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

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

\[
\begin{align*}
(\text{F } (F v_1)) \\
(\text{F } (F v_2)) \\
(\text{F } (F v_n)) \\
\end{align*}
\]

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

Express mapping via higher-order my-map

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

> (my-map (λ (x) (* 2 x)) '(7 2 4))

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

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

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

**Higher-order Liss Funs**

**map-scale**

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)

**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) '(1 2 3))

> (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)

> (my-map (                        8) LOL) ; fill in the blank

**Higher-order Liss Funs**

**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))

'(8 7 6)
**Built-in Racket map Function**
Maps over Any Number of Lists

```scheme
> (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)
```

**Higher-order Lists Fun**

**Recall the List Filtering Pattern**

\[
\text{filterP (list } v1 \ v2 \ \cdots \ vn)\]

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

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

```scheme
(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**

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

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

\[
\begin{array}{c}
\text{combine} \\
\text{combine} \\
\text{combine} \\
\text{nullval}
\end{array}
\]

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

Express Recursive List Accumulation via Higher-order my-foldr

\[
\begin{array}{c}
\text{combine} \\
\text{combine} \\
\text{combine} \\
\text{nullval}
\end{array}
\]

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

my-foldr Examples

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

More my-foldr Examples

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

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

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

\begin{verbatim}
(define (sumProdList '(5 2 4 3)) -> (14 . 120)
(define (sumProdList '()) -> (0 . 1))
(define (sumProdList nums)
  (foldr ; combiner
    ; nullval
    nums))
\end{verbatim}

\textit{Higher-order Liss Funs} 17

Mapping & Filtering in terms of \textit{my-foldr}

\begin{verbatim}
(define (my-map f xs)
  (my-foldr ; combiner
    ; nullval
    xs))

(define (my-filter pred xs)
  (my-foldr ; combiner
    ; nullval
    xs))
\end{verbatim}

\textit{Higher-order Liss Funs} 18

\begin{center}
\textbf{Built-in Racket foldr Function}

\textbf{Folds over Any Number of Lists}
\end{center}

\begin{verbatim}
> (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)
\end{verbatim}

\textit{Higher-order Liss Funs} 19

\begin{center}
\textbf{Problematic for foldr}

\begin{verbatim}
(keepBiggerThanNext nums) returns a new list that keeps all nums that are bigger than the following num. It never keeps the last num.
\end{verbatim}

\begin{verbatim}
> (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?
\end{verbatim}

\begin{verbatim}
(define (keepBiggerThanNext nums)
  (foldr <combiner> <nullvalue> nums))
\end{verbatim}

\textit{Higher-order Liss 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))))
      [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**

It's your turn.

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

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

Higher-order List Funs 21