We’ve seen that we can represent a point in space using a `posn` structure:

```
(make-posn 10 3)
```

has x-coordinate 10 and y-coordinate 3 and

```
(make-posn 5 20)
```

has x-coordinate 5 and y-coordinate 20.
If we have two points, we might want to compute their distance 

\[ d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \] 

(as you need to for the extra credit for Assignment 3):

\[
(sqrt (+ (sqr (- (posn-x (make-posn 10 3))
           (posn-x (make-posn 5 20))))
   (sqr (- (posn-y (make-posn 10 3))
           (posn-y (make-posn 5 20))))))
\]
We don’t want to write out those `make-posn` expressions every time we want to work with those two points, so we can give them names:

```scheme
(define P1 (make-posn 10 3))
(define P2 (make-posn 5 20))
```
**Naming** is a basic but essential form of abstraction. Now we can use our names and the values are substituted for them:

\[
(sqrt (+ (sqr (- (posn-x (make-posn 10 3))
(posn-x (make-posn 5 20))))
(sqr (- (posn-y (make-posn 10 3))
(posn-y (make-posn 5 20))))))
\]

becomes

\[
(sqrt (+ (sqr (- (posn-x P1)
(posn-x P2)))
(sqr (- (posn-y P1)
(posn-y P2))))))
\]
But every time we have a new set of points and we want to compute their distance, we need to rewrite (or copy-and-paste) that expression and change the variable names.

```scheme
(define P3 (make-posn 6 300))
(define P4 (make-posn 20 0))

(sqrt (+ (sqr (- (posn-x P3) (posn-x P4)))
         (sqr (- (posn-y P3) (posn-y P4))))
```
We automate this process of duplicating code and changing values by writing *functions*, e.g.,

```
(define (dist posn1 posn2)
  (sqrt (+ (sqr (- (posn-x posn1)
                (posn-x posn2)))
         (sqr (- (posn-y posn1)
                (posn-y posn2))))))
```

A function is an abstraction over the individual expressions we could write, where we give names to the values that differ between each expression.

Those expressions can now be replaced with calls to the function:

```
(dist P1 P2)
(dist P3 P4)
```
We can follow the same functional approach when we have *functions* that differ in one spot.
Big fish

A function that gets the big fish (> 5 lbs):

;; [List-of Number] -> [List-of Number]
(define (big l)
  (cond [(empty? l) '()]
        [(cons? l)
          (if (> (first l) 5)
              (cons (first l) (big (rest l)))
              (big (rest l)))]))

(check-expect (big '()) '())
(check-expect (big '(7 4 9)) '(7 9))
Big fish

A function that gets the *big* fish (> 5 lbs):

```
;; [List-of Number] -> [List-of Number]
(define (big l)
  (cond [(empty? l) '()] [(cons? l)
    [(if (> (first l) 5)
      (cons (first l) (big (rest l)))
      (big (rest l)))]()))
```

(check-expect (big '()) '())
(check-expect (big '(7 4 9)) '(7 9))

Suppose we also need to find *huge* fish…
Huge fish

A function that gets the huge fish (> 10 lbs):

;; [List-of Number] -> [List-of Number]
(define (huge l)
  (cond [(empty? l) '()]
        [(cons? l)
          (if (> (first l) 10)
              (cons (first l) (huge (rest l)))
              (huge (rest l))))])

(check-expect (huge '()) '())
(check-expect (huge '(17 4 9)) '(17))
Huge fish

A function that gets the huge fish (> 10 lbs):

;;; [List-of Number] -> [List-of Number]
(define (huge l)
  (cond [[(empty? l) '()]
        [(cons? l)
          (if (> (first l) 10)
            (cons (first l) (huge (rest l)))
            (huge (rest l)))]))

(check-expect (huge '()) '())
(check-expect (huge '(17 4 9)) '(17))

How do you suppose I made this slide?
Huge fish

A function that gets the huge fish (> 10 lbs):

;; [List-of Number] -> [List-of Number]
(define (huge l)
  (cond [(empty? l) '()][(cons? l)
    [(if (> (first l) 10)
      (cons (first l) (huge (rest l)))
      (huge (rest l)))]))

(check-expect (huge '()) '())
(check-expect (huge '(17 4 9)) '(17))

How do you suppose I made this slide?

Copy and paste!
The trouble with copy-and-paste

(define (big l)
  (cond [((empty? l)) '()] [(cons? l)
    (if (> (first l) 5)
      (cons (first l) (big (rest l)))
      (big (rest l)))]))

(define (huge l)
  (cond [((empty? l)) '()] [(cons? l)
    (if (> (first l) 10)
      (cons (first l) (huge (rest l)))
      (huge (rest l)))]))
The trouble with copy-and-paste

```
;; [List-of Number]  -->  [List-of Number]
(define (big l)
  (cond [(empty? l) '()]  
        [(cons? l) 
           (if (> (first l) 5) 
               (cons (first l) (big (rest l))) 
               (big (rest l)))]))
```

Copy-and-paste

```
;; [List-of Number]  -->  [List-of Number]
(define (huge l)
  (cond [(empty? l) '()]  
        [(cons? l) 
           (if (> (first l) 10) 
               (cons (first l) (huge (rest l))) 
               (huge (rest l)))]))
```
The trouble with copy-and-paste

After copy-and-paste, improvement is twice as hard!
The trouble with copy-and-paste

```scheme
;; [List-of Number] -> [List-of Number]
(define (big l)
  (cond [(empty? l) '()]
        [(cons? l)
         (if (> (first l) 5)
             (cons (first l) (big (rest l)))
             (big (rest l))))])
```

```scheme
;; [List-of Number] -> [List-of Number]
(define (huge l)
  (cond [(empty? l) '()]
        [(cons? l)
         (if (> (first l) 10)
             (cons (first l) (huge (rest l)))
             (huge (rest l)))]))
```
The trouble with copy-and-paste

```scheme
;; [List-of Number] -> [List-of Number]
(define (big l)
  (cond [(empty? l) '()] 
        [(cons? l)
           (if (> (first l) 5)
               (cons (first l) (big (rest l)))
               (big (rest l)))]))
```

Copy-and-paste

```scheme
;; [List-of Number] -> [List-of Number]
(define (huge l)
  (cond [(empty? l) '()] 
        [(cons? l)
           (if (> (first l) 10)
               (cons (first l) (huge (rest l)))
               (huge (rest l)))]))
```
The trouble with copy-and-paste

After copy-and-paste, bugs multiply!
The moral: Avoid copying and pasting code between functions!
How to avoid copy-and-paste

Start with the original function...

;;; [List-of Number] -> [List-of Number]
(define (big l)
  (cond [(empty? l) '()] [(cons? l)
      [(if (> (first l) 5)
          (cons (first l) (big (rest l)))
          (big (rest l))))]))
How to avoid copy-and-paste

...and add arguments for parts that should change.

;;; [List-of Number] Number -> [List-of Number]
(define (bigger l n)
  (cond [[(empty? l) '()]
         [(cons? l)
          (if (> (first l) n)
            (cons (first l) (bigger (rest l) n))
            (bigger (rest l) n))])))
How to avoid copy-and-paste

...and add arguments for parts that should change.

;;; [List-of Number] Number -> [List-of Number]
(define (bigger l n)
  (cond [(empty? l) '()]
        [(cons? l)
         (if (> (first l) n)
             (cons (first l) (bigger (rest l) n))
             (bigger (rest l) n))])

(define (big l) (bigger l 5))
How to avoid copy-and-paste

...and add arguments for parts that should change.

```
;; [List-of Number] Number -> [List-of Number]
(define (bigger l n)
  (cond [[(empty? l) '()]
          [(cons? l)
            (if (> (first l) n)
              (cons (first l) (bigger (rest l) n))
              (bigger (rest l) n))])

(define (big l) (bigger l 5))

(define (huge l) (bigger l 10))
```
bigger

abstraction of

big
  n=5

abstraction of

huge
  n=10
Small fish

Now we want the small fish:
Small fish

Now we want the small fish:

;; [List-of Number] Number -> [List-of Number]
(define smaller l n)
  (cond [(empty? l) '()]
        [(cons? l)
         (if (< (first l) n)
             (cons (first l) (smaller (rest l) n))
             (smaller (rest l) n))])

(define small l) (smaller l 5)
Small fish

Now we want the small fish:

```scheme
;; [List-of Number] Number -> [List-of Number]
(define (smaller l n)
  (cond [(empty? l) '()]
        [(cons? l)
         (if (< (first l) n)
             (cons (first l) (smaller (rest l) n))
             (smaller (rest l) n))]]

(define (small l) (smaller l 5))
```

*No! Don't copy-and-paste!*
Sized fish

;;; [List-of Number] Number ... -> [List-of Number]
(define (sized l n COMP)
  (cond [(empty? l) '()]
        [(cons? l)
          (if (COMP (first l) n)
              (cons (first l) (sized (rest l) n COMP))
              (sized (rest l) n COMP))])))

(define (bigger l n) (sized l n >))

(define (smaller l n) (sized l n <))

Does this work? What's the signature for sized?
Functions as values

The definition

```
(define (bigger l n) (sized l n >))
```

works because *functions are values*. 
Functions as values

The definition

\[
  \text{(define (bigger \ l \ n) (sized \ l \ n >))}
\]

works because \textit{functions are values}.

10 is a \textit{Number}

#false is a \textit{Boolean}
Functions as values

The definition

(define (bigger l n) (sized l n >))

works because functions are values.

10 is a Number

#false is a Boolean

< is a (Number Number -> Boolean)
Functions as values

The definition

\[
\text{(define (bigger l n) (sized l n >))}
\]

works because \textit{functions are values}.

10 is a \textit{Number}

#false is a \textit{Boolean}

< is a (\textit{Number Number} \to \textit{Boolean})

So the signature for \textbf{sized} is:

\[
;; \ [\text{List-of Number}] \ \text{Number} \ (\text{Number Number} \to \text{Boolean})
\]

\[
;; \to \ [\text{List-of Number}]
\]
sized

smaller

big

bigger

huge
Sized fish

;; [List-of Number] Number (Number Number -> Boolean)
;; -> [List-of Number]
(define (sized l n COMP)
  (cond [(empty? l) '()] [(cons? l)
      (if (COMP (first l) n)
        (cons (first l) (sized (rest l) n COMP))
        (sized (rest l) n COMP))]))

(define (tiny l) (sized l 2 <))

(define (medium l) (sized l 5 =))
We could redefine **big** and **huge** in terms of sized without using the **smaller** or **bigger** functions.
Sized fish

;;; [List-of Number] Number (Number Number -> Boolean)
;;; -> [List-of Number]
(define (sized l n COMP)
  (cond [(empty? l) '()]
        [(cons? l)
            (if (COMP (first l) n)
                (cons (first l) (sized (rest l) n COMP))
                (sized (rest l) n COMP))])))

How about all fish between 3 and 7 lbs?
Mediumish fish

;; Number Number → Boolean
(define (btw-3-and-7 a ignored-zero)
  (and (>= a 3)
       (<= a 7)))

(define (mediumish l) (sized l 0 btw-3-and-7))
Mediumish fish

;; Number Number -> Boolean
(define (btw-3-and-7 a ignored-zero)
  (and (>= a 3)
       (<= a 7)))

(define (mediumish l) (sized l 0 btw-3-and-7))

Programmer-defined functions are values too.

Note that the contract of btw-3-and-7 matches the kind expected by sized.
Mediumish fish

;; Number Number -> Boolean
(define (btw-3-and-7 a ignored-zero)
  (and (>= a 3)
       (<= a 7)))

(define (mediumish l) (sized l 0 btw-3-and-7))

Programmer-defined functions are values too.

Note that the contract of btw-3-and-7 matches the kind expected by sized.

But the ignored 0 suggests a simplification of sized...
A generic number filter

;;; (Number -> Boolean) [List-of Number] -> [List-of Number]
(define (filter-nums PRED l)
  (cond [(empty? l) '()]
        [(cons? l)
           (if (PRED (first l))
               (cons (first l) (filter-nums PRED (rest l)))
               (filter-nums PRED (rest l)))]))
A generic number filter

;; (Number -> Boolean) [List-of Number] -> [List-of Number]
(define (filter-nums PRED l)
  (cond [(empty? l) '()]
       [(cons? l)
        [(cons? l)
         (if (PRED (first l))
          (cons (first l) (filter-nums PRED (rest l)))
          (filter-nums PRED (rest l)))])))

(define (btw-3&7 n) (and (>= n 3) (<= n 7))

(define (mediumish l) (filter-nums btw-3&7 l))
Big and huge fish, again

(define (more-than-5 n)
  (> n 5))

(define (big l)
  (filter-nums more-than-5 l))

(define (more-than-10 n)
  (> n 10))

(define (huge l)
  (filter-nums more-than-10 l))
filter-nums

btw-3&7  mediumish  big  huge
Functional abstraction is the process of creating abstract functions such as filter-nums.

- It reduces code size.
- It avoids copy-and-paste.
- Bugs can be fixed in one place instead of many.
- Improving one functional abstraction improves many applications.
Acknowledgments

This lecture incorporates material from:

Matthias Felleisen
Robert Bruce Findler
Matthew Flatt
Shriram Krishnamurthi
Marc Smith