Sorting a List

- Multiple Complex Inputs
- Natural Numbers
Implement \texttt{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.

; list-of-num list-of-num -> list-of-num

(check-expect (append-lists '() '()) '())

(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.

```
; list-of-num list-of-num -> list-of-num
(check-expect (parallel-sum '() '()) '())
(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.

; list-of-num list-of-num -> list-of-num

(check-expect (merge-lists '() '()) '())

(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

; 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

\[
\text{(check-expect (\text{parallel-sum} (\text{list} 1 3 5) (\text{list} 0 4 6)) (\text{list} 1 7 11))}
\]

\[
\text{(define (\text{parallel-sum} \text{al} \text{bl})}
\]
\[
\text{(cond}
\]
\[
[\text{(empty? \text{al}) ...}]
\]
\[
[\text{(cons? \text{al})}
\]
\[
... (\text{first \text{al}) ... (first \text{bl})}
\]
\[
... (\text{parallel-sum (rest \text{al}) (rest \text{bl})} ...]]))
\]
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 **create-list**, which takes a non-negative integer \( n \) and produces a list of numbers from \( n \) to 0, inclusive

\[
; \text{num} \rightarrow \text{list-of-num}
\]

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

(check-expect (create-list 0) (list 0))
Numbers to Generate Lists

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

; num -> list-of-num

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

(check-expect (create-list 0) (list 0))

The template for `num` isn’t much help:

(define (func-for-num n)
  ...)

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

; num -> list-of-num

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

(check-expect (create-list 0) (list 0))

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

(define (func-for-num \texttt{n})
  ...)

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)))
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)) ...]))
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

```
; num -> list-of-num
(check-expect (create-list 3) (list 0 1 2 3))
(check-expect (create-list 0) (list 0))
```
Generating the List the Other Way

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

\[
; \text{num} \rightarrow \text{list-of-num}
\]

\[
(\text{check-expect} \ (\text{create-list} \ 3) \ (\text{list} \ 0 \ 1 \ 2 \ 3))
\]

\[
(\text{check-expect} \ (\text{create-list} \ 0) \ (\text{list} \ 0))
\]

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

; uh oh... can't cons onto recur result
(define (create-up-list n)
  (create-up-to-n-list n n))

; Creates a list with d elements before n
(define (create-up-to-n-list d n)
  (cond
   [(zero? d) (list n)]
   [else
    (cons (- n d)
      (create-up-to-n-list (sub1 d) n))])))
Using Subtraction to Count Up

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

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

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

As \( d \) goes down, \( m \) goes up...
Counting Up Directly

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

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

(define (create-up-list n)
  (create-m-to-n-list 0 n))

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

Use the stepper to see how it works
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.