

## Higher-Order List Functions in Racket

### SOLUTIONS



CS251 Programming  
Languages  
Spring 2019, Lyn Turbak

Department of Computer Science  
Wellesley College

## Higher-order List Functions

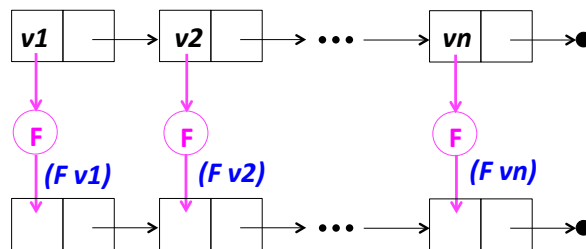
A function is **higher-order** if it takes another function as an input and/or returns another function as a result. E.g. `app-3-5`, `make-linear-function`, `flip2` from the previous lecture

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

Higher-order List Funs 2

## Recall the List Mapping Pattern

`(map F (list v1 v2 ... vn))`



```
(define (map F xs)
  (if (null? xs)
      null
      (cons (F (first xs))
            (map F (rest xs)))))
```

Higher-order List Funs 3

## Express Mapping via Higher-order `my-map`

Rather than defining a *list recursion pattern* for mapping, let's instead capture this pattern as a *higher-order list function* **my-map**:

```
(define (my-map f xs)
  (if (null? xs)
      null
      (cons (f (first xs))
            (my-map f (rest xs)))))
```

This way, we write the mapping list recursion function exactly once, and use it as many times as we want!

Higher-order List Funs 4

## my-map Examples Solutions



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

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

> (my-map (make-linear-function 4 7) '(0 1 2 3))
'(7 11 15 19)

> (my-map app-3-5 (list sub2 + avg pow (flip2 pow)
                    make-linear-function))
'(-2 8 4 243 125 #<procedure:...t-class-funs.rkt:17:4>)
```

Printed representation of  
procedure in Racket

## map-scale Solutions



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

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

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

```
(define (map-scale n nums)
  (my-map (λ (num) (* n num))
          nums))
```

## Currying Solutions



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)
20

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

> (my-map ((curry2 pow) 4) '(1 2 3))
'(4 16 64)

> (my-map ((curry2 (flip2 pow)) 4) '(1 2 3))
'(1 16 64)

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

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

> (my-map (curry2 snoc) 8) LOL ; fill in the blank
'((2 3 8) (4 8) (5 6 7 8))
```



Haskell Curry

## Mapping with binary functions

```
(define (my-map2 binop xs ys)
  (if (or (null? xs) (null? ys)) ; design decision:
      ; result has length of
      ; shorter list
      null
      (cons (binop (first xs) (first ys))
              (my-map2 binop (rest xs) (rest 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 (λ (x) (* x 2)) (range 1 5))
'(2 4 6 8)

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

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

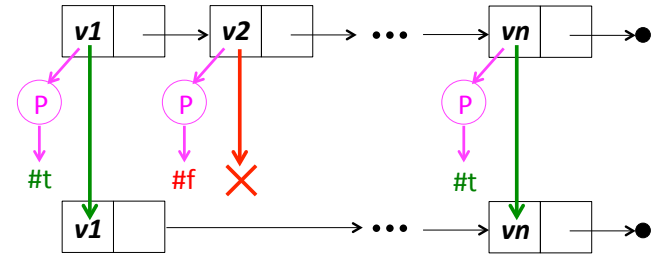
> (map pow '(2 3 4 5) '(6 4 2))
ERROR: map: all lists must have same size;
arguments were: #<procedure:pow> '(2 3 4 5) '(6 4 2)
```

Racket makes different design decision than `my-map2`: generate error when lists have different length

Higher-order List Funs 9

## Recall the List Filtering Pattern

```
(filter P (list v1 v2 ... vn))
```



```
(define (filter P xs)
  (if (null? xs)
      null
      (if (P (first xs))
          (cons (first xs) (filter P (rest xs)))
          (filter P (rest xs)))))
```

Higher-order List Funs 10

## Express Filtering via Higher-order `my-filter`

Similar to `my-map`, let's capture the filtering list recursion pattern via *higher-order list function* `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)))))
```

The built-in Racket `filter` function acts just like `my-filter`

Higher-order List Funs 11

## filter Examples Solutions



```
> (filter (λ (x) (> x 0)) '(7 -2 -4 8 5))
'(7 8 5)

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

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

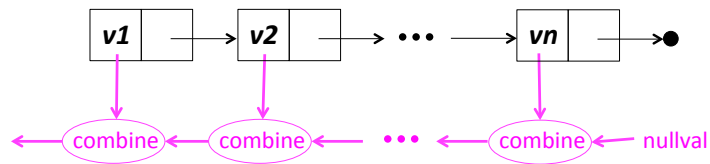
> (filter number? '(17 #t 3.141 "a" (1 2) 3/4 5+6i))
'(17 3.141 3/4 5+6i)

> (filter (lambda (binop) (>= (app-3-5 binop)
                              (app-3-5 (flip2 binop))))
          (list sub2 + * avg pow (flip2 pow)))
; The printed rep would show 4 #<procedure>s,
; but the returned list would be equivalent to
; (list + * avg pow)
```

Higher-order List Funs 12

## Recall the Recursive List Accumulation Pattern

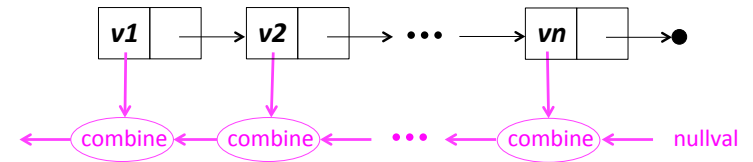
```
(recf (list v1 v2 ... vn))
```



```
(define (rec-accum xs)
  (if (null? xs)
      nullval
      (combine (first xs)
                (rec-accum (rest xs)))))
```

Higher-order List Funs 13

## Express Divide/Conquer/GlueList Recursion via Higher-order **my-foldr**



```
(define (my-foldr combine nullval vals)
  (if (null? vals)
      nullval
      (combine (first vals)
                (my-foldr combine
                           nullval
                           (rest vals)))))
```

This way, we never need to write another DCG list recursion!  
Instead, we instead just call **my-foldr** with the right arguments.

Higher-order List Funs 14

## my-foldr Examples Solutions



```
> (my-foldr + 0 '(7 2 4)) =>* 13 ; (+ 7 (+ 2 (+ 4 0)))
> (my-foldr * 1 '(7 2 4)) =>* 56 ; (* 7 (* 2 (* 4 1)))
> (my-foldr - 0 '(7 2 4)) =>* 9 ; (- 7 (- 2 (- 4 0)))
> (my-foldr min +inf.0 '(7 2 4))
=>* 2 ; (min 7 (min 2 (min 4 +inf.0)))
> (my-foldr max -inf.0 '(7 2 4))
=>* 7 ; (max 7 (max 2 (max 4 -inf.0)))
> (my-foldr cons '(8) '(7 2 4))
=>* '(7 2 4 8) ; (cons 7 (cons 2 (cons 4 '(8))))
> (my-foldr append null '((2 3) (4) (5 6 7))) =>* '(2 3 4 5 6 7)
; (append '(2 3) (append '(4) (append '(5 6 7) '())))
> (define (my-length L)
  (my-foldr (λ (fst sublen) (+ 1 sublen)) 0
            L)) ; fill in the blank
> (define (filter-positive nums)
  (my-foldr (λ (num subPoss)
              (if (> num 0) (cons num subPoss) subPoss))
            '()
            nums)) ; fill in the blank
```

Higher-order List Funs 15

## More my-foldr Examples Solutions



```
> (my-foldr (λ (fst subBool) (and fst subBool)) #t
  (list #t #t #t))
#t ; (and #t (and #t (and #t #t)))
> (my-foldr (λ (fst subBool) (and fst subBool)) #t
  (list #t #f #t))
#f ; (and #t (and #f (and #t #t)))
> (my-foldr (λ (fst subBool) (or fst subBool)) #f
  (list #t #f #t))
#t ; (or #t (or #f (or #t #t)))=
> (my-foldr (λ (fst subBool) (or fst subBool)) #f
  (list #f #f #f))
#f ; (or #f (or #f (or #f #f)))
;; This doesn't work. Why not?
> (my-foldr and #t (list #t #t #t))
Because and is a syntactic sugar keyword, not a first-class function
```

Higher-order List Funs 16



## Your turn: sumProdList Solutions

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

```
(sumProdList '(5 2 4 3)) -> '(14 . 120)
(sumProdList '()) -> '(0 . 1)
```

```
(define (sumProdList nums)
  (foldr (λ (num subPair) ; combiner
         (cons (+ num (car subPair))
               (* num (cdr subPair)))
         '(0 . 1) ; nullval
         nums))
; (1) Good idea to begin combiner (λ (num subPair) ... )
;     or λ with two other descriptive param names
; (2) Use "pretty printing" indentation to align
;     3 args to foldr and 2 args to cons
```

Higher-order List Funs 17



## Mapping & Filtering in terms of my-foldr Solutions

```
(define (my-map f xs)
  (my-foldr (λ (x subMap) ; combiner
            (cons (f x) subMap))
            '(); nullval
            xs))
```

```
(define (my-filter pred xs)
  (my-foldr (λ (x subFilter) ; combiner
            (if (pred x)
                (cons x subFilter)
                subFilter))
            '(); nullval
            xs))
```

Higher-order List Funs 18

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

```
> (foldr + 0 '(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))
```

Same design decision as in map

**ERROR:** foldr: given list does not have the same size as the first list: '(5 6 7)

Higher-order List Funs 19

## Problematic for foldr Solutions

`(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 1 3 9 5 4))
'(7 9 5)
> (keepBiggerThanNext '(2 7 1 3 9 5 4))
'(7 9 5)
> (keepBiggerThanNext '(6 2 7 1 3 9 5 4))
'(6 7 9 5)
```

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

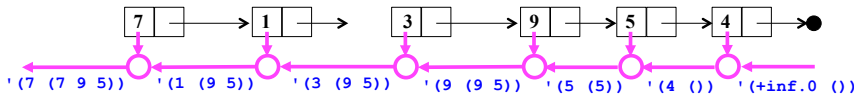
```
(define (keepBiggerThanNext nums)
  (foldr <combiner> <nullvalue> nums))
```

Because combiner can only use first of current list and result of recursively processing rest of list, but does not have access to rest of list itself, so cannot determine whether or not to keep first element.

Higher-order List Funs 20

## 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



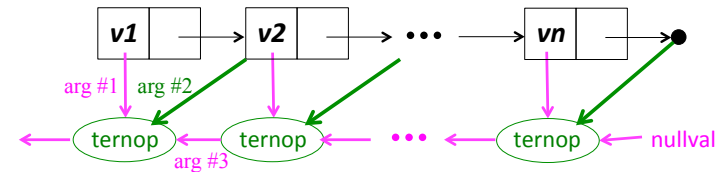
```
(define (keepBiggerThanNext nums)
  (second
    (foldr (λ (thisNum nextNum&subResult)
            (let {[nextNum (first nextNum&subResult)]
                  [subResult (second nextNum&subResult)]}
              (list thisNum ; becomes nextNum for elt to left
                    (if (> thisNum nextNum)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

Higher-order List Funs 21

## 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)))))
```

Higher-order List Funs 22

## keepBiggerThanNext with foldr-ternop Solutions



```
(define (keepBiggerThanNext nums)
  (foldr-ternop
    (λ (thisNum restNums subResult) ; combiner
      (if (null? restNums)
          ; special case for singleton list; *must*
          ; test restNums, not subResult, for null? Why?
          '()
          (if (> thisNum (first restNums))
              (cons thisNum subResult)
              subResult)))
    '() ; nullval
    nums))
```

```
> (keepBiggerThanNext '(6 2 7 1 3 9 5 4))
'(6 7 9 5)
```

Higher-order List Funs 23