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))
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])))])
```
Big Fish

Better with \texttt{local}:

\begin{verbatim}
; big : list-of-nums \rightarrow 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]))]))
\end{verbatim}

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]))]))
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?
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))])]]
)

cut and paste

; 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))])])

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]))]))

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)])))))

; 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]))]))

Avoid cut and paste!

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]))))
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:
Now we want the small fish:

; 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))
Small Fish

Now we want the small fish:

\[
\begin{align*}
; \text{smaller} : \text{list-of-nums num} & \rightarrow \text{list-of-nums} \\
\text{(define \ (smaller \ l \ n)} \\
& \text{(cond} \\
& \hspace{1em} [(\text{empty?} \ l) \ \text{empty}] \\
& \hspace{1em} [(\text{cons?} \ l) \\
& \hspace{2em} \text{(local \ [(\text{define} \ r \ (\text{smaller} \ \text{(rest \ l) \ n)})]} \\
& \hspace{3em} \text{(cond} \\
& \hspace{4em} [(< \ \text{first \ l) \ n} \\
& \hspace{5em} \text{(cons \ (first \ l) \ r)] \\
& \hspace{4em} [\text{else} \ r])])]) \\
& \text{(define \ (small \ l) \ (smaller \ l \ 5))}
\end{align*}
\]
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 <))
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

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

works because \textit{functions are values}
Functions as Values

The definition

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

works because \textit{functions are values}

- \textit{10} is a \texttt{num}
- \texttt{false} is a \texttt{bool}
Functions as Values

The definition

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

works because *functions are values*

- **10** is a **num**
- **false** is a **bool**
- **<** is a **(num num -> bool)**
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 \ \rightarrow \ bool)}

So the contract for \textbf{sized} is

\[; \text{list-of-nums num (num num \ \rightarrow \ bool)} \]
\[; \rightarrow \text{list-of-nums}\]
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 l) (sized l 0 btw-3-and-7))
Mediumish Fish

; btw-3-and-7 : num num -> bool
(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

; btw-3-and-7 : num num -> bool
(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

; 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]))]]))
; 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))
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)))
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

(define (bigger-than l m)
  (local [(define (more-than-m n)
          (> n m))]
    (filter-nums more-than-m l)))

(define (big l) (bigger-than l 5))
(define (huge l) (bigger-than l 10))
Big Example

...  
(define (bigger-than l m)
    (local [(define (more-than-m n)
                 (> n m))]
        (filter_nums more-than-m l)))
(define (big l) (bigger-than l 5)) ...
(big '(7 4 9))
(huge '(7 4 9))
Big Example

...  
(define (bigger-than 1 m)  
  (local [(define (more-than-m n)  
             (> n m))])  
  (filter-nums more-than-m 1)))  
(define (big 1) (bigger-than 1 5)) ...
(big '(7 4 9))
(huge '(7 4 9))

→

...  
(define (bigger-than 1 m)  
  (local [(define (more-than-m n)  
             (> n m))])  
    (filter-nums more-than-m 1)))  
...  
(bigger-than '(7 4 9) 5)
(huge '(7 4 9))
Big Example

...  
(define (bigger-than l m)
    (local [(define (more-than-m n)
                 (> n m))]
        (filter_nums more-than-m l)))
...  
(bigger-than '(7 4 9) 5)
(huge '(7 4 9))
Big Example

... 
(define (bigger-than l m)
  (local [(define (more-than-m n)
           (> n m))]
    (filter-nums more-than-m l)))
...
(bigger-than '(7 4 9) 5)
(huge '(7 4 9))

→

...
(local [(define (more-than-m n)
         (> n 5))]
    (filter-nums more-than-m '(7 4 9)))
(huge '(7 4 9))
Big Example

...(local [(define (more-than-m n)
               (> n 5))]
      (filter-nums more-than-m '(7 4 9)))
(huge '(7 4 9))
Big Example

...  
(local [(define (more-than-m n)  
    (> n 5))]  
  (filter-nums more-than-m '(7 4 9)))  
(huge '(7 4 9))

→

...
(define (more-than-m42 n)  
  (> n 5))  
(filter-nums more-than-m42 '(7 4 9))  
(huge '(7 4 9))
Big Example

...  
(define (more-than-m42 n)
    (> n 5))
(filter-nums more-than-m42 '(7 4 9))
(huge '(7 4 9))
Big Example

...  
(define (more-than-m42 n)  
  (> n 5))  
(filter-nums more-than-m42 '(7 4 9))  
(huge '(7 4 9))

→

...  
(define (more-than-m42 n)  
  (> n 5))  
'(7 9)  
(huge '(7 4 9))

after many steps
...  
(define (more-than-m42 n)  
   (> n 5))  
'(7 9)  
(huge '(7 4 9))
Big Example

... (define (more-than-m42 n)
    (> n 5))
'(7 9)
(huge '(7 4 9))

→

... (define (bigger-than l m)
    (local [(define (more-than-m n)
               (> n m))]
            (filter-nums more-than-m l)))

...(define (more-than-m42 n)
    (> n 5))
'(7 9)
(bigger-than '(7 4 9) 10)
Big Example

...  
(define (bigger-than l m)
  (local [(define (more-than-m n)
          (> n m))]
   (filter-nums more-than-m l)))
...
(define (more-than-m42 n)
  (> n 5))
'(7 9)
(bigger-than '(7 4 9) 10)
Big Example

... 
(define (bigger-than l m)
  (local [(define (more-than-m n)
           (> n m))]
          (filter-nums more-than-m l)))
...
(define (more-than-m42 n)
  (> n 5))
'(7 9)
(bigger-than '(7 4 9) 10)

→

...
(define (more-than-m42 n)
  (> n 5))
'(7 9)
(bigger-than '(7 4 9) 10)

(local [(define (more-than-m n)
          (> n 10))]
         (filter-nums more-than-m '(7 4 9)))
Big Example

...  
(define (more-than-m42 n)
  (> n 5))
'(7 9)
(local [(define (more-than-m n)
  (> n 10))]
  (filter-nums more-than-m '(7 4 9)))
Big Example

...  
(define (more-than-m42 n)  
  (> n 5))  
'(7 9)  
(local [(define (more-than-m n)  
          (> n 10))]  
  (filter-nums more-than-m '(7 4 9)))

→

...
(define (more-than-m42 n)  
  (> n 5))  
'(7 9)  
(define (more-than-m79 n)  
  (> n 10))  
(filter-nums more-than-m79 '(7 4 9))

Etc.
Abstraction

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