Answer for rest = (p₁ p₂ p₃ p₄ ... pₙ)
Return: (p₁ p₂ p₃ p₄ ... pₙ)

Case 2: first = n (negative)
Answer for rest = (p₁ p₂ p₃ p₄ ... pₙ)
Return: (p₁ p₂ p₃ p₄ ... pₙ)

Recursive definition of remove-negatives

(define remove-negatives
  (lambda (lst)
    (cond
      ;; Base case: reached the end of lst
      (null? lst) '()
      ;; Recursive case 1:
      ;; First element is negative; skip it.
      ((negative? (first lst))
        (remove-negatives (rest lst)))
      ;; Recursive case 2:
      ;; First element is positive; include it.
      (else
        (cons (first lst)
          (remove-negatives (rest lst)))))))
Problem: Removing specified elements from a list

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

> (remove-all 'a '())
()  
> (remove-all 'a '(z a z))
(z z)  
> (remove-all 'a '(a z a))
(z)  
> (remove-all 'a '(a a a))
()  

Case 1: \( \text{lst} = (x_0 \ x_1 \ x_2 \ldots \ x_n) \) where \( x_0 = \text{item} \)
Answer for rest \( = (y_1 \ y_2 \ldots \ y_n) \)
Return: \( (y_1 \ y_2 \ldots \ y_n) \)

Case 2: \( \text{lst} = (x_0 \ x_1 \ x_2 \ldots \ x_n) \) where \( x_0 \neq \text{item} \)
Answer for rest \( = (y_1 \ y_2 \ldots \ y_n) \)
Return: \( (x_0 \ y_1 \ y_2 \ldots \ y_n) \)

Recursive definition of remove-all

(define remove-all
  (lambda (elt lst)
    (cond
      ;; Base case:
      ;; lst is empty; return ()
      ((null? lst) '())
      ;; Recursive case 1:
      ;; first elt of lst is supposed to be removed.
      ;; (equal? elt (first lst))
      ;; (remove-all elt (rest lst)))
      ;; Recursive case 2:
      ;; first elt of lst should NOT be removed
      (else
        (cons (first lst)
          (remove-all elt (rest lst))))))))

(tester '(remove-all 5 '(1 5 2 5 3 5 4 5)))
(tester '(remove-all 'a '(a b a c a d a z)))
Recursive definition of remove-first

\[
\text{define remove-first} = \lambda (\text{elt lst}) \{
\begin{array}{ll}
\text{cond} & \\
\text{;; Base case 1:} & \text{lst is empty; return ()} \\
& (\text{null? lst}) '() \\
\text{;; Base case 2:} & \text{first elt of lst is supposed to be removed; we're done!} \\
& (\text{equal? elt (first lst)} \\
& \text{rest lst}) \\
\text{;; Recursive case:} & \text{first elt of lst should NOT be removed} \\
& \text{else} \\
& \text{cons (first lst)} \\
& \text{remove-first elt (rest lst))})})
\end{array}
\]
\]

Take every second element of a list

\[
\text{define every-second-element} = \lambda (\text{lst}) ...?
\]
\[
> (\text{every-second-element '()} '())
> (\text{every-second-element '(a)} '(') \\
> (\text{every-second-element '(a b)} '('b) \\
> (\text{every-second-element '(1 2 3 4 5 6 7)} '2 4 6)
\]

Constructing solution to larger problem from solution to smaller one

Argument = \((a_1 \ b_1 \ a_2 \ b_2 \ a_3 \ b_3 \ ... \ a_n \ b_n)\)

\[
\text{first} = a_1 \\
\text{rest} = (b_1 \ a_2 \ b_2 \ a_3 \ b_3 \ ... \ a_n \ b_n)
\]

rest of rest (cddr) = \((a_2 \ b_2 \ a_3 \ b_3 \ ... \ a_n \ b_n)\)

Answer for rest of rest (cddr) = \((b_2 \ b_3 \ ... \ b_n)\)

Return: \((b_1 \ b_2 \ b_3 \ ... \ b_n)\)

Recursive definition of every-second-element

\[
\text{define every-second-element} = \lambda (\text{lst}) \{
\begin{array}{ll}
\text{cond} & \\
\text{;; Base case 1:} & \text{lst is empty; return ()} \\
& (\text{null? lst}) '() \\
\text{;; Base case 2:} & \text{first elt of lst is supposed to be removed; we're done!} \\
& (\text{null? (rest lst)} '()) \\
\text{else} & \text{cons (first (rest lst))} \\
& \text{every-second-element (rest (rest lst))))})
\end{array}
\]
\]
Recursive predicates

When writing predicates that test whether something holds of all elements of a list – or at least one element of a list – you can often make the recursive function call with an and or or.

Problem: Write a function that returns #t if every element of a list is a positive number.

```
(define all-positive?
  (lambda (lst)
    ;; Base case: List is empty; return #t
    (or (null? lst)
      ;; Recursive case: first element positive...
      (and (number? (first lst))
           (positive? (first lst))
           ;; ... and so are the rest.
           (all-positive? (rest lst))))))
```

(tester '(all-positive? '(3 8 2 4 9 6)))
(tester '(all-positive? '(3 8 2 -4 9 6)))
We can modify all-positive? to return #t if any of the numbers in the list is positive.

```scheme
(define any-positive?
  (lambda (lst)
    ;; Base case: List is empty; return #f
    (and (not (null? lst))
      ;; Recursive case: first element is positive...
      (or (and (number? (first lst))
                 (positive? (first lst)))
          ;; ... or one of the rest of them is.
          (any-positive? (rest lst))))))
(tester '(any-positive? '(-3 -2 #t #f () -8)))
(tester '(any-positive? '(-3 -2 #t #f 4 -8)))
```

Acknowledgments
This lecture incorporates material from:

Tom Ellman
Luke Hunsberger