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

\[
\text{map}_F \left( \text{list } v_1 \ v_2 \ ... \ v_n \right)
\]

\[
\begin{align*}
&v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \\
&F \left( F \ v_1 \right) \left( F \ v_2 \right) \left( F \ v_n \right)
\end{align*}
\]

\[
\begin{align*}
&(\text{define } (\text{map}_F \ xs)) \\
&(\text{if } (\text{null?} \ xs) \text{ null}) \\
&(\text{cons } (F \ (\text{first} \ xs)) \\
&(\text{map}_F \ (\text{rest} \ xs))))
\end{align*}
\]

### 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`:

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

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

**map-scale Solutions**

Define \((\text{map-scale} ~ n ~ \text{nums})\), 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)

(def (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))
'(1 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))

**Mapping with binary functions**

(define (my-map2 binop xs ys)
  (if (or (null? xs) (null? ys)) 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

```lisp
> (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.

**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`:

```lisp
(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`.

**Recall the List Filtering Pattern**

\[
\text{filter}_P \left( \text{list } v_1 \ v_2 \ldots \ v_n \right)
\]

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

**Filter Examples Solutions**

```lisp
> (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 + * pow (flip2 pow)))
; The printed rep would show 4 #<procedure>s,
; but the returned list would be equivalent to
; (list + * avg pow)
```
Recall the Recursive List Accumulation Pattern

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

\[
v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \rightarrow \bullet
\]

\[
\text{combine} \quad \cdot
\]

\[
\text{nullval} \quad \cdot
\]

\[
\text{(define } (\text{rec-accum } \text{xs})
\]

\[
(\text{if } (\text{null? } \text{xs})
\]

\[
\text{nullval}
\]

\[
(\text{combine } (\text{first } \text{xs})
\]

\[
(\text{rec-accum } (\text{rest } \text{xs})))
\]

Express Divide/Conquer/Glue List Recursion via Higher-order \textit{my-foldr}

\[
v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \rightarrow \bullet
\]

\[
\text{combine} \quad \cdot
\]

\[
\text{nullval} \quad \cdot
\]

\[
(\text{define } (\text{my-foldr } \text{combine} \text{ nullval} \text{ vals})
\]

\[
(\text{if } (\text{null? } \text{vals})
\]

\[
\text{nullval}
\]

\[
(\text{combine } (\text{first } \text{vals})
\]

\[
(\text{my-foldr } \text{combine} \text{ nullval}
\]

\[
(\text{rest } \text{vals})))
\]

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

\[
\text{my-foldr Examples Solutions}
\]

\[
> (\text{my-foldr} + 0 '(7 2 4)) \Rightarrow 13 ; (+ 7 (+ 2 (+ 4 0)))
\]

\[
> (\text{my-foldr} * 1 '(7 2 4)) \Rightarrow 56 ; (* 7 (* 2 (* 4 1)))
\]

\[
> (\text{my-foldr} - 0 '(7 2 4)) \Rightarrow 9 ; (- 7 (- 2 (- 4 0)))
\]

\[
> (\text{my-foldr} \text{ max } +\text{inf.0} '(7 2 4))
\]

\[
\Rightarrow 2 ; (\text{max } 7 (\text{min } 2 (\text{min } 4 +\text{inf.0})))
\]

\[
> (\text{my-foldr} \text{ min } -\text{inf.0} '(7 2 4))
\]

\[
\Rightarrow 7 ; (\text{max } 7 (\text{min } 2 (\text{max } 4 -\text{inf.0})))
\]

\[
> (\text{my-foldr} \text{ cons } '(8) '(7 2 4))
\]

\[
\Rightarrow '(7 2 4 8) ; (\text{cons } 7 (\text{cons } 2 (\text{cons } 4 '(8))))
\]

\[
> (\text{my-foldr} \text{ append } \text{null} '((2 3) (4) (5 6 7))) \Rightarrow '(2 3 4 5 6 7)
\]

\[
(\text{append } '(2 3) (\text{append } '(4) (\text{append } '(5 6 7) '())))
\]

\[
(\text{define } (\text{my-length } \text{L})
\]

\[
(\text{my-foldr } (\lambda (\text{fst sublen}) (+ 1 \text{sublen})) 0 \text{ L}) ; \text{fill in the blank}
\]

\[
(\text{define } (\text{filter-positive } \text{num})
\]

\[
(\text{my-foldr } (\lambda (\text{num subPoss})
\]

\[
(\text{if } (> \text{num } 0) (\text{cons num subPoss} \text{ subPoss}))
\]

\[
() \text{ num}) ; \text{fill in the blank}
\]

More \textit{my-foldr} Examples Solutions

\[
> (\text{my-foldr} (\lambda (\text{fst subBool}) (\text{and fst subBool})) \#t
\]

\[
(\text{list } \#t \#t \#t))
\]

\[
\#t ; (\text{and } \#t (\text{and } \#t (\text{and } \#t \#t)))
\]

\[
> (\text{my-foldr} (\lambda (\text{fst subBool}) (\text{and fst subBool})) \#f
\]

\[
(\text{list } \#f \#f \#t))
\]

\[
\#f ; (\text{and } \#f (\text{and } \#f (\text{and } \#t \#t)))
\]

\[
> (\text{my-foldr} (\lambda (\text{fst subBool}) (\text{or fst subBool})) \#f
\]

\[
(\text{list } \#f \#f \#f))
\]

\[
\#f ; (\text{or } \#f (\text{or } \#f (\text{or } \#f \#f)))
\]

;; This doesn’t work. Why not?

\[
> (\text{my-foldr and } \text{null} (\text{list } \#t \#t \#t))
\]

Because and is a syntactic sugar keyword, not a first-class function
Define `sumProdList` (from scope lecture) in terms of `foldr`.

Is `let` necessary here like it was in scoping lecture?

```scheme
(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

#### Problems for `foldr` Solutions

`foldr` 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 to keep first element.

```
(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.

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

```
(\( \text{nums} \) ->
  (second
   (foldr
    (\( \text{thisNum} \ \text{nextNum}& \text{subResult} \) ->
      let {
        \text{nextNum} = (first nextNum)& 
        \text{subResult} = (second nextNum)& 
      }
      (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)))
```

keepBiggerThanNext with foldr-ternop

Solutions

```
(define (keepBiggerThanNext nums)
  (foldr-ternop
   (\( \text{thisNum} \ \text{restNums} \ \text{subResult} \) ->
     
     (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)
```

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))))
```

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