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` from the previous lecture.

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

Recall the List Mapping Pattern

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

Express Mapping via Higher-order `my-map`

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

```
> (my-map (λ (x) (* 2 x)) '(7 2 4))
'(14 4 8)

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

> (my-map (make-linear-function 4 7) '(0 1 2 3))
'(4 8 12 16)

> (my-map app-3-5 (list sub2 + avg pow (flip2 pow) make-linear-function))
```

**map-scale**

Define (map-scale \( n \) \( \text{nums} \)), which returns a list that results from scaling each number in \( \text{nums} \) by \( n \).

```
> (map-scale 3 '(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 \( \text{binop} \))
(λ (x) (λ (y) (binop x y))))

(define curried-\* (curry2 *))

> ((curried-\* 5) 4)
20

> (my-map (curried-\* 3) '(1 2 3))
'(3 6 9)

> (my-map (curry2 pow) 4) '(1 2 3)

> (my-map (curry2 (flip2 pow)) 4) '(1 2 3)

> (define LOL '((2 3) (4) (5 6 7)))

> (my-map ((curry2 cons) 8) LOL)
'((2 3 8) (4 8) (5 6 7 8))
```

**Mapping with binary functions**

```
(define (my-map2 \( \text{binop} \) \( \text{xs} \) \( \text{ys} \))
(if (or (null? \( \text{xs} \)) (null? \( \text{ys} \)))
  ;; result has length of shorter list
  null
  (cons (binop (first \( \text{xs} \)) (first \( \text{ys} \)))
  (my-map2 \( \text{binop} \) (rest \( \text{xs} \)) (rest \( \text{ys} \))))))

> (my-map2 pow '(2 3 5) '(6 4 2))
'(64 81 25)

> (my-map2 cons '(2 3 5) '(6 4 2))
'((2 . 6) (3 . 4) (5 . 2))

> (my-map2 + '(2 3 4 5) '(6 4 2))
'((2 . 6) (3 . 4) (5 . 2))
```
**Built-in Racket map Function**
Maps over Any Number of Lists

> (map (\( (x) (* x 2) \)) (range 1 5))
'(2 4 6 8)

> (map pow '(2 3 5) '(6 4 2))
'(64 81 25)

> (map (\( (a b x) (+ (* a x) b) \))
'(2 3 5) '(6 4 2) '(0 1 2))
'(6 7 12)

> (map pow '(2 3 4 5) '(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))

(v1 v2 ... vn)

---

**Express Filtering via Higher-order my-filter**

```racket
(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 (\( (x) (> x 0) \)) '(7 -2 -4 8 5))

> (filter (\( (n) (= 0 (remainder n 2)) \))
'(7 -2 -4 8 5))

> (filter (\( (xs) (>= (len xs) 2)) \))
'((2 3) (4) (5 6 7))

> (filter number? 
'(17 #t 3.141 "a" (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

\[
\text{(recf (list } v_1 \ v_2 \ \ldots \ v_n))
\]

\[
\begin{array}{c}
\text{combine} \\
\hline
v_1 \\
\rightarrow \ \\
\downarrow \\
\text{combine} \\
\hline \ldots \\
\rightarrow \ \\
\downarrow \\
\text{combine} \\
\hline
v_2 \\
\rightarrow \ \\
\downarrow \\
\text{combine} \\
\hline \ldots \\
\rightarrow \ \\
\downarrow \\
\text{nullval}
\end{array}
\]

\[
(\text{define (rec-accum } xs) \\
\text{(if (null? } xs) \\
\text{nullval} \\
\text{(combine (first } xs) \\
\text{(rec-accum (rest } xs)))))
\]

Express Recursive List Accumulation via Higher-order my-foldr

\[
\begin{array}{c}
\text{combine} \\
\hline
v_1 \\
\rightarrow \ \\
\downarrow \\
\text{combine} \\
\hline \ldots \\
\rightarrow \ \\
\downarrow \\
\text{combine} \\
\hline \ldots \\
\rightarrow \ \\
\downarrow \\
\text{combine} \\
\hline
v_2 \\
\rightarrow \ \\
\downarrow \\
\text{combine} \\
\hline \ldots \\
\rightarrow \ \\
\downarrow \\
\text{nullval}
\end{array}
\]

\[
(\text{define (my-foldr } combine \ \text{nullval} \ \text{vals}) \\
\text{(if (null? } \text{vals) \\
\text{nullval} \\
\text{(combine (first } \text{vals) \\
\text{(my-foldr } combine \\
\text{nullval} \\
\text{(rest } \text{vals)}))))}
\]

---

### my-foldr Examples

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

---

### More my-foldr Examples

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

;; This doesn’t work. Why not?
- \(\text{(my-foldr and #t (list #t #t #t))}\)
Your turn: sumProdList

Define sumProdList (from scope lecture) in terms of foldr.
Is let necessary here like it was in scoping lecture?

(define (sumProdList nums)
  (foldr ; combiner
    ; nullval
    nums))

Your turn: sumProdList

(define (sumProdList nums)
  (foldr ; combiner
    ; nullval
    nums))

Built-in Racket foldr Function
Folds over Any Number of Lists

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

Mapping & Filtering in terms of my-foldr

(define (my-map f xs)
  (my-foldr ; combiner
    ; nullval
    xs))

(define (my-filter pred xs)
  (my-foldr ; combiner
    ; nullval
    xs))

Problematic for foldr

(keepBiggerThanNext nums) returns a new list that keeps all nums that
are bigger than the following num. It never keeps the last num.

> (keepBiggerThanNext '(7 5 3 9 8))
'(7 5 9)
> (keepBiggerThanNext '(2 7 5 3 9 8))
'(7 5 9)
> (keepBiggerThanNext '(4 2 7 5 3 9 8))
'(4 7 5 9)

keepBiggerThanNext cannot be defined by fleshing out the following
template. Why not?

(define (keepBiggerThanNext nums)
  (foldr <combiner> <nullvalue> nums))
**keepBiggerThanNext with foldr**

keepBiggerThanNext needs (1) next number and (2) list result from below. With foldr, we can provide both #1 and #2, and then return #2 at end.

(define (keepBiggerThanNext nums)
  (second
    (foldr (λ (thisNum nextNum&subResult)
      (let {
          [nextNum (first nextNum&subResult)]
          [subResult (second nextNum&subResult)])
      (list thisNum ; becomes nextNum for elt to left
        (if (> thisNum nextNum)
          (cons thisNum subResult) ; keep
          subResult))))) ; don’t keep
    (list +inf.0 '()) ; +inf.0 guarantees last num
    ; in nums won’t be kept
    nums)))

**foldr-ternop: more info for combiner**

In cases like keepBiggerThanNext, it helps for the combiner to also take rest of list as an extra arg.

(foldr-ternop ternop nullval (list v1 v2 ... vn))

(define (foldr-ternop ternop nullval vals)
  (if (null? vals)
    nullval
    (ternop (first vals) ; arg #1
      (rest vals) ; extra arg # 2 to ternop
      ; arg #3
      (foldr-ternop ternop nullval (rest vals)))))

**keepBiggerThanNext with foldr-ternop**

(define (keepBiggerThanNext nums)
  (foldr-ternop
    ; combiner
    ; nullval
    nums))

> (keepBiggerThanNext '(4 2 7 5 3 9 8))
'(4 7 5 9)