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

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

Express Mapping via Higher-order `my-map`

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

map-scale

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)

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 ( (((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)) ; 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))

Haskell Curry
Built-in Racket \texttt{map} Function
Maps over Any Number of Lists

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

Recall the List Filtering Pattern
\begin{verbatim}
(filterP (list v1 v2 ... vn))
\end{verbatim}

Express Filtering via Higher-order \texttt{my-filter}

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

Built-in Racket \texttt{filter} function acts just like \texttt{my-filter}

\begin{verbatim}
> (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))
\end{verbatim}
Recall the Recursive List Accumulation Pattern

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

\[
\begin{array}{cccccc}
  v_1 & \rightarrow & v_2 & \rightarrow & \cdots & \rightarrow \bullet \\
  \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
  \text{combine} & \downarrow & \text{combine} & \downarrow & \cdots & \downarrow \text{nullval}
\end{array}
\]

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

\[
\begin{array}{l}
\text{if } (\text{null? } xs) \\
\quad \text{nullval} \\
\quad (\text{combine } (\text{first } xs) \\
\quad \text{(rec-accum } \text{(rest } xs)\text{))})
\end{array}
\]

Express Recursive List Accumulation via Higher-order my-foldr

\[
\begin{array}{cccccc}
  v_1 & \rightarrow & v_2 & \rightarrow & \cdots & \rightarrow \bullet \\
  \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
  \text{combine} & \downarrow & \text{combine} & \downarrow & \cdots & \downarrow \text{nullval}
\end{array}
\]

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

\[
\begin{array}{l}
\text{if } (\text{null? } \text{vals}) \\
\quad \text{nullval} \\
\quad (\text{combine } (\text{first } \text{vals}) \\
\quad \text{(my-foldr } \text{combine nullval} \\
\quad \text{(rest } \text{vals}))\text{)})
\end{array}
\]

my-foldr Examples

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

More my-foldr Examples

> (my-foldr (\(a \ b) \ (and \ a \ b)) \#t \ (list \#t \#t \#t))
> (my-foldr (\(a \ b) \ (and \ a \ b)) \#t \ (list \#t \#f \#t))
> (my-foldr (\(a \ b) \ (or \ a \ b)) \#f \ (list \#t \#f \#t))
> (my-foldr (\(a \ b) \ (or \ a \ b)) \#f \ (list \#f \#f \#f))

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

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

\[
\begin{align*}
\text{(sumProdList '(5 2 4 3)) &\rightarrow (14 . 120) \\
\text{(sumProdList '()) &\rightarrow (0 . 1)}
\end{align*}
\]

(\text{define (sumProdList nums)})

<table>
<thead>
<tr>
<th>foldr</th>
<th>; combiner</th>
</tr>
</thead>
<tbody>
<tr>
<td>nums</td>
<td>; nullval</td>
</tr>
</tbody>
</table>

\textit{Higher-order Liss Funs 17}

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

\[
\begin{align*}
\text{(define (my-map f xs)} &\quad ; \text{combiner} \\
\text{(my-foldr)} &\quad ; \text{nullval} \\
\text{xs}) &\text{)}
\end{align*}
\]

\[
\begin{align*}
\text{(define (my-filter pred xs)} &\quad ; \text{combiner} \\
\text{(my-foldr)} &\quad ; \text{nullval} \\
\text{xs}) &\text{)}
\end{align*}
\]

\textit{Higher-order Liss Funs 18}

Built-in Racket \textit{foldr} Function

Folds over Any Number of Lists

\[
\begin{align*}
> (\text{foldr} + 0 '(7 2 4)) &\rightarrow 13 \\
> (\text{foldr} \lambda (a b sum) (+ (* a b) sum)) &\rightarrow 0 \\
&\quad '(2 3 4) \\
&\quad '(5 6 7)) &\rightarrow 56 \\
> (\text{foldr} \lambda (a b sum) (+ (* a b) sum)) &\rightarrow 0 \\
&\quad '(1 2 3 4) \\
&\quad '(5 6 7)) &\text{ERROR: foldr: given list does not have the same size as the first list: '(5 6 7)}
\end{align*}
\]

\textit{Higher-order Liss Funs 19}

Problematic for \textit{foldr}

\[
\begin{align*}
> (\text{keepBiggerThanNext } '(7 5 3 9 8)) &\rightarrow '(7 5 9) \\
> (\text{keepBiggerThanNext } '(2 7 5 3 9 8)) &\rightarrow '(7 5 9) \\
> (\text{keepBiggerThanNext } '(4 2 7 5 3 9 8)) &\rightarrow '(4 7 5 9)
\end{align*}
\]

\text{keepBiggerThanNext cannot be defined by fleshing out the following template. Why not?}

\[
\begin{align*}
> (\text{define (keepBiggerThanNext nums)} &\rightarrow (\text{foldr} <\text{combiner}> <\text{nullvalue}> nums))
\end{align*}
\]

\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)])
                 (if (not nextNum) ; At last num, no next one
                   (list thisNum ; thisNum is nextNum one back
                       (if (> thisNum nextNum)
                       (cons thisNum subResult)
                       subResult))))))
  (list #f '()))
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))
```

```
(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)
```