## 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*

## Top-Level Procedure Definitions

Concrete syntax:

```
<prog>     ::=  { <id> <funcdef> } * in <expr>
<funcdef>  ::=  (<id>*) = <expr>
<expr>     ::=  (<id> <expr>*)
```

**identity**(**x**) = **x**
**in** (**identity** 7)

## Top-Level Procedure Definitions

Concrete syntax:

```
<prog>     ::=  { <id> <funcdef> } * in <expr>
<funcdef>  ::=  (<id>*) = <expr>
<expr>     ::=  (<id> <expr>*)
```

**fact**(**n**) = **if n then** *(**n**, (**fact** -(**n**, 1))) **else** 1
**identity**(**x**) = **x**
**in** (**identity** (**fact** 10))

## Top-Level Procedure Definitions

Abstract syntax:

```
<prog>     ::=  (a-program
                  (list <id>*) (list <funcdef>*) <expr>)
<funcdef>  ::=  (a-funcdef (list <id>*) <expr>)
<expr>     ::=  (app-exp <id> (list <expr>*))
```

- 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)

## How to Design Better Programs

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:

```
(define (medium l) (filter-fish = l))
```

## More Filters

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

```
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:

```
(define (size-2-or-4 n m)
  (or (= n 2) (= n 4))) ; ignores m

(define (2-or-4-fish l) (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))])]))

(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?

## Evaluation with Local Functions

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

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

(big '(2 4 8))                               (rel-big '(2 4 8) 5)
```

## Evaluation with Local Functions

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

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

(rel-big '(2 4 8) 5)                         (let ([gt-m (λ (n) (> n 5))])
                                               (filter gt-m '(2 4 8)))
```

## Evaluation with Local Functions

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

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

(let ([gt-m (λ (n) (> n 5))])
  (filter gt-m '(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))

(define (gt-m98 n) (> n 5))))

(filter gt-m98 '(2 4 8))
```

Every time we call **rel-big** we get a brand-new **gt-m**

## Filter and Map

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

```
;; map : (<num> → <num>) <list-of-num> → <list-of-num>
(define (map F l)
  (cond
    [(null? l) '()]
    [else (cons (F (car l)) (map F (cdr l)))]))

(map add1 '(1 2 3)) = '(2 3 4)
```

## Map, More Generally

Actually, **map** is more general

```
;; 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!

;; **map** : ($X_1$ ... $X_n$ → Y) l-o-$X_1$ ... l-o-$X_n$ → 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))
```

## Closing Thought

Why must functions always have a name?