Higher-order List Functions in Racket

CS251 Programming Languages
Fall 2017, Lyn Turbak
Department of Computer Science
Wellesley College

Higher-order List Functions

A function is **higher-order** if it takes another function as an input and/or returns another function as a result. E.g. app-3-5, make-linear-function, flip2.

We will now study **higher-order list functions** that capture the recursive list processing patterns we have seen.

Recall the List Mapping Pattern

\[(\text{map}\, F\, (\text{list}\ v1\ v2\ \ldots\ vn))\]

Express Mapping via Higher-order **my-map**

\[(\text{define}\ (\text{my-map}\ f\ xs))\]
\[
\begin{align*}
\text{if} & \ (\text{null?}\ x) \\
\text{null} & \\
\text{null} & \\
\text{(cons}\ (f\ (\text{first}\ x)) & \\
\text{(my-map}\ f\ (\text{rest}\ x))))
\end{align*}
\]
**my-map Examples**

> (my-map (λ (x) (* 2 x)) (list 7 2 4))

> (my-map first (list (list 2 3) (list 4) (list 5 6 7)))

> (my-map (make-linear-function 4 7) (list 0 1 2 3))

> (my-map app-3-5 (list sub2 + avg pow (flip pow)
  make-linear-function))

---

**Your turn**

(map-scale n nums) returns a list that results from scaling each number in nums by n.

> (map-scale 3 (list 7 2 4))
  "'21 6 12"

> (map-scale 6 (range 0 5))
  "'0 6 12 18 24"

---

**Currying**

A curried binary function takes one argument at a time.

(define (curry2
          binop)
          (λ (x)
              (λ (y) (binop x y))))

(define curried-mul (curry2 *)
> ((curried-mul 5) 4)

> (my-map (curried-mul 3) (list 1 2 3))

> (my-map (curried-mul 3) (list 1 2 3))

---

**Mapping with binary functions**

(define (map2 binop xs ys)
  (if (not (= (length xs) (length ys)))
      (error "map2 requires same-length lists")
      (if (or (null? xs) (null? ys))
          null
          (cons (binop (first xs) (first ys))
                  (map2 binop (rest xs) (rest ys))))))

> (map2 pow (list 2 3 5) (list 6 4 2))
  "'(64 81 25)

> (map2 cons (list 2 3 5) (list 6 4 2))
  "'(2 . 6) (3 . 4) (5 . 2))

> (map2 cons (list 2 3 4 5) (list 6 4 2))
  ERROR: map2 requires same-length lists
Built-in Racket map Function
Maps over Any Number of Lists

> (map (λ (x) (* x 2)) (range 1 5))
'(2 4 6 8)
> (map pow (list 2 3 5) (list 6 4 2))
'(64 81 25)
> (map (λ (a b x) (+ (* a x) b))
   (list 2 3 5) (list 6 4 2) (list 0 1 2))
'(6 7 12)
> (map pow (list 2 3 4 5) (list 6 4 2))
ERROR: map: all lists must have same size;
arguments were: '#<procedure:pow> '(2 3 4 5) '(6 4 2)

Recall the List Filtering Pattern

(filterP (list v1 v2 ... vn))

Express Filtering via Higher-order my-filter

(define (my-filter pred xs)
  (if (null? xs)
      null
      (if (pred (first xs))
        (cons (first xs) (my-filter pred (rest xs)))
        (my-filter pred (rest xs))))))

Built-in Racket filter function acts just like my-filter

(filter Examples

> (filter (λ (x) (> x 0)) (list 7 -2 -4 8 5))
> (filter (λ (n) (= 0 (remainder n 2)))
   (list 7 -2 -4 8 5))
> (filter (λ (xs) (> (len xs) 2))
   (list (list 2 3) (list 4) (list 5 6 7))
> (filter number?
   (list 17 #t 3.141 "a" (list 1 2) 3/4 5+6i))
> (filter (lambda (binop) (>=(app-3-5 binop)
                       (app-3-5 (flip2 binop))))
   (list sub2 + * avg pow (flip2 pow)))
Recall the Recursive List Accumulation Pattern

(\text{recf} \ (\text{list} \ v1 \ v2 \ \ldots \ vn))

Express Recursive List Accumulation via Higher-order \text{my-foldr}

\begin{align*}
&\text{(define } (\text{my-foldr} \ \text{combine} \ \text{nullval} \ xs) \\
&\quad (\text{if} \ (\text{null?} \ xs) \\
&\quad \quad \text{nullval} \\
&\quad \quad (\text{combine} \ (\text{first} \ xs) \\
&\quad \quad \quad (\text{my-foldr} \ \text{combine} \\
&\quad \quad \quad \quad \text{nullval} \\
&\quad \quad \quad \quad (\text{rest} \ xs))))))
\end{align*}

\text{my-foldr} \text{ Examples}

> (\text{my-foldr} \ + \ 0 \ (\text{list} \ 7 \ 2 \ 4))
> (\text{my-foldr} \ * \ 1 \ (\text{list} \ 7 \ 2 \ 4))
> (\text{my-foldr} \ - \ 0 \ (\text{list} \ 7 \ 2 \ 4))
> (\text{my-foldr} \ \text{min} \ +\text{inf.0} \ (\text{list} \ 7 \ 2 \ 4))
> (\text{my-foldr} \ \text{max} \ -\text{inf.0} \ (\text{list} \ 7 \ 2 \ 4))
> (\text{my-foldr} \ \text{cons} \ (\text{list} \ 8) \ (\text{list} \ 7 \ 2 \ 4))
> (\text{my-foldr} \ \text{append} \ \text{null} \\
&\quad (\text{list} \ (\text{list} \ 2 \ 3) \ (\text{list} \ 4) (\text{list} \ 5 \ 6 \ 7)))

\text{More my-foldr} \text{ Examples}

> (\text{my-foldr} \ \lambda \ (a \ b) \ (\text{and} \ a \ b)) \ #t \ (\text{list} \ #t \ #t \ #t))
> (\text{my-foldr} \ \lambda \ (a \ b) \ (\text{and} \ a \ b)) \ #t \ (\text{list} \ #t \ #f \ #t))
> (\text{my-foldr} \ \lambda \ (a \ b) \ (\text{or} \ a \ b)) \ #f \ (\text{list} \ #t \ #f \ #t))
> (\text{my-foldr} \ \lambda \ (a \ b) \ (\text{or} \ a \ b)) \ #f \ (\text{list} \ #f \ #f \ #f))

;; This doesn’t work. Why not?
> (\text{my-foldr} \ \text{and} \ #t \ (\text{list} \ #t \ #t \ #t))
Mapping & Filtering in terms of `my-foldr`

(define (my-map f xs)
  (my-foldr ???
    ???
    xs))

(define (my-filter pred xs)
  (my-foldr ???
    ???
    xs))

More `foldr` Examples

> (foldr + 0 (list 7 2 4))
13
> (foldr (lambda (a b sum) (+ (* a b) sum)) 0
  (list 2 3 4)
  (list 5 6 7))
56
> (foldr (lambda (a b sum) (+ (* a b) sum)) 0
  (list 1 2 3 4)
  (list 5 6 7))
ERROR: foldr: given list does not have the same size as the first list: '(5 6 7)