- **Sorting a List**
- **Multiple Complex Inputs**
- **Natural Numbers**

Sorting Lists

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)

(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)

(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)

(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

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

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

```
; create-list : 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)
  ...)
```

But **`create-list`** actually takes a ***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

```
; 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

```
(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-m-list (sub1 d) n))]))
```

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

As **d** goes down, **m** goes up...

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

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