Exercises

-> (map ((curry +) 3) '(1 2 3 4 5))

-> (exists? ((curry =) 3) '(1 2 3 4 5))

-> (filter ((curry >) 3) '(1 2 3 4 5))

; tricky
Answers

\[
\rightarrow \quad (\text{map} \quad ((\text{curry} +) \ 3) \quad '(1 \ 2 \ 3 \ 4 \ 5)) \\
(4 \ 5 \ 6 \ 7 \ 8)
\]

\[
\rightarrow \quad (\exists \quad ((\text{curry} =) \ 3) \quad '(1 \ 2 \ 3 \ 4 \ 5)) \\
\text{#t}
\]

\[
\rightarrow \quad (\text{filter} \quad ((\text{curry} >) \ 3) \quad '(1 \ 2 \ 3 \ 4 \ 5)) \\
(1 \ 2)
\]
Defining exists?

-> (define exists? (p? xs)
    (if (null? xs)
        #f
        (or (p? (car xs))
            (exists? p? (cdr xs))))))

-> (exists? even? '(1 3))
  #f

-> (exists? even? '(1 2 3))
  #t

-> (exists? ((curry =) 0) '(1 2 3))
  #f

-> (exists? ((curry =) 0) '(0 1 2 3))
  #t
all?
Defining all?

-> (define all? (p? xs)
     (if (null? xs)
         #t
         (and (p? (car xs))
             (all? p? (cdr xs))))))

-> (all? even? '(1 3))  #f
-> (all? even? '(2))    #t
-> (all? ((curry =) 0) '(1 2 3))  #f
-> (all? ((curry =) 0) '(0 0 0))  #t
Filter
Defining filter

-> (define filter (p? xs)
   (if (null? xs)
       ()
       (if (p? (car xs))
           (cons (car xs) (filter p? (cdr xs)))
           (filter p? (cdr xs)))))

-> (filter (lambda (n) (> n 0)) '(1 2 -3 -4 5 6))
(1 2 5 6)

-> (filter (lambda (n) (<= n 0)) '(1 2 -3 -4 5 6))
(-3 -4)

-> (filter ((curry <) 0) '(1 2 -3 -4 5 6))
(1 2 5 6)

-> (filter ((curry >=) 0) '(1 2 -3 -4 5 6))
(-3 -4)
Composition Revisited: List Filtering

-> (val positive? ((curry <) 0))
<procedure>
-> (filter positive? '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter (o not positive?) '(1 2 -3 -4 5 6))
(-3 -4)
Map
Defining map

-> (define map (f xs)
   (if (null? xs)
       ()
       (cons (f (car xs)) (map f (cdr xs))))))

-> (map number? '(3 a b (5 6)))
(#t #f #f #f)

-> (map ((curry *) 100) '(5 6 7))
(500 600 700)

-> (val square* ((curry map) (lambda (n) (* n n))))
<procedure>

-> (square* '(1 2 3 4 5))
(1 4 9 16 25)
Foldr
Algebraic laws for foldr

Idea: $\lambda+. \lambda^0.x_1+\cdots+x_n+0$

$$(\text{foldr (plus zero ')()}) = \text{zero}$$
$$(\text{foldr (plus zero (cons y ys)))} =$$

$$(\text{plus y (foldr plus zero ys)}$$)

Note: Binary operator + associates to the right.

Note: zero should be identity of plus.
Code for foldr

Idea: $\lambda+.\lambda0.x_1 + \cdots + x_n + 0$

-> (define foldr (plus zero xs)
   (if (null? xs)
       zero
       (plus (car xs) (foldr plus zero (cdr xs)))))

-> (val sum (lambda (xs) (foldr + 0 xs)))

-> (sum '(1 2 3 4))
  10

-> (val prod (lambda (xs) (foldr * 1 xs)))

-> (prod '(1 2 3 4))
  24
Another view of operator folding

\[
(1 \ 2 \ 3 \ 4) = \text{(cons 1 (cons 2 (cons 3 (cons 4 '()))))}
\]
\[
(\text{foldr } + \ 0 \ (1 \ 2 \ 3 \ 4))
\]
\[
= (+ \ 1 (+ \ 2 (+ \ 3 (+ \ 4 0 ))))
\]
\[
(\text{foldr } f \ z \ (1 \ 2 \ 3 \ 4))
\]
\[
= (f \ 1 (f \ 2 (f \ 3 (f \ 4 z ))))
\]
Exercise

Idea: $\lambda+. \lambda 0. x_1 + \cdots + x_n + 0$

$\Rightarrow \ (\text{define} \ \text{combine} \ (x \ a) \ (+ \ 1 \ a))$

$\Rightarrow \ (\text{foldr} \ \text{combine} \ 0 \ ' (2 \ 3 \ 4 \ 1))$

???
Answer

Idea: \( \lambda + . \lambda 0. x_1 + \cdots + x_n + 0 \)

\[
\Rightarrow \quad \text{(define combine (x a) (+ 1 a))}
\]

\[
\Rightarrow \quad \text{(foldr combine 0 '(2 3 4 1))}
\]

4
What is tail position?

Tail position is defined inductively:

• The body of a function is in tail position
• When `(if e1 e2 e3)` is in tail position, so are `e2` and `e3`
• When `(let (...) e)` is in tail position, so is `e`, and similarly for `letrec` and `let*`.
• When `(begin e1 ... en)` is in tail position, so is `en`.

Idea: The last thing that happens
Tail-call optimization

Before executing a call in tail position, abandon your stack frame

Results in asymptotic space savings

Works for any call!
Example of tail position

(define reverse (xs)
  (if (null? xs) '()
      (append (reverse (cdr xs))
              (list1 (car xs)))))
Example of tail position

(define reverse (xs)
  (if (null? xs) '()
      (append (reverse (cdr xs))
              (list1 (car xs))))))
Another example of tail position

(define revapp (xs zs)
  (if (null? xs) zs
      (revapp (cdr xs) (cons (car xs) zs)))))
Another example of tail position

(define revapp (xs zs)
  (if (null? xs) zs
      (revapp (cdr xs) (cons (car xs) zs)))))
Question

In your previous life, what did you call a construct that

1. Transfers control to an arbitrary point in the code?

2. Uses no stack space?