Higher-order List Functions in Racket

CS251 Programming Languages
Fall 2017, 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`.

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

Recall the List Mapping Pattern

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

\[
\begin{array}{c}
\text{v}_1 \\
\downarrow F \\
(F \ v_1) \\
\downarrow \text{F} \\
\text{v}_1 \\
\downarrow \text{F} \\
(F \ v_1) \\
\downarrow \text{F} \\
 \text{v}_n \\
\end{array}
\]

\[
\begin{array}{c}
\text{map}\ F \ \text{xs} \\
\text{if} \ (\text{null}\? \ \text{xs}) \\
\text{null} \\
\text{cons} \ (F \ (\text{first} \ \text{xs})) \\
\text{map}\ F \ (\text{rest} \ \text{xs}))
\end{array}
\]

Express Mapping via Higher-order **my-map**

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

> (my-map (λ (x) (* 2 x)) (list 7 2 4))

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

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

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

Your turn

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

> (map-scale 3 (list 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) (list 1 2 3))

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

Mapping with binary functions

(define (my-map2 binop xs ys)
  (if (not (= (length xs) (length ys)))
    (error "my-map2 requires same-length lists")
    (if (or (null? xs) (null? ys))
      null
      (cons (binop (first xs) (first ys))
            (my-map2 binop (rest xs) (rest ys))))))

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

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

ERROR: my-map2 requires same-length lists

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

'((2 3 4 5) (6 4 2))

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

ERROR: my-map2 requires same-length lists

Haskell Curry

define pow n = (flip pow n)

> (define (my-map2 pow n nums)
    (map (flip pow n) nums))

> (my-map2 pow n (list 7 2 4))

> (my-map2 pow n (range 0 5))

'((0 1) (1 2) (2 4) (3 8) (4 16) (5 32) (6 64) (7 128) (8 256))

> (my-map2 pow n (range 0 5))

'((0 1) (1 2) (2 4) (3 8) (4 16) (5 32) (6 64) (7 128) (8 256))

> (define pow n = (flip pow n))

ERROR: bad expression, possibly bad syntax

Haskell Curry
**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 (list 2 3 5) (list 6 4 2))` → `'(64 81 25)
- `(map (λ (a b x) (+ (* a x) b)) (list 2 3 5) (list 6 4 2) (list 0 1 2))` → `'(6 7 12)
- `(map pow (list 2 3 4 5) (list 6 4 2))` → `ERROR: map: all lists must have same size; arguments were: #<procedure:pow> '(2 3 4 5) '(6 4 2)`

**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`
Recall the Recursive List Accumulation Pattern

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

\[
\begin{array}{ccc}
v_1 & \rightarrow & v_2 \\
\uparrow & \downarrow & \downarrow \\
\text{combine} & \rightarrow & \text{combine} \\
& \downarrow & \downarrow \\
& \text{...} & \rightarrow \text{combine} \\
& \downarrow & \downarrow \\
& \rightarrow & \text{nullval} \\
\end{array}
\]

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

Express Recursive List Accumulation via Higher-order \text{my-foldr}

\[
\begin{array}{ccc}
v_1 & \rightarrow & v_2 \\
\uparrow & \downarrow & \downarrow \\
\text{nullval} & \rightarrow & \text{combine} \\
& \downarrow & \downarrow \\
& \text{...} & \rightarrow \text{combine} \\
& \downarrow & \downarrow \\
& \rightarrow & \text{nullval} \\
\end{array}
\]

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

my-foldr Examples

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

More my-foldr Examples

\[
> (\text{my-foldr } (\lambda (a b) (\text{and } a b)) \#t (\text{list } \#t \#t \#t))
\]
\[
> (\text{my-foldr } (\lambda (a b) (\text{and } a b)) \#t (\text{list } \#t \#f \#t))
\]
\[
> (\text{my-foldr } (\lambda (a b) (\text{or } a b)) \#f (\text{list } \#t \#f \#t))
\]
\[
> (\text{my-foldr } (\lambda (a b) (\text{or } a b)) \#f (\text{list } \#f \#f \#f))
\]
\[
;; \text{This doesn’t work. Why not?}
> (\text{my-foldr and } \#t (\text{list } \#t \#t \#t))
\]
Mapping & Filtering in terms of `my-foldr`

```scheme
(define (my-map f xs)
  (my-foldr ???
    ???
    xs))

(define (my-filter pred xs)
  (my-foldr ???
    ???
    xs))
```

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

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

More `foldr` Examples

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

Problematic for `foldr`

```scheme
(locallyBig nums) returns a new list that keeps all nums that are bigger than the following num. It always keeps the last num.

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

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

```scheme
(define (locallyBig nums)
  (foldr <combiner> <nullvalue> nums))
```
locallyBig with foldr

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

(define (locallyBig nums)
  (second
    (foldr
      (λ (thisNum nextNum locallyBigRest)
        (let ((nextNum (first locallyBigRest))
          (locallyBigRest
            (second nextNum locallyBigRest)))
          (list thisNum ; #1: nextNum for elt to left
              nextNum ; #2: list from below
              (if (> thisNum nextNum)
                (cons thisNum locallyBigRest)
                locallyBigRest)))))
    (list -inf.0 ; #1 initial nextNum
      '()) ; #1 initial list
  nums)))

foldr-ternop: more info for combiner

In cases like locallyBig, helps for combiner to also take rest of list.

(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)))))

locallyBig with foldr-ternop

(define (locallyBigTernop nums)
  (foldr-ternop
    (λ (thisNum restNums locallyBigRest)
      (if (null? restNums)
        (list thisNum) ; Always include last num in nums
        (let ((nextNum (first restNums))) ; Key info from
          ; extra arg
          (if (> thisNum nextNum)
            (cons thisNum locallyBigRest)
            locallyBigRest))))
    '()
    nums))

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