Sorting a List

Multiple Complex Inputs

Natural Numbers
Implement `sort-list`, which takes a list of numbers and returns a sorted list of the same numbers.
Sorting a List

Multiple Complex Inputs

Natural Numbers
Multiple Complex Arguments

Implement `append-lists`, which takes two lists of numbers and returns a list with all of the numbers from the first list followed by all of the numbers from the second list.

Implement `parallel-sum`, which takes two lists of numbers (of the same length) and returns a list of sums.

Implement `merge-lists`, which takes two *sorted* lists of numbers and returns a sorted list with all of the numbers.
Multiple Complex Arguments

Implement **append-lists**, which takes two lists of numbers and returns a list with all of the numbers from the first list followed by all of the numbers from the second list

Implement **parallel-sum**, which takes two lists of numbers (of the same length) and returns a list of sums

Implement **merge-lists**, which takes two sorted lists of numbers and returns a sorted list with all of the numbers

; append-lists : list-of-num list-of-num -> list-of-num

(check-expect (append-lists empty empty empty) empty)

(check-expect (append-lists (list 1 3 5) (list 0 4 6))
             (list 1 3 5 0 4 6))
Multiple Complex Arguments

Implement **append-lists**, which takes two lists of numbers and returns a list with all of the numbers from the first list followed by all of the numbers from the second list.

Implement **parallel-sum**, which takes two lists of numbers (of the same length) and returns a list of sums.

Implement **merge-lists**, which takes two sorted lists of numbers and returns a sorted list with all of the numbers.

; parallel-sum : list-of-num list-of-num -> list-of-num
(check-expect (parallel-sum empty empty empty) empty)

(check-expect (parallel-sum (list 1 3 5) (list 0 4 6))
(list 1 7 11))
Multiple Complex Arguments

Implement **append-lists**, which takes two lists of numbers and returns a list with all of the numbers from the first list followed by all of the numbers from the second list

Implement **parallel-sum**, which takes two lists of numbers (of the same length) and returns a list of sums

Implement **merge-lists**, which takes two sorted lists of numbers and returns a sorted list with all of the numbers

; merge-lists : list-of-num list-of-num -> list-of-num

(check-expect (merge-lists empty empty empty) empty)

(check-expect (merge-lists (list 1 3 5) (list 0 4 6))
  (list 0 1 3 4 5 6))
Multiple Complex Arguments

Implement **append-lists**, which takes two lists of numbers and returns a list with all of the numbers from the first list followed by all of the numbers from the second list.

Implement **parallel-sum**, which takes two lists of numbers (of the same length) and returns a list of sums.

Implement **merge-lists**, which takes two sorted lists of numbers and returns a sorted list with all of the numbers.

; func : list-of-num list-of-num -> list-of-num

What template do we use for a function for two lists?
Multiple Complex Arguments

Sometimes a complex argument is “along for the ride,” so use the template for the other argument

```
(check-expect (append-lists (list 1 3 5) (list 0 4 6))
             (list 1 3 5 0 4 6))
```

```
(define (append-lists al bl)
  (cond
   [(empty? al) ...]
   [(cons? al)
     ... (first al)
     ... (append-lists (rest al) bl) ...]]))
```
Multiple Complex Arguments

Sometimes the arguments are exactly the same shape, so use essentially the one-argument template

```scheme
(check-expect (parallel-sum (list 1 3 5) (list 0 4 6))
  (list 1 7 11))
```

```scheme
(define (parallel-sum al bl)
  (cond
   [(empty? al) ...]
   [(cons? al)
    ... (first al) ... (first bl)
    ... (parallel-sum (rest al) (rest bl)) ...]))
```
Multiple Complex Arguments

Sometimes you have to consider all possible combinations, so use a template that considers all combinations

```
(check-expect (merge-lists (list 1 3 5) (list 0 4 6))
  (list 0 1 3 4 5 6))
```

```
(define (merge-lists al bl)
  (cond
    [(and (empty? al) (empty? bl)) ...]
    [(and (empty? al) (cons? bl))
      ... (first bl) ... (merge-lists al (rest bl)) ...]
    [(and (cons? al) (empty? bl))
      ... (first al) ... (merge-lists (rest al) bl) ...]
    [(and (cons? al) (cons? bl))
      ... (first al) ... (first bl)
      ... (merge-lists (rest al) bl)
      ... (merge-lists al (rest bl))
      ... (merge-lists (rest al) (rest bl)) ...]))
```
Sorting a List

Multiple Complex Inputs

Natural Numbers
Numbers to Generate Lists

Implement \texttt{create-list}, which takes a non-negative integer \(n\) and produces a list of numbers from \(n\) to 0, inclusive

\begin{verbatim}
; create-list : num -> list-of-num
(check-expect (create-list 3) (list 3 2 1 0))
(check-expect (create-list 0) (list 0))
\end{verbatim}

The template for \texttt{num} isn’t much help:

\begin{verbatim}
(define (func-for-num n)
  ...)
\end{verbatim}

But \texttt{create-list} actually takes a \textit{natural number}
Natural Numbers

; A nat is either
; - 0
; - (add1 nat)

Examples:

0

(add1 0)

(add1 (add1 (add1 0)))

These examples have shortcuts

0, 1, and 3

but the long forms correspond to the template
Template for Natural Numbers

; A nat is either
;   - 0
;   - (add1 nat)

(define (func-for-nat n)
  (cond
   [(zero? n) ...]
   [else ... (func-for-nat (sub1 n)) ...]))

(define (create-list n)
  (cond
   [(zero? n) (list 0)]
   [else (cons n (create-list (sub1 n)))]))
Generating the List the Other Way

Implement `create-up-list`, which takes a non-negative integer `n` and produces a list of numbers from 0 to `n` inclusive.

```scheme
; create-up-list : num -> list-of-num

(check-expect (create-list 3) (list 0 1 2 3))
(check-expect (create-list 0) (list 0))

(define (create-up-list n)
  (cond
   [(zero? n) (list 0)]
   [else
    ... n
    ... (create-up-list (sub1 n)) ...]]))

; uh oh... can't cons onto recur result
Using Subtraction to Count Up

\[
(\text{define } (\text{create-up-list } n) \\
\quad (\text{create-up-to-n-list } n n))
\]

; Creates a list with d elements before n
(\text{define } (\text{create-up-to-n-list } d n) \\
\quad (\text{cond} \\
\qquad [(\text{zero? } d) (\text{list } n)] \\
\qquad [\text{else} \\
\qquad\quad (\text{cons } (- n d) \\
\qquad\qquad (\text{create-up-to-m-list } (\text{sub1 } d) n))]))

... or replace \text{d} with \text{m} = (+ \text{d} \text{n})

As \text{d} goes down, \text{m} goes up...
Counting Up Directly

\[
\text{(define (create-up-list n)}
\text{ (create-m-to-n-list 0 n))}
\]

; Creates a list from m to n
\[
\text{(define (create-m-to-n-list m n)}
\text{ (cond}
\text{ [ (= m n) (list n)]}
\text{ [else}
\text{ (cons m}
\text{ (create-m-to-n-list (add1 m) n))])})
\]

Use the stepper to see how it works

Similar ideas work for counting by fives, counting down to 20, etc.