Functions that produce functions
We’ve written functions that *consume* functions.

Now that we have lambda expressions, we can write functions that *produce* functions too!
;; Number -> [Number -> Number]
;; Return a function that adds n to its input
(define make-adder n)
  (lambda (m) (+ m n))

> (make-adder 10)
(lambda (a1) ...)
> ((make-adder 10) 5)
15
> (define add-ten (make-adder 10))
> (add-ten 5)
15
Higher-order functions: filter
A [List-of Symbol] program:

;; [List-of Symbol] -> [List-of Symbol]
;; Remove any occurrence of the symbol 'apple
(define (eat-apples l)
  (cond [(empty? l) '()]
        [(cons? l)
          (local [(define ate-rest (eat-apples (rest l)))]
            (if (symbol=? (first l) 'apple)
                ate-rest
                (cons (first l) ate-rest))))))
A **[List-of Symbol]** program:

```scheme
;; [List-of Symbol] -> [List-of Symbol]
;; Remove any occurrence of the symbol 'apple
(define (eat-apples l)
  (cond [(empty? l) '()]
        [(cons? l)
         (local [(define ate-rest (eat-apples (rest l)))]
           (if (symbol=? (first l) 'apple)
               ate-rest
               (cons (first l) ate-rest)))]))
```

How about **eat-bananas**?
A \textit{List-of Symbol} program:

\begin{verbatim}
;; [List-of Symbol] -> [List-of Symbol]
;; Remove any occurrence of the symbol 'apple
(define (eat-apples l)
  (cond ((empty? l) '())
        ((cons? l)
         (local [((define ate-rest (eat-apples (rest l)))]
          (if (symbol=? (first l) 'apple)
             ate-rest
             (cons (first l) ate-rest))))))
\end{verbatim}

How about \texttt{eat-bananas}?

How about \texttt{eat-non-apples}?
A **List-of Symbol** program:

```scheme
;; [List-of Symbol] -> [List-of Symbol]
;; Remove any occurrence of the symbol 'apple
(define (eat-apples l)
  (cond [(empty? l) '()] [(cons? l)
      (local [(define ate-rest (eat-apples (rest l)))]
        (if (symbol=? (first l) 'apple)
          ate-rest
          (cons (first l) ate-rest))))))
```

How about **eat-bananas**?

How about **eat-non-apples**?

We know where this leads…
;;  [Symbol -> Boolean]  [List-of Symbol]  ->  [List-of Symbol]
;;  Remove any symbol *not* matched by PRED
(define (filter-syms PRED l)
  (cond [(empty? l) '()]
        [(cons? l)
         (local [(define r (filter-syms PRED (rest l)))]
                (if (PRED (first l))
                    (cons (first l) r)
                    r)))]))
;; [Symbol $\rightarrow$ Boolean] [List-of Symbol] $\rightarrow$ [List-of Symbol]

;; Remove any symbol *not* matched by PRED

(define (filter-syms PRED l)
  (cond [(empty? l) '()]
        [(cons? l)
         (local [(define r (filter-syms PRED (rest l)))]
                 (if (PRED (first l))
                     (cons (first l) r)
                     r))])))

This should look familiar.
When we were learning about abstraction in Lecture 16, we wrote this function.

;;; [Number -> Boolean] [List-of Number] -> [List-of Number]
;;; Remove any number *not* matched by PRED
(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)))]))

We hadn’t covered local yet, but we can update this to use it.
;; [Number -> Boolean] [List-of Number] -> [List-of Number]  
;; Remove any number *not* matched by PRED
(define (filter-nums PRED l)
  (cond [(empty? l) '()]
        [(cons? l)
          (local [(define r (filter-nums PRED (rest l)))]
            (if (PRED (first l))
                (cons (first l) r)
                r))])))
(define (filter-syms PRED l)
  (cond [[(empty? l) '()]
       [(cons? l)
          (local [[(define r (filter-syms PRED (rest l)))]
            (if (PRED (first l))
              (cons (first l) r)
              r))]]]))

(define (filter-nums PRED l)
  (cond [[(empty? l) '()]
       [(cons? l)
          (local [[(define r (filter-nums PRED (rest l)))]
            (if (PRED (first l))
              (cons (first l) r)
              r))]]))
;; [Symbol -> Boolean] [List-of Symbol] -> [List-of Symbol]
;; Remove any symbol *not* matched by PRED
(define (filter-syms PRED l)
  (cond [[(empty? l) '()] 
        [(cons? l) 
          (local [(define r (filter-syms PRED (rest l)))] 
            (if (PRED (first l)) 
              (cons (first l) r) r)))]))

;; [Number -> Boolean] [List-of Number] -> [List-of Number]
;; Remove any number *not* matched by PRED
(define (filter-nums PRED l)
  (cond [[(empty? l) '()] 
        [(cons? l) 
          (local [(define r (filter-nums PRED (rest l)))] 
            (if (PRED (first l)) 
              (cons (first l) r) r)))]))

How do we avoid copy-and-paste?
;; [Number -> Boolean] [List-of Number] --> [List-of Number]
;; Remove any number *not* matched by PRED
(define (filter PRED l)
  (cond [(empty? l) '()] [(cons? l)
      (local [(define r (filter PRED (rest l)))]
        (if (PRED (first l))
            (cons (first l) r)
            r)))]))

This function will work for both number and symbol lists.

But what’s its signature?
How about this?

`; [NumberOrSymbol -> Boolean]
`; [List-of NumberOrSymbol]
`; -> [List-of NumberOrSymbol]

with the data definition

`; A NumberOrSymbol is either
`; – Number
`; – Symbol
How about this?

;; [NumberOrSymbol -> Boolean]
;; [List-of NumberOrSymbol]
;; -> [List-of NumberOrSymbol]

This signature is too weak to define **eat-apples**:

;; [List-of Symbol] -> [List-of Symbol]
(define (eat-apples l) (filter not-apple? l))

;; Symbol -> Boolean
(define (not-apple? s) (not (symbol=? s 'apple)))

eat-apples must return a **[List-of Symbol]**, but by its signature, filter might return a **[List-of Number]**.

not-apple? only works on **Symbols**, but by its signature, filter might give it a **Number**.
The reason filter works is that if we give it a \[\text{List-of Symbol}\], then it returns a \[\text{List-of Symbol}\].

Also, if we give it a \[\text{List-of Symbol}\], then it calls \text{PRED} with \text{Symbols} only.
A better signature:

```plaintext
;; filter:
;;   ([Number -> Boolean] [List-of Number]
;;    -> [List-of Number])
;; or
;;   ([Symbol -> Boolean] [List-of Symbol]
;;    -> [List-of Symbol])
```

But what about a list of images, posns, rabbits?
The real signature needs to use a variable:

```haskell
;; filter : [X -> Boolean] [List-of X] -> [List-of X]
```

where the caller of `filter` gets to pick a type for `X`, and all `X`s in the signature must be replaced with the same type.
Signatures like these match the ways that we’ve used variables in data definitions, e.g.,

```plaintext
;; A [List–of X] is either:
;; – '()
;; – (cons X [List–of X])
```
The **filter** function is so useful that it’s built in to Racket:

```scheme
(define (eat-apples l)
  (local [(define (not-apple? s)
            (not (symbol=? s 'apple)))
           (filter not-apple? l))]
  (filter not-apple? l)))

or, using an anonymous function,

```scheme
(define (eat-apples l)
  (filter (lambda (s)
            (not (symbol=? s 'apple)))
          l))
```
Currying and composing

This is advanced, potentially confusing, material, which is unlikely to be on an exam, but try to follow along.
Suppose we need to filter for different symbols:

\[
\text{filter (lambda (s) (symbol=? s 'apple)) l) }
\]
\[
\text{filter (lambda (s) (symbol=? s 'banana)) l) }
\]
\[
\text{filter (lambda (s) (symbol=? s 'cherry)) l) }
\]

Instead of repeating the long \texttt{lambda} expression, we can abstract:

\[
\text{;;; mk-is-sym : Symbol -> [Symbol -> Boolean]}
\]
\[
\text{(define (mk-is-sym a)}
\]
\[
\text{(lambda (s) (symbol=? a s)))}
\]

\[
\text{(filter (mk-is-sym 'apple) l) }
\]
\[
\text{(filter (mk-is-sym 'banana) l) }
\]
\[
\text{(filter (mk-is-sym 'cherry) l) }
\]

\text{\texttt{mk-is-sym} is a \textit{curried} version of \texttt{symbol=?}}
Currying functions

Named in honor of the logician Haskell Curry, currying turns an $n$-argument function into an $n-1$-argument function.
We can define a function to carry out this transformation for any two-argument function:

;;; \textit{curry} : [X \ Y \rightarrow Z] \rightarrow [X \rightarrow [Y \rightarrow Z]]

(define (\textit{curry} \ f)
  (lambda (v1)
    (lambda (v2)
      (f v1 v2)))))

(define \textit{mk-is-sym} (\textit{curry} symbol=?))

(filter (\textit{mk-is-sym} 'apple) l)
(filter (\textit{mk-is-sym} 'banana) l)
(filter (\textit{mk-is-sym} 'cherry) l)
We can define a function to carry out this transformation for any two-argument function:

;; curry : \[X \ Y \rightarrow Z\] \rightarrow \[X \rightarrow [Y \rightarrow Z]\]
(define (curry f)
  (lambda (v1)
    (lambda (v2)
      (f v1 v2))))

(filter ((curry symbol=?)) 'apple) l)
(filter ((curry symbol=?)) 'banana) l)
(filter ((curry symbol=?)) 'cherry) l)
But we want symbols that don’t match the given symbol.

We can compose that negation with the curried function:

```
;; compose : [Y -> Z] [X -> Y] -> [X -> Z]
(define (compose f g)
  (lambda (x) (f (g x))))

(filter (compose not
         ((curry symbol=??) 'apple))
        l)
```
Higher-order functions: map
Functions like *filter* that take a predicate as argument are called *higher-order functions*, because they are an abstraction over an infinite number of more specific functions.

These can save you from writing lots of repetitive code!

Let’s look for some more built-in higher-order functions!
Recall **feed-fish**:

```scheme
;; ListOfNumbers -> ListOfNumbers
(define (feed-fish l)
  (cond [(empty? l) '()]
        [else
         (cons (+ 1 (first l))
               (feed-fish (rest l)))]))
```
Recall **feed-fish**:

```scheme
;; ListOfNumbers -> ListOfNumbers
(define (feed-fish l)
  (cond [(empty? l) '()]
        [else
          (cons (+ 1 (first l))
               (feed-fish (rest l)))]))
```

Is there a built-in function to help?
Recall **feed-fish**:

```
;; ListOfNumbers -> ListOfNumbers
(define (feed-fish l)
  (cond [(empty? l) '()]
        [else
         (cons (+ 1 (first l))
              (feed-fish (rest l)))])
```

Is there a built-in function to help?

Yes! It’s **map**.
(define (map CONV l)
  (cond [(empty? l) '()]
         [else
          (cons (CONV (first l))
                (map CONV (rest l)))]))

(define (map CONV l)
  (cond [(empty? l) '()] [else
       (cons (CONV (first l))
             (map CONV (rest l)))]))

;; [List-of Number] -> [List-of Number]
(define (feed-fish l)
  (local [(define (feed-one n) (+ n 1))]
    (map feed-one l)))

(define (feed-animals l)
  (map feed-animal l))
The signature for \texttt{map}

\begin{equation*}
\begin{align*}
\text{(define (\texttt{map CONV l})}
\hspace{1cm}
(\text{cond [((empty? l) '())]}
\hspace{1cm}
[\text{else}
\hspace{1cm}
(\text{cons (CONV (first l))}
\hspace{1cm}
(\text{map CONV (rest l)))]])
\end{align*}
\end{equation*}

The \texttt{l} argument must be a list of \texttt{X}

The \texttt{CONV} argument must accept each \texttt{X}

If \texttt{CONV} returns a new \texttt{X} each time, then the signature for \texttt{map} is:

\begin{equation*}
\begin{align*}
\text{;; map : [X \rightarrow X] [List-of X] \rightarrow [List-of X]}
\end{align*}
\end{equation*}
posns and distances

Consider this function:

;; [List-of Posn] -> [List-of Number]
;; Compute the distance of each posn from the origin.
(define (distances l)
  (cond [(empty? l) '()] [(cons? l)
    (cons (distance-to-0 (first l)) (distances (rest l))))])
**posns and distances**

Consider this function:

```scheme
;;; [List-of Posn] -> [List-of Number]
;;; Compute the distance of each posn
;;; from the origin.
(define (distances l)
  (cond [[(empty? l) '()]
        [(cons? l)
         (cons (distance-to-0 (first l))
               (distances (rest l)))]))
```

This looks just like `map`, except that `distances-to-0` is `Posn → Number`, not `Posn → Posn`.
The true signature of map

Despite the signature mismatch, this works:

\[
\text{(define (distance } l) \\
\text{ (map distance-to-0 } l))
\]
The true signature of `map`

Despite the signature mismatch, this works:

```scheme
(define (distance l)
  (map distance-to-0 l))
```

That’s because the true signature of `map` is:

```scheme
;; [X -> Y] [List of X] -> [List-of Y]
```

That is, the caller gets to pick both `X` and `Y` independently.
More uses of map

;;; flip-posns : [List-of Posn] → [List-of Posn]
(define (flip-posns lop)
  ;; Replaces 4 lines:
  (map flip-posn lop))

;;; flip-posn : Posn → Posn
...

...
More uses of map

;; rob-train : [List-of Car] -> [List-of Car]
(define (rob-train l)
    ;; Replaces 4 lines:
    (map rob-car l))

;; rob-car : Car -> Car
...

Higher-order functions: foldr
How about `sum`?

```haskell
;; sum : [List-of Number] -> Number
```

Doesn’t return a list, so neither `filter` nor `map` help.
Abstracting over functions like \texttt{sum} and \texttt{product} leads to \texttt{combine-nums}:

\begin{verbatim}
;; combine-nums :
;;  [List-of Number] Number [Number Number \rightarrow Number]
;;  \rightarrow Number
(define (combine-nums l base-n COMB)
  (cond [(empty? l) base-n]
        [(cons? l)
         (COMB (first l)
                 (combine-nums (rest l) base-n COMB))])
\end{verbatim}

We need to specify not just a function but a base value, namely the identity for addition and multiplication respectively.
Folding a list

There’s a built-in function like this, though the order of the arguments is different:

```scheme
(define (foldr COMB base l)
  (cond [(empty? l) base]
        [(cons? l)
         (COMB (first l)
               (foldr COMB base (rest l))))])
```

The **sum** and **product** functions become trivial:

```scheme
(define (sum l) (foldr + 0 l))
(define (product l) (foldr * 1 l))
```
(define (foldr COMB base l)
  (cond [(empty? l) base]
       [(cons? l)
        (COMB (first l)
            (foldr COMB base (rest l))))]))

If we define a helper that takes two arguments, we can use foldr to compute the sum of all the distances of posns from the origin:

;; total-distance : [List-of Posn] -> Number
(define (total-distance l)
  (local [(define (add-distance p n)
               (+ (distance-to-0 p) n))
          (foldr add-distance 0 l)))]
(define (foldr COMB base l)
  (cond [(empty? l) base]
        [(cons? l)
         (COMB (first l)
               (foldr COMB base (rest l))))]))

In fact,

(define (map f l)
  (local [(define (comb i r)
               (cons (f i) r))]
         (foldr comb '() l)))

(define (foldr COMB base l)
  (cond [(empty? l) base]
      [(cons? l)
        (COMB (first l)
          (foldr COMB base (rest l))))])

And filter too!

(define (filter f l)
  (local [(define (check i r)
            (if (f i)
                (cons i r)
                r))]
    (foldr check '() l)))
How can `foldr` be so powerful?

Template for a `[List-of X]`:

```scheme
(define (fun-for-loX l)
  (cond [(empty? l) ...]
       [(cons? l)
        (... (first l) ...
        (fun-for-loX (rest l)) ...)])
)
```

Fold:

```scheme
(define (foldr COMB base l)
  (cond [(empty? l) base]
       [(cons? l)
        (COMB (first l)
        (foldr COMB base (rest l)))]))
)```
Other built-in list functions

More specializations of \texttt{foldr}:

\begin{verbatim}
;; ormap : [X \rightarrow Boolean] [List-of X] \rightarrow Boolean
;; andmap : [X \rightarrow Boolean] [List-of X] \rightarrow Boolean
\end{verbatim}

Examples:

\begin{verbatim}
;; got-milk? : [List-of Symbol] \rightarrow Boolean
(define (got-milk? l)
  (local [(define (is-milk? s)
            (symbol=? s 'milk))]
            (ormap is-milk? l)))

;; all-passed? : [List-of Grade] \rightarrow Boolean
(define (all-passed? l)
  (andmap passing-grade? l))
\end{verbatim}
Higher-order functions: build-list
;; Natural [Natural -> X] -> [List-of X]
;; Produce (list (f 0) ... (f (- n 1))
(define (build-list n f) ...)

> (identity 3)
3
> (build-list 3 identity)
(list 0 1 2)
> (build-list 4 sqr)
(list 0 1 4 9)
Practice using built-in higher-order functions
Complete the design of the following function using a built-in abstract (i.e., higher-order) list function.

(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 30 20 "solid" "yellow"))
(define I3 (rectangle 40 50 "solid" "green"))
(define I4 (rectangle 60 50 "solid" "blue"))
(define I5 (rectangle 90 90 "solid" "orange"))

(define (wide? img)
  (> (image-width img) (image-height img)))

;; [List-of Image] -> [List-of Image]
;; Produce list of only those images that are wide?
(define (wide-only loi) '())

(check-expect (wide-only (list I1 I2 I3 I4 I5))
  (list I2 I4))
Complete the design of the following function using a built-in abstract (i.e., higher-order) list function.

(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 30 20 "solid" "yellow"))
(define I3 (rectangle 40 50 "solid" "green"))
(define I4 (rectangle 60 50 "solid" "blue"))
(define I5 (rectangle 90 90 "solid" "orange"))

(define (wide? img)
  (> (image-width img) (image-height img)))

;; [List-of Image] -> [List-of Image]
;; Produce list of only those images that are wide?
(define (wide-only loi)
  (filter wide? loi))

(check-expect (wide-only (list I1 I2 I3 I4 I5))
  (list I2 I4))
We can move helper functions into a local environment:

```scheme
;; [List-of Image] -> [List-of Image]
;; Produce list of only those images that are wide?
(define (wide-only loi)
  (local [(define (wide? img)
            (> (image-width img) (image-height img)))]
    (filter wide? loi)))

This makes sense if the helper will only be used for this function.
```
If the helper function depends on one of the main function’s parameters, then we need to define it locally (or else create a curried version, but that’s trickier):

```scheme
;; Number [List-of Image] -> [List-of Image]
;; Produce list of only those images with width > w
(define (wider-than-only w loi)
  (local [(define (wider-than? img)
              (> (image-width img) w))]
    (filter wider-than? loi)))
```
Complete the design of the following function using a built-in abstract (i.e., higher-order) list function.

(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 30 20 "solid" "yellow"))
(define I3 (rectangle 40 50 "solid" "green"))
(define I4 (rectangle 60 50 "solid" "blue"))
(define I5 (rectangle 90 90 "solid" "orange"))

(define (tall? img)
  (< (image-width img) (image-height img)))

;; [List-of Image] -> Boolean
;; Are all of the images in loi tall?
(define (all-tall? loi) #false)

(check-expect (all-tall? (list I1 I2 I3 I4 I5)) #false)
(check-expect (all-tall? (list I1 I3)) #true)
Complete the design of the following function using a built-in abstract (i.e., higher-order) list function.

(define I1 (rectangle 10 20 "solid" "red"))
(define I2 (rectangle 30 20 "solid" "yellow"))
(define I3 (rectangle 40 50 "solid" "green"))
(define I4 (rectangle 60 50 "solid" "blue"))
(define I5 (rectangle 90 90 "solid" "orange"))

(define (tall? img)
  (< (image-width img) (image-height img)))

;; [List-of Image] -> Boolean
;; Are all of the images in loi tall?
(define (all-tall? loi)
  (andmap tall? loi))

(check-expect (all-tall? (list I1 I2 I3 I4 I5)) #false)
(check-expect (all-tall? (list I1 I3)) #true)
;; [List-of Number] -> Number
;; Sum the elements of a list
(define (sum lon) 0)

(check-expect (sum (list 1 2 3 4)) 10)
;; [List-of Number] -> Number
;; Sum the elements of a list
(define (sum lon)
  (foldr ... ... lon))

(check-expect (sum (list 1 2 3 4)) 10)
;; [List-of Number] -> Number
;; Sum the elements of a list
(define (sum lon)
  (foldr ... ... lon))

(check-expect (sum (list 1 2 3 4)) 10)
;; [List-of Number] -> Number
;; Sum the elements of a list
(define (sum lon)
  (foldr + 0 lon))

(check-expect (sum (list 1 2 3 4)) 10)
;; Natural -> Natural
;; Produce the sum of the first n natural numbers
(define (sum-to n) 0)

(check-expect (sum-to 3) (+ 0 1 2))
;; Natural -> Natural
;; Produce the sum of the first n natural numbers
(define (sum-to n)
  (foldr ... ... (build-list n ...)))

(check-expect (sum-to 3) (+ 0 1 2))
;; Natural -> Natural
;; Produce the sum of the first n natural numbers
(define (sum-to n)
  (foldr ... ... (build-list n identity)))

(check-expect (sum-to 3) (+ 0 1 2))
;;; Natural -> Natural
;;; Produce the sum of the first n
;;; natural numbers
(define (sum-to n)
  (foldr + 0 (build-list n identity)))

(check-expect (sum-to 3) (+ 0 1 2))
Acknowledgments

This lecture incorporates material from:

- Matthias Felleisen
- Robert Bruce Findler
- Matthew Flatt
- Gregor Kiczales
- Shriram Krishnamurthi
- Marc Smith