Abstraction
Big Fish

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

; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (cond
     [(> (first l) 5)
      (cons (first l) (big (rest l)))]
     [else (big (rest l))]))])

(check-expect (big empty) empty)
(check-expect (big '(7 4 9)) '(7 9))
Big Fish

Better with `local`:

```scheme
; big : list-of(nums) -> list-of(nums)
(define (big l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define big-rest (big (rest l)))]
        (cond
          [(> (first l) 5)
           (cons (first l) big-rest)]
          [else big-rest]))]))
```

Suppose we also need to find huge fish...
Huge Fish

Huge fish (> 10 lbs):

; huge : list-of-nums -> list-of-nums
(define (huge l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define h-rest (huge (rest l)))]
        (cond
          [(> (first l) 10)
            (cons (first l) h-rest)]
          [else h-rest]))]))

How do you suppose I made this slide?

Cut and Paste!
The Trouble With Cut and Paste

; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (cond
        [(> (first l) 5)
          (cons (first l) (big (rest l)))]
        [else (big (rest l)))]))]])

; huge : list-of-nums -> list-of-nums
(define (huge l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (cond
        [(> (first l) 10)
          (cons (first l) (huge (rest l)))]
        [else (huge (rest l))])]))
The Trouble With Cut and Paste

; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (cond
        [(> (first l) 5)
          (cons (first l) (big (rest l)))]
        [else (big (rest l))]))])

; huge : list-of-nums -> list-of-nums
(define (huge l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (cond
        [(> (first l) 10)
          (cons (first l) (huge (rest l)))]
        [else (huge (rest l))]))])
The Trouble With Cut and Paste

; big : list-of nums -> list-of nums
(define (big l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (cond
        [(> (first l) 5)
          (cons (first l) (big (rest l)))]
        [else (big (rest l))])])

After cut-and-paste, improvement is twice as hard
The Trouble With Cut and Paste

; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
   [(empty? l) empty]
   [(cons? l)
     (local [(define big-rest (big (rest l)))]
       (cond
         [>(first l) 5]
         (cons (first l) big-rest)])
     [else big-rest]))))
The Trouble With Cut and Paste

; big : list-of(nums) => list-of(nums)
(define (big l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define rest (big (rest l)))]
        (cond
          [(> (first l) 5)
            (cons (first l) big-rest)]
          [else big-rest]))]))

cut and paste
cut and paste

cut and paste

cut and paste

; huge : list-of(nums) => list-of(nums)
(define (huge l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define h-rest (huge (rest l)))]
        (cond
          [(> (first l) 10)
            (cons (first l) h-rest)]
          [else h-rest]]))))
The Trouble With Cut and Paste

; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (local [(define rest (big (rest l)))]
      (cond
       [(> (first l) 5)
        (cons (first l) big-rest)]
       [else big-rest]))]))

cut and paste

; huge : list-of-nums -> list-of-nums
(define (huge l)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (local [(define rest (huge (rest l)))]
      (cond
       [(> (first l) 10)
        (cons (first l) h-rest)]
       [else h-rest]))]))

After cut-and-paste, bugs multiply
The Trouble With Cut and Paste

; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (local [(define rest (big (rest l)))]
      (cond
       [(> (first l) 5)
        (cons (first l) big-rest)]
       [else big-rest]))]))

; huge : list-of-nums -> list-of-nums
(define (huge l)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (local [(define rest (huge (rest l)))]
      (cond
       [(> (first l) 10)
        (cons (first l) h-rest)]
       [else h-rest]))]))

After cut-and-paste, bugs multiply
How to Avoid Cut-and-Paste

Start with the original function...

; big : list-of-nums -> list-of-nums
(define (big l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define big-rest (big (rest l)))]
        (cond
          [(> (first l) 5)
            (cons (first l) big-rest)]
          [else big-rest]))]))
How to Avoid Cut-and-Paste

... and add arguments for parts that should change

; bigger : list-of-nums num -> list-of-nums
(define (bigger l n)
  (cond
   [(empty? l) empty]
   [(cons? l)
     (local [(define r (bigger (rest l) n))]
       (cond
        [ (> (first l) n)
          (cons (first l) r)]
        [else r])))))

(define (big l) (bigger l 5))
(define (huge l) (bigger l 10))
Small Fish

Now we want the small fish:

```scheme
; smaller : list-of-nums num -> list-of-nums
(define (smaller l n)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (local [(define r (smaller (rest l) n))]
      (cond
       [(< (first l) n)
        (cons (first l) r)]
       [else r]))))
  (define (small l) (smaller l 5))
```
Sized Fish

; sized : list-of-nums num ... -> list-of-nums
(define (sized l n COMP)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define r
                (sized (rest l) n COMP))]
        (cond
          [(COMP (first l) n)
           (cons (first l) r)]
          [else r])))])

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

Does this work? What is the contract for sized?
Functions as Values

The definition

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

works because \textit{functions are values}

- \textbf{10} is a \textbf{num}
- \textbf{false} is a \textbf{bool}
- \textbf{<} is a \textbf{(num num -> bool)}

So the contract for \textit{sized} is

\[; \text{list-of-nums num (num num -> bool)}\]
\[; \rightarrow \text{list-of-nums}\]
Sized Fish

;; sized : list-of-nums num (num num -> bool)
;; -> list-of-nums
(define (sized l n COMP)
  (cond
   [(empty? l) empty]
   [(cons? l)
     (local [(define r
                      (sized (rest l) n COMP))]
        (cond
         [(COMP (first l) n)
          (cons (first l) r)]
         [else r]])))]
  (define (tiny l) (sized l 2 <))
  (define (medium l) (sized l 5 =))
Sized Fish

; sized : list-of-nums num (num num -> bool)
; -> list-of-nums
(define (sized l n COMP)
    (cond
        [(empty? l) empty]
        [(cons? l)
            (local [(define r
                        (sized (rest l) n COMP))]
                (cond
                    [(COMP (first l) n)
                        (cons (first l) r)]
                    [else r]]))])

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

; btw-3-and-7 : num num -> bool
(define (btw-3-and-7 a ignored-zero)
  (and (>= a 3)
       (<= a 7)))

(define (mediumish 1) (sized 1 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

; filter-nums : (num -> bool) list-of-num
; -> list-of-num
(define (filter-nums PRED l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define r
                (filter-nums PRED (rest l)))]
        (cond
          [(PRED (first l))
            (cons (first l) r)]
          [else r]))))))

(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))

The more-than-5 and more-than-10 functions are really only useful to big and huge

We could make them local to clarify...
Big and Huge Fish, Improved

```
(define (big l)
  (local [([define (more-than-5 n)
            (> n 5)])
          (filter-nums more-than-5 l))]

(define (huge l)
  (local [([define (more-than-10 n)
            (> n 10)])
          (filter-nums more-than-10 l)])

Cut and paste alert!

You don’t think I typed that twice, do you?
```
Big and Huge Fish, Generalized

\[
\text{(define (bigger-than l m)} \\
\quad \text{(local [(define (more-than-m n)} \\
\quad \quad (> n m))]} \\
\quad \text{(filter nums more-than-m l))})
\]

\[
\text{(define (big l) (bigger-than l 5))} \\
\text{(define (huge l) (bigger-than l 10))}
\]
Abstraction

- Avoiding cut and paste is **abstraction**
- No real programming task succeeds without it
Type Abstraction
Symbols

A list-of-sym program:

; eat-apples : list-of-sym -> list-of-sym
(define (eat-apples l)
  (cond
    [[(empty? l) empty]
     [(cons? l)
       (local [(define ate-rest (eat-apples (rest l)))]
         (cond
           [[(symbol=? (first l) 'apple) ate-rest]
            [else (cons (first l) ate-rest)]]))]
    )))

• How about eat-bananas?

• How about eat-non-apples?

We know where this leads...
Filtering Symbols

; filter-syms : (sym -> bool) list-of-sym
; -> list-of-sym
(define (filter-syms PRED l)
  (cond
    [(empty? l) empty]
   [(cons? l)
      (local [(define r
                   (filter-syms PRED (rest l)))]
        (cond
          [(PRED (first l))
            (cons (first l) r)]
          [else r]))]))

This looks really familiar
Last Time: Filtering Numbers

; filter-nums : (num -> bool) list-of-num
; -> list-of-num
(define (filter-nums PRED l)
  (cond
   [(empty? l) empty]
   [(cons? l)
    (local [(define r
               (filter-nums PRED (rest l)))]
     (cond
      [(PRED (first l))
       (cons (first l) r)]
      [else r]))))))

How do we avoid cut and paste?
Filtering Lists

We know this function will work for both number and symbol lists:

\[
\text{; filter : ... (define (filter PRED l)
  (cond
    [(empty? l) empty]
    [(cons? l)
      (local [(define r
        (filter PRED (rest l)))]
        (cond
          [(PRED (first l))
            (cons (first l) r)]
          [else r]))]))}
\]

But what is its contract?
The Contract of Filter

How about this?

\[(\text{num-OR-sym} \rightarrow \text{bool}) \text{ list-of-num-OR-list-of-sym} \rightarrow \text{list-of-num-OR-list-of-sym}\]

; A num-OR-sym is either
;   - num
;   - sym

; A list-of-num-OR-list-of-sym is either
;   - list-of-num
;   - list-of-sym
The Contract of Filter

How about this?

\[(\text{num-OR-sym} \rightarrow \text{bool}) \ \text{list-of-num-OR-list-of-sym} \rightarrow \text{list-of-num-OR-list-of-sym}\]

This contract is too weak to define \texttt{eat-apples}

\[
; \texttt{eat-apples} : \text{list-of-sym} \rightarrow \text{list-of-sym}
\]
\[
(\text{define} \ (\texttt{eat-apples} \ l) \n  \ (\text{filter} \ \texttt{not-apple?} \ l))
\]

\[
; \texttt{not-apple?} : \text{sym} \rightarrow \text{bool}
\]
\[
(\text{define} \ (\texttt{not-apple?} \ s) \n  \ (\text{not} \ (\text{symbol=?} \ s \ \texttt{'apple})))
\]

\texttt{eat-apples} must return a \texttt{list-of-sym}, but by its contract, \texttt{filter} might return a \texttt{list-of-num}
The Contract of Filter

How about this?

\[(\text{num-OR-sym} \to \text{bool}) \ \text{list-of-num-OR-list-of-sym} \not\Rightarrow \text{list-of-num-OR-list-of-sym}\]

This contract is too weak to define \texttt{eat-apples}

\[
\text{; eat-apples : list-of-sym} \to \text{list-of-sym}
\]

\[
\text{} \text{(define (eat-apples } \text{l}) \text{)}
\]

\[
\text{} \text{(filter not-apple? } \text{l})\)
\]

\[
\text{; not-apple? : sym} \to \text{bool}
\]

\[
\text{} \text{(define (not-apple? } \text{s}) \text{)}
\]

\[
\text{} \text{(not (symbol=? } \text{s 'apple}))\)
\]

\textit{not-apple?} only works on symbols, but by its contract \texttt{filter}

\textit{might give it a num}
The Contract of Filter

The reason \texttt{filter} works is that if we give it a \texttt{list-of-sym},
then it returns a \texttt{list-of-sym}

Also, if we give \texttt{filter} a \texttt{list-of-sym}, then it calls \texttt{PRED} with
symbols only

A better contract:

\begin{verbatim}
filter :
  ((num -> bool) list-of-num
   -> list-of-num)
OR
  ((sym -> bool) list-of-sym
   -> list-of-sym)
\end{verbatim}

But what about a list of \texttt{images}, \texttt{posns}, or \texttt{snakes}?
The True Contract of Filter

The real contract is

\[ \text{filter} : ((X \rightarrow \text{bool}) \text{ list-of-X} \rightarrow \text{list-of-X}) \]

where \( X \) stands for any type

- The caller of \text{filter} gets to pick a type for \( X \)
- All \( Xs \) in the contract must be replaced with the same type

Data definitions need type variables, too:

\[
\begin{align*}
; \text{A list-of-X is either} \\
; & - \text{empty} \\
; & - (\text{cons } X \text{ list-of-X})
\end{align*}
\]
Using Filter

The `filter` function is so useful that it’s built in

```
(define (eat-apples l)
  (local [(define (not-apple? s)
             (not (symbol=? s 'apple)))]
    (filter not-apple? l)))
```
Looking for Other Built-In Functions

Recall \texttt{feed-fish}:

\begin{verbatim}
; feed-fish : list-of-num -> list-of-num
(define (feed-fish l)
  (cond
    [(empty? l) empty]
    [else (cons (+ 1 (first l))
               (feed-fish (rest l)))]))
\end{verbatim}

Is there a built-in function to help?

\textbf{Yes: map}
Using Map

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

; feed-fish : list-of-num -> list-of-num
(define (feed-fish l)
  (local [(define (feed-one n)
            (+ n 1))]
         (map feed-one l)))

; feed-animals : list-of-animal -> list-of-animal
(define (feed-animals l)
  (map feed-animal l))
```
The Contract for Map

\[
\text{(define (map CONV l)}
\begin{align*}
\text{ (cond)} \\
\text{ [(empty? l) empty]} \\
\text{ [else (cons (CONV (first l)} \\
\text{ (map CONV (rest l)))]))]}
\end{align*}
\]

- The \text{l} argument must be a list of \text{X}
- The \text{CONV} argument must accept each \text{X}
- If \text{CONV} returns a new \text{X} each time, then the contract for \text{map} is

\[
\text{map : (X \rightarrow X) list-of-X \rightarrow list-of-X}
\]
Posns and Distances

; distances : list-of-posn -> list-of-num
(define (distances l)
  (cond
    [(empty? l) empty]
    [(cons? l) (cons (distance-to-0 (first l))
      (distances (rest l)))]))

The distances function looks just like map, except that distances-to-0 is

    posn -> num

not

    posn -> posn
The True Contract of Map

Despite the contract mismatch, this works:

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

The true contract of `map` is

```
map : (X -> Y) list-of-X -> list-of-Y
```

The caller gets to pick both `X` and `Y` independently
More Uses of Map

; flip-posns : list-of-posn -> list-of-posn
(define (rsvp l)
    ; replaces 4 lines:
    (map flip-posn l))

; flip-posn : posn -> posn
    ....
More Uses of Map

; align-bricks : list-of-num -> list-of-num
(define (align-bricks lon)
  ; replaces 4 lines:
  (map round lon))
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
...

Folding a List

How about \texttt{sum}?

\[
\texttt{sum : list-of-num} \rightarrow \texttt{num}
\]

Doesn’t return a list, so neither \texttt{filter} nor \texttt{map} help

Abstracting over \texttt{sum} and \texttt{product} leads to \texttt{combine-nums}:

\[
; \texttt{combine-nums : list-of-num }\rightarrow\texttt{num}
; (\texttt{num num} \rightarrow \texttt{num}) \rightarrow \texttt{num}
\]

\[
\texttt{(define (combine-nums l base-n COMB)}
\]

\[
\texttt{(cond}
\]

\[
[\texttt{(empty? l)} \texttt{base-n}]
\]

\[
[\texttt{(cons? l)}
\]

\[
\texttt{(COMB}
\]

\[
\texttt{(first l)}
\]

\[
\texttt{(combine-nums (rest l) base-n COMB))}]
\]

\[
\texttt{))))\]
The Foldr Function

; foldr : (X Y -> Y) Y list-of-X -> Y
(define (foldr COMB base l)
  (cond
    [(empty? l) base]
    [(cons? l)
     (COMB (first l)
           (foldr COMB base (rest l))))]))

Thesum and product functions become trivial:

(define (sum l) (foldr + 0 l))
(define (product l) (foldr * 1 l))
The Foldr Function

; foldr : (X Y -> Y) Y list-of-X -> Y
(define (foldr COMB base l)
  (cond
   [(empty? l) base]
   [(cons? l)
    (COMB (first l)
      (foldr COMB base (rest l))))]]))

; total-distance : list-of-posn -> num
(define (total-distance l)
  (local [(define (add-distance p n)
            (+ (distance-to-0 p) n))
    (foldr add-distance 0 l))]
The Foldr Function

; foldr : (X Y -> Y) Y list-of-X -> Y
(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 empty l)))
The Foldr Function

; foldr : (X Y -> Y) Y list-of-X -> Y
(define (foldr COMB base l)
  (cond
   [(empty? l) base]
   [(cons? l)
    (COMB (first l)
       (foldr COMB base (rest l))))]))

Yes, filter too:

(define (filter f l)
  (local [(define (check i r)
            (cond
             [(f i) (cons i r)]
             [else r)]))]
  (foldr check empty l)))
The Source of Foldr

How can foldr be so powerful?
The Source of Foldr

Template:

```scheme
(define (func-for-loX l)
  (cond
    [(empty? l) ...]
    [(cons? l) ... (first l)
      ... (func-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 \textbf{foldr}:

\begin{verbatim}
ormap : (X -> bool) list-of-X -> bool
andmap : (X -> bool) list-of-X -> bool
\end{verbatim}

Examples:

\begin{verbatim}
; got-milk? : list-of-sym -> bool
(define (got-milk? l)
  (local [(define (is-milk? s)
               (symbol=? s 'milk))]
    (ormap is-milk? l)))

; all-passed? : list-of-grade -> bool
(define (all-passed? l)
  (andmap passing-grade? l))
\end{verbatim}
What about Non-Lists?

Since it’s based on the template, the concept of fold is general

; fold-ftn : (sym num sym Z Z -> Z) Z ftn -> Z
(define (fold-ftn COMB base ftn)
  (cond
   [(empty? ftn) base]
   [(child? ftn)
    (COMB (child-name ftn) (child-date ftn) (child-eyes ftn)
       (fold-ftn COMB BASE (child-father ftn))
       (fold-ftn COMB BASE (child-mother ftn)))]])

(define (count-persons ftn)
  (local [(define (add name date color c-f c-m)
           (+ 1 c-f c-m)]
          (fold-ftn add 0 ftn)))

(define (in-family? who ftn)
  (local [(define (here? name date color in-f? in-m?)
           (or (symbol=? name who) in-f? in-m?)]
          (fold-ftn here? false ftn)))
Anonymous Functions
Values and Names

Some Values:

• Numbers: 1, 17.8, 4/5

• Booleans: true, false

• Lists: empty, (cons 7 empty)

• ...

• Function names: less-than-5, first-is-apple?

given

(define (less-than-5? n) ...)
(define (first-is-apple? a b) ...)

Why do only function values require names?
Naming Everything

Having to name every kind of value would be painful:

```lisp
(local [(define (first-is-apple? a b)
          (symbol=? a 'apple))]
       (choose '(apple banana)
                '(cherry cherry)
                first-is-apple?))
```

would have to be

```lisp
(local [(define (first-is-apple? a b)
          (symbol=? a 'apple))
       (define al '(apple banana))
       (define bl '(cherry cherry))]
       (choose al bl first-is-apple?))
```

Fortunately, we don’t have to name lists
Naming Nothing

Can we avoid naming functions?

In other words, instead of writing

```
(local [(define (first-is-apple? a b)
           (symbol=? a 'apple))]
      ... first-is-apple? ...)
```

we’d like to write

```
... function that takes a and b
      and produces (symbol=? a 'apple)
... 
```

We can do this in Intermediate with Lambda
Lambda

An **anonymous function** value:

\[
\text{(lambda (a b) (symbol=? a 'apple))}
\]

Using `lambda` the original example becomes

\[
\text{(choose '(apple banana)
\quad '(cherry cherry)
\quad (lambda (a b) (symbol=? a 'apple)))}
\]

The funny keyword `lambda` is an 80-year-old convention: the Greek letter \( \lambda \) means “function”
Using Lambda

In DrRacket:

```
> (lambda (x) (+ x 10))
(l lambda (a1) ...)
```

Unlike most kinds of values, there’s no one shortest name:

• The argument name is arbitrary

• The body can be implemented in many different ways

So DrRacket gives up — it invents argument names and hides the body
Using Lambda

In DrRacket:

> ((lambda (x) (+ x 10)) 17)
27

The function position of an **application** (i.e., function call) is no longer always an identifier

Some former syntax errors are now run-time errors:

> (2 3)

*procedure application: expected procedure, given 2*
Defining Functions

What’s the difference between

\[
(define \ (f \ a \ b) \\
 \quad (+ \ a \ b))
\]

and

\[
(define \ f \ (lambda \ (a \ b) \\
 \quad (+ \ a \ b)))
\]

? 

Nothing — the first one is (now) a shorthand for the second
Lambda and Built-In Functions

Anonymous functions work great with \texttt{filter}, \texttt{map}, etc.:

\begin{verbatim}
(define (eat-apples l)
  (filter (lambda (a)
            (not (symbol=? a 'apple)))
           l))

(define (inflate-by-4% l)
  (map (lambda (n) (* n 1.04)) l))

(define (total-blue l)
  (foldr (lambda (c n)
          (+ (color-blue c) n))
         0 l))
\end{verbatim}
Functions that Produce Functions

We already have functions that take function arguments

\[
\text{map} : (X \rightarrow Y) \text{ list-of-X} \rightarrow \text{list-of-Y}
\]

How about functions that \textit{produce} functions?

Here’s one:

\[
; \text{make-adder} : \text{num} \rightarrow (\text{num} \rightarrow \text{num})
\]

\[
\text{(define (make-adder n)}
\]

\[
\quad (\text{lambda (m) (+ m n)))
\]

\[
\text{(map (make-adder 10) '(1 2 3))}
\]

\[
\text{(map (make-adder 11) '(1 2 3))}
\]
Using Functions that Produce Functions

Suppose that we need to filter different symbols:

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

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

\[
; \text{mk-is-sym} : \text{sym} \rightarrow (\text{sym} \rightarrow \text{bool}) \\
(\text{define } \text{mk-is-sym} \ s) \\
\quad (\lambda a \ (\text{symbol}=? s a))
\]

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

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

This `curry` function curries any 2-argument function:

```
; curry : (X Y -> Z) -> (X -> (Y -> Z))
(define (curry f)
  (lambda (v1)
    (lambda (v2)
      (f v1 v2)))))

(define mk-is-sym (curry symbol=?))

(filter (mk-is-sym 'apple) l)
(filter (mk-is-sym 'banana) l)
(filter (mk-is-sym 'cherry) l)
```
Currying Functions

This **curry** function curries any 2-argument function:

```
; curry : (X Y -> Z) -> (X -> (Y -> 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)
```
Composing Functions

But we want non-symbols

; compose (Y -> Z) (X ->Y) -> (X -> Z)
(define (compose f g)
  (lambda (x) (f (g x)))))

(filter (compose not
         ((curry symbol=?) 'apple))
       l)
Uncurrying Functions

Sometimes it makes sense to **uncurry**:

\[
; \text{curry} : (X \rightarrow (Y \rightarrow Z)) \rightarrow (X \ Y \rightarrow Z)
\]

\[
\text{(define (uncurry f)}
\]

\[
(\text{lambda (v1 v2)}
\]

\[
((f \ v1) \ v2))
\]

\[
\text{(define (map f l)}
\]

\[
(\text{foldr (uncurry (compose (curry cons) f))}
\]

\[
\text{empty l))}
\]

\[
\text{(define (total-blue l)}
\]

\[
(\text{foldr (uncurry (compose (curry +)}
\]

\[
\text{color-blue))}
\]

\[
0 \ l))
\]
Lambda in Math

; derivative : (num -> num) -> (num -> num)
(define (derivative f)
  (lambda (x)
    (/ (- (f (+ x delta)))
      (f (- x delta)))
    (* 2 delta)))
(define delta 0.0001)

(define (square n) (* n n))
(((derivative square) 10)

Produces roughly 20, because the derivative of \(x^2\) is 2x
Graphical User Interfaces (GUIs) often use functions as values, including anonymous functions

*Java equivalent: inner classes*

Button click ⇒ update bottom text
GUI Library

make-text : string -> gui-item

text-contents : gui-item -> string

make-message : string -> gui-item

draw-message : gui-item string -> bool

make-button : string (event -> bool) -> gui-item

create-window : list-of-list-of-gui-item -> bool
(define (greet what)
    (draw-message greet-msg
        (string-append
            what "", "
            (text-contents name-field)))))

(define name-field
    (make-text "Name:"))
(define hi-button
    (make-button "Hello" (lambda (evt) (greet "Hi"))))
(define bye-button
    (make-button "Goodbye" (lambda (evt) (greet "Bye"))))
(define greet-msg
    (make-message ""))
(define (mk-greet what)
  (lambda (evt)
    (draw-message greet-msg
      (string-append
        what "", "
        (text-contents name-field))))
)

(define name-field
  (make-text "Name:")
)

(define hi-button
  (make-button "Hello" (mk-greet "Hi")))

(define bye-button
  (make-button "Goodbye" (mk-greet "Bye")))

(define greet-msg
  (make-message "______________________"))