Conjunction, disjunction of predicates

(define conjoin (p? q?)
  (lambda (x) (if (p? x) (q? x) #f)))

(define disjoin (p? q?)
  (lambda (x) (if (p? x) #t (q? x))))
Functions create new functions

-> (define o (f g) (lambda (x) (f (g x))))
-> (define even? (n) (= 0 (mod n 2)))
-> (val odd? (o not even?))
-> (odd? 3)
#t
-> (odd? 4)
#f
Higher-order list functions

-> (all? even? '(1 2 3 4))
#f

-> (exists? even? '(1 2 3 4))
#t

-> (map (lambda (n)
          (if (even? n) n 'odd))
      '(1 2 3 4))
(odd 2 odd 4)

-> (filter even? '(1 2 3 4))
(2 4)
To get one-argument functions: Curry

-> (val positive? (lambda (y) (< 0 y)))
-> (positive? 3)
#t
-> (val <-c (lambda (x) (lambda (y) (< x y))))
-> (val positive? (<-c 0)) ; "partial application"
-> (positive? 0)
#f
What’s the algebraic law for \texttt{curry}?

\[ \ldots \ (\text{curry } f) \ldots = \ldots f \ldots \]

Keep in mind:
All you can do with a function is apply it!

\[ (((\text{curry } f) \ x) \ y) = (f \ x \ y) \]
No need to Curry by hand!

;; curry : binary function -> value -> function
-> (val curry
   (lambda (f)
     (lambda (x)
       (lambda (y) (f x y))))
-> (val positive? ((curry <) 0))
-> (positive? -3)
#f
-> (positive? 11)
#t
Your turn!

-> (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

-> (map ((curry +) 3) '(1 2 3 4 5))
   (4 5 6 7 8)
-> (exists? ((curry =) 3) '(1 2 3 4 5))
   #t
-> (filter ((curry >) 3) '(1 2 3 4 5))
   (1 2)
Defining exists?

-> (define exists? (p? xs)
   (if (null? xs)
       #f
       (if (p? (car xs))
           #t
           (exists? p? (cdr xs))))))
-> (exists? pair? '(1 2 3))
   #f
-> (exists? pair? '(1 2 (3)))
   #t
-> (exists? ((curry =) 0) '(1 2 3))
   #f
-> (exists? ((curry =) 0) '(0 1 2 3))
   #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:** \[ \lambda + \lambda^0 . x_1 + \cdots + x_n + 0 \]

\[
(foldr \ (\text{plus} \ \text{zero} \ ' ())) \quad = \quad \text{zero}
\]
\[
(foldr \ (\text{plus} \ \text{zero} \ (\text{cons} \ y \ ys))) \quad =
\quad (\text{plus} \ y \ (foldr \ \text{plus} \ \text{zero} \ ys))
\]

**Note:** Binary operator + associates to the right.

**Note:** \text{zero} should be identity of plus.
Code for foldr

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

$\rightarrow$ (define foldr (plus zero xs)
  (if (null? xs)
    zero
    (plus (car xs) (foldr plus zero (cdr xs)))))

$\rightarrow$ (val sum (lambda (xs) (foldr + 0 xs)))

$\rightarrow$ (sum ' (1 2 3 4))

10

$\rightarrow$ (val prod (lambda (xs) (foldr * 1 xs)))

$\rightarrow$ (prod ' (1 2 3 4))

24
Another view of operator folding

\[(1 \ 2 \ 3 \ 4) = (\text{cons} \ 1 \ (\text{cons} \ 2 \ (\text{cons} \ 3 \ (\text{cons} \ 4 \ '()'))))\]

(\text{foldr} + 0 \ '(1 \ 2 \ 3 \ 4))

\[= (+ \ 1 \ (+ \ 2 \ (+ \ 3 \ (+ \ 4 \ 0 ))))\]

(\text{foldr} f \ z \ '(1 \ 2 \ 3 \ 4))

\[= (f \ 1 \ (f \ 2 \ (f \ 3 \ (f \ 4 \ z ))))\]
Your turn

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

$\rightarrow (\text{define combine} \ (x\ a) \ (+\ 1\ a))$
$\rightarrow (\text{foldr combine} \ 0 \ '(2\ 3\ 4\ 1))$

???
Wait for it
Answer

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

$\rightarrow$ (define combine (x a) (+ 1 a))
$\rightarrow$ (foldr combine 0 '(2 3 4 1))
4

What functionality have we just duplicated?
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 past, what did you call a construct that

1. Transfers control to a point in the code?

2. Uses no stack space?