Exam 1

Due at 11:59 p.m.

Assignment 2

Extended to Thursday, 5:00 p.m.
Basic abstraction
We’ve seen that we can represent a point in space using a `posn` structure:

\[
\text{(make-posn 10 3)}
\]

has x-coordinate 10 and y-coordinate 3 and

\[
\text{(make-posn 5 20)}
\]

has x-coordinate 5 and y-coordinate 20.
If we have two points, we might want to compute their distance $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$:

```
(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:

```
(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:

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

becomes

\[
\sqrt{+ (\text{sqr} (- (\text{posn-x}\ P1)
\text{ (posn-x}\ P2))
\text{ (sqr} (- (\text{posn-y}\ P1)
\text{ (posn-y}\ P2))))})
\]
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.

```
(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.
Functional abstraction
Many functions have the same basic shape.
The difference between two functions might just be a single value.
Parameterizing
;; ListOfStrings -> Boolean
;; Does l contain "dog"?
(define (contains-dog? l)
  (and (not (empty? l))
       (or (string=? (first l) "dog")
           (contains-dog? (rest l))))))

;; ListOfStrings -> Boolean
;; Does l contain "cat"?
(define (contains-cat? l)
  (and (not (empty? l))
       (or (string=? (first l) "cat")
           (contains-cat? (rest l))))))
;; ListOfStrings -> Boolean
;; Does l contain "dog"?
(define (contains-dog? l)
  (and (not (empty? l))
       (or (string=? (first l) "dog"
                   (contains-dog? (rest l))))))

;; ListOfStrings -> Boolean
;; Does l contain "cat"?
(define (contains-cat? l)
  (and (not (empty? l))
       (or (string=? (first l) "cat"
                    (contains-cat? (rest l))))))
We can **parametrize** the difference between the functions, namely the string we’re checking for:

```lisp
;; ListOfStrings, String -> Boolean
;; Does l contain the string s?
(define (contains? l s)
  (and (not (empty? l))
       (or (string=? (first l) s)
           (contains? (rest l) s))))
```
This is a kind of **functional abstraction**: 

`contains-dog?` and `contains-cat?` can now be defined in terms of `contains??:

```scheme
;; ListOfStrings -> Boolean
;; Does l contain "dog"?
(define (contains-dog? l)
  (contains? l "dog"))

;; ListOfStrings -> Boolean
;; Does l contain "cat"?
(define (contains-cat? l)
  (contains? l "cat"))
```
Dangerous duplication
Big fish

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

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

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

```scheme
;; ListOfNumbers -> ListOfNumbers
(check-expect (big '()) '())
(check-expect (big '(7 4 9)) '(7 9))

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

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

;;; ListOfNumbers -> ListOfNumbers
(check-expect (big '()) '())
(check-expect (big '(7 4 9)) '(7 9))

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

Suppose we also need to find huge fish…
Huge fish

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

```scheme
;; ListOfNumbers -> ListOfNumbers
(check-expect (huge '()) '())
(check-expect (huge '(17 4 9)) '(17))
```
Huge fish

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

```scheme
;; ListOfNumbers -> ListOfNumbers
(check-expect (huge '()) '())
(check-expect (huge '(17 4 9)) '(17))

(define (huge l)
  (cond [(empty? l) '()]
        [(cons? l)
          (if (> (first l) 10)
            (cons (first l) (huge (rest l)))
            (huge (rest l)))]))
```
Huge fish

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

;;; ListOfNumbers -> ListOfNumbers
(check-expect (huge '()) '())
(check-expect (huge '(17 4 9)) '(17))

(define (huge l)
  (cond [(empty? l) '()]
        [(cons? l)
         (if (> (first l) 10)
             (cons (first l) (huge (rest l)))
             (huge (rest l)))]))

How do you suppose I made this slide?
Huge fish

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

```scheme
;; ListOfNumbers -> ListOfNumbers
(check-expect (huge '()) '())
(check-expect (huge '(17 4 9)) '(17))

(define (huge l)
  (cond [(empty? l) '()]
       [(cons? l)
        (if (> (first l) 10)
            (cons (first l) (huge (rest l)))
            (huge (rest l))))])
```

How do you suppose I made this slide?

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

```scheme
;; ListOfNumbers -> ListOfNumbers
(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
;; ListOfNumbers -> ListOfNumbers
(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

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

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

Copy-and-paste

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

;;; ListOfNumbers -> ListOfNumbers
(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:** Form an abstraction instead of copying and modifying any code.

Doing this gives you one place to make changes to the code, e.g., to fix a bug. If you make copies of code in many similar functions, you'll have many places to update.
How to avoid copy-and-paste

Start with the original function...

;;; ListOfNumbers -> ListOfNumbers
(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.

;; ListOfNumbers, Number -> ListOfNumbers
(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.

;;; ListOfNumbers, Number -> ListOfNumbers
(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.

;;; ListOfNumbers, Number -> ListOfNumbers
(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

$\ n=10$
Parameterizing functions
;; ListOfNumbers, Number -> ListOfNumbers
(define (bigger l n)
  (cond [(empty? l) '()]
        [(cons? l)
         (if (> (first l) n)
           (cons (first l) (bigger (rest l) n))
           (bigger (rest l) n))])))
Now we want to select fish smaller than some size:
Now we want to select fish smaller than some size:

```scheme
;; ListOfNumbers, Number -> ListOfNumbers
(define (smaller l n)
  (cond [(empty? l) '()
            [(cons? l)
                (if (< (first l) n)
                    (cons (first l) (smaller (rest l) n))
                    (smaller (rest l) n)))]))
```
Now we want to select fish smaller than some size:

```
;; ListOfNumbers, Number -> ListOfNumbers
(define (smaller l n)
  (cond [(empty? l) '()]
        [(cons? l)
         (if (< (first l) n)
             (cons (first l) (smaller (rest l) n))
             (smaller (rest l) n)))])
```
;; ListOfNumbers, Number -> ListOfNumbers
(define (bigger l n)
  (cond [(empty? l) '()]
        [(cons? l)
         (if (> (first l) n)
             (cons (first l) (bigger (rest l) n))
             (bigger (rest l) n))]))

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

;; ListOfNumbers, Number -> ListOfNumbers
(define (smaller l n)
  (cond [(empty? l) '()] [(cons? l)
    (if (< (first l) n)
      (cons (first l) (smaller (rest l) n))
      (smaller (rest l) n))]))
Sized fish

;; ListOfNumbers, Number, ... -> ListOfNumbers
(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

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

works because \textit{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

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

works because \textit{functions are values}.

\(10\) is a \textit{Number}

\#false is a \textit{Boolean}

\(<\) is a \[\text{Number, Number } \rightarrow \text{Boolean}\]

So the signature for \textit{sized} is:

\[
;; \text{ListOfNumbers, Number, [Number, Number } \rightarrow \text{Boolean} \rightarrow \text{ListOfNumbers}
\]
To run this code, you’ll need to switch to the *Intermediate Student* language.
Sized fish

;; ListOfNumbers, Number, [Number, Number -> Boolean] -> ListOfNumbers
(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

;; ListOfNumbers, Number, [Number, Number -> Boolean] -> ListOfNumbers
(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 \texttt{btw-3-and-7} matches the kind expected by \texttt{sized}.

But the ignored 0 suggests a simplification of \texttt{sized}...
A generic number filter

;; [Number -> Boolean], ListOfNumbers -> ListOfNumbers
(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], ListOfNumbers -> ListOfNumbers
(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))))])

(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))
**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.
Abstracting data definitions
Many data definitions have the same basic shape.
Data definitions

;;; A ListOfNumbers is one of:
;;; - '()
;;; - (cons Number ListOfNumbers)

;;; A ListOfStrings is one of:
;;; - '()
;;; - (cons String ListOfStrings)

Parametric data definition

;;; A [List-of ITEM] is one of:
;;; - '()
;;; - (cons ITEM [List-of ITEM])

;;; ListOfNumbers = [List-of Number]
;;; ListOfStrings = [List-of String]
A [Maybe X] is one of:
- #false
- X

This parametric data definition lets us talk about:

[Maybe String]
[Maybe [List-of String]]
[List-of [Maybe String]]
Abstractions!
Abstractions are simplified models of something complicated, with a simple interface.

Good abstractions hide the details of the machinery implementing them.

This allows programmers to debug them without needing to dig into the details of the underlying machines.
For example, a file is presented as a container for a string of bits with two operations: read and write.

Its more complicated implementation – records scattered across a hard disk – is hidden.
Finding good abstractions is an essential design skill for programmers and software engineers.
Acknowledgments

This lecture incorporates material from:

Peter Denning & Matti Tedre, Computational Thinking
Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi, How to Design Programs, second ed.
Marc Smith, Vassar College