Today

1. Add top-level function defines to the Book language
   ◦ not in the book

Before we implement local functions...

2. How to design better programs with local functions
   ◦ also not in the book, but in *HtDP*
Concrete syntax:

\[
\begin{align*}
<\text{prog}> & \ ::= \ \{ \ <\text{id}> \ <\text{funcdef}> \ \} \ \ast \ \text{in} \ <\text{expr}> \\
<\text{funcdef}> & \ ::= \ (\ <\text{id}> \ast ) = <\text{expr}> \\
<\text{expr}> & \ ::= \ (\ <\text{id}> \ <\text{expr}> \ast )
\end{align*}
\]

\[\text{identity}(x) = x\]

\[\text{in} \ (\text{identity} \ 7)\]
Top-Level Procedure Definitions

Concrete syntax:

\[
\begin{align*}
\text{<prog>} & ::= \{ \text{id} \ <\text{funcdef}> \}^* \text{ in } \text{<expr>} \\
\text{<funcdef>} & ::= (\text{id}^*) = \text{<expr>} \\
\text{<expr>} & ::= (\text{id} \ <\text{expr}>^*)
\end{align*}
\]

\[
\begin{align*}
\text{fact}(n) = \text{if n then } *(n, (\text{fact} - (n, 1))) \text{ else 1} \\
\text{identity}(x) = x \\
\text{in} \ (\text{identity} \ (\text{fact} \ 10))
\end{align*}
\]
Top-Level Procedure Definitions

Abstract syntax:

\[
\begin{align*}
<\text{prog}> & ::= (\text{a-program} \\
& \quad (\text{list } <\text{id}>^*) (\text{list } <\text{funcdef}>^*) <\text{expr}>) \\
<\text{funcdef}> & ::= (\text{a-funcdef} (\text{list } <\text{id}>^*) <\text{expr}>) \\
<\text{expr}> & ::= (\text{app-exp } <\text{id}> (\text{list } <\text{expr}>^*)) \n\end{align*}
\]

- When evaluating a procedure application, we'll need a way to find a defined procedure

  - Use an environment (so we have two: local and top-level)
Implementing Top-Level Procedure Definitions

(implement in DrScheme)
Let's open an aquarium

- At first, we only care about the weight of each fish
- Represent a fish as a number
- Represent the aquarium as a list of numbers
- Functions include **big**, which takes an aquarium and returns only the fish bigger than 5 pounds
Aquarium Functions

Start with a template (generic):

;; lon-function : <list-of-num> → <????>
(define (lon-function l)
  (cond
    [(null? l) ...]
    [(pair? l) ... (car l) ... (lon-function (cdr l)) ...]))
Getting the Big Fish

;; big : <l-o-n> → <l-o-n>
(define (big l)
  (cond
    [(null? l) '()]
    [(pair? l)
      (cond
        [(> (car l) 5) (cons (car l) (big (cdr l)))]
        [else (big (cdr l))]]))

(big '(2 4 10)) → '(10)
Getting the Small Fish

;; small : <l-o-n> → <l-o-n>
(define (small l)
  (cond
    [(null? l) '()]
    [(pair? l)
      (cond
        [(< (car l) 5) (cons (car l) (small (cdr l)))]
        [else (small (cdr l))])]]

• Tiny changes to big, so cut-and-paste old code?
A Note on Cut and Paste

When you cut and paste code, you cut and paste bugs

*Avoid cut-and-paste whenever possible!*

- Alternative to cut and paste: *abstraction*
Filtering Fish

;; filter-fish : (<num> <num> → <bool>) <l-o-n> → <l-o-n>
(define (filter-fish OP l)
  (cond
    [(null? l) '()]
    [(pair? l)
      (cond
        [(OP (car l) 5) (cons (car l) (filter-fish OP (cdr l)))]
        [else (filter-fish OP (cdr l))]])
    ])

(define (big l) (filter-fish > l))

(define (small l) (filter-fish < l))
More Filters

• Medium fish?

No problem:

\[
\text{(define (medium } l) \ (\text{filter-fish = } l))
\]
More Filters

• How about fish that are *roughly* medium, between 4 and 6 pounds?

```scheme
(close-to : <num> <num> → <bool>
(define (close-to n m)
  (and (>= n (- m 1)) (<= n (+ m 1))))

(define (roughly-medium l) (filter-fish close-to l))
```

Remember: *function names are values!*

*Note the contract for close-to*
More Filters

• How about 2-pound fish?

Abstract filter-fish with respect to the number 5?

;; filter-fish : ... <num> <l-o-n> → <l-o-n>
(define (filter-fish OP N l)
  (cond
    [(null? l) '()]  
    [(pair? l)
      (cond
        [(OP (car l) N) (cons (car l) (filter-fish OP N (cdr l)))]
        [else (filter-fish OP N (cdr l))])]))
More Filters

• How about 2-pound fish?

Abstract filter-fish with respect to the number 5?

• How about fish that are either 2 pounds or 4 pounds?

Actually, we can write either of those already:

\[
(\text{define } (\text{size-2-or-4 } n m) \\
(\text{or } (= n 2) (= n 4)) ; \text{ignores } m)
\]

\[
(\text{define } (2-\text{-or-4-fish } l) (\text{filter-fish size-2-or-4 } l))
\]

This suggests a simplification of filter-fish
Filter

;;; filter : (<num> → <bool>) <l-o-n> → <l-o-n>
(define (filter PRED l)
  (cond
    [(null? l) '()]
    [(pair? l)
      (cond
        [(PRED (car l)) (cons (car l) (filter PRED (cdr l)))]
        [else (filter PRED (cdr l))])]
    [else (filter PRED (cdr l))])
)

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

(define (big l) (filter greater-than-5 l))
Local Helpers

Since only **big** needs to use **greater-than-5**, make it local:

```
(define (big l)
  (let ([greater-than-5 (lambda (n) (> n 5))])
    (filter greater-than-5 l)))
```

• Suppose we move to Texas, where "big" means more than 10 pounds

```
(define (texas-big l)
  (let ([greater-than-10 (lambda (n) (> n 10))])
    (filter greater-than-10 l)))
```

*More cut-and-paste?!
Abstraction over Local Functions

(define (relatively-big l m)
  (let ([greater-than-m (lambda (n) (> n m))])
    (filter greater-than-m l)))

(define (big l) (relatively-big l 5))
(define (texas-big l) (relatively-big l 10))

(big '(2 4 8 11)) = '(8 11)
(texas-big '(2 4 8 11)) = '(11)

How does that work?
(define (rel-big l m)
  (let ([gt-m (λ (n) (> n m))])
    (filter gt-m l)))

(define (big l)
  (rel-big l 5))

(big '(2 4 8))

(define (rel-big l m)
  (let ([gt-m (λ (n) (> n m))])
    (filter gt-m l)))

(define (big l)
  (rel-big l 5))

(rel-big '(2 4 8) 5)
Evaluation with Local Functions

\[
\text{(define (rel-big l m)} \quad \rightarrow \quad \text{(define (rel-big l m)}
\]
\[
\text{ (let ([gt-m (\lambda (n) (> n m))])} \quad \text{(let ([gt-m (\lambda (n) (> n m))])}
\]
\[
\text{ (filter gt-m l))} \quad \text{(filter gt-m l))}
\]
\[
\text{(define (big l)} \quad \text{(define (big l)}
\]
\[
\text{ (rel-big l 5))} \quad \text{(rel-big l 5))}
\]
\[
\text{(rel-big '(2 4 8) 5)} \quad \text{(let ([gt-m (\lambda (n) (> n 5))])}
\]
\[
\text{ (filter gt-m '(2 4 8)))}
\]
Evaluation with Local Functions

\[
\text{(define (rel-big l m)} \rightarrow \text{(define (rel-big l m)} \\
\text{\hspace{1cm} (let ([gt-m (\lambda (n) (> n m)))])} \hspace{1cm} (let ([gt-m (\lambda (n) (> n m)))] \\
\text{\hspace{2cm} (filter gt-m l))} \hspace{2cm} (filter gt-m l))
\]

\[
\text{(define (big l)} \hspace{1cm} (define (big l)} \\
\text{\hspace{1cm} (rel-big l 5))} \hspace{1cm} (rel-big l 5))
\]

\[
\text{(let ([gt-m (\lambda (n) (> n 5)))])} \hspace{1cm} \text{(define (gt-m\textsubscript{98} n) (> n 5)))} \\
\text{\hspace{2cm} (filter gt-m '(2 4 8))} \hspace{2cm} \text{(filter gt-m\textsubscript{98} '(2 4 8))}
\]

Every time we call \text{rel-big} we get a brand-new \text{gt-m}
Filter and Map

- A function like `filter` is so useful that it's usually built in
  - But not in the EoPL language, unfortunately
- Here's one that's even more useful (and is built in):

  ```scheme
  ;; map : ( <num> → <num> ) <list-of-num> → <list-of-num>
  (define (map F I)
    (cond
     [(null? I) '()]
     [else (cons (F (car I)) (map F (cdr I)))]))
  
  (map add1 '(1 2 3)) = '(2 3 4)
  ```
**Map, More Generally**

Actually, `map` is more general

```plaintext
;; map : (X → Y) list-of-X → list-of-Y

(map even? '(1 2 3)) = '(#f #t #f)
(map car '((1 2) (3 4) (5 6))) = '(1 3 5)
```

Actually, `map` is more general!

```plaintext
;; map : (X₁ ... Xₙ → Y) l-o-X₁ ... l-o-Xₙ → l-o-Y

(map + '(1 2 3) '(4 5 6)) = '(5 7 9)
(map cons '(1 2 3) '('#f #f #t)) = '((1 . #f) (2 . #f) (3 . #t))
```
Why must functions always have a name?