Last time

; (exists? p? '()) == #f
; (exists? p? (cons y ys)) == (p? y) or (exists p? ys)

-> (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
Your turn: `map` and `filter`

```lisp
-> (map ((curry +) 3) '(1 2 3 4 5))
(4 5 6 7 8)

;; (map f '()) =
;; (map f (cons y ys)) =

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

;; (filter p? '()) =
;; (filter p? (cons y ys)) =
```
Answers: map and filter

-> (map ((curry +) 3) '(1 2 3 4 5))
(4 5 6 7 8)

; (map f '()) == '()
; (map f (cons y ys)) == (cons (f y) (map f ys))

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

; (filter p? '()) == '()
; (filter p? '(cons y ys)) == (cons y (filter p? ys)),
when (p? y)
; (filter p? '(cons y ys)) == (filter p? ys),
not (p? y)
Foldr
Algebraic laws for foldr

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

(foldr (plus zero '())) = zero
(foldr (plus zero (cons y ys))) =
  (plus y (foldr plus zero ys))

Note: Binary operator + associates to the right.

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

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

```scheme
-> (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

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

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

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

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

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

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

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

???
Wait for it
Answer

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

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

What functionality have we just duplicated?
Your turn: Explain the design

1. Functions like `exists?`, `map`, `filter` are subsumed by
2. Function `foldr`, which is subsumed by
3. Recursive functions

Seems redundant: Why?
Next up: Costs of function calls
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?
How functions finish

Direct: return answer;
True CPS: throw k answer;
uScheme: (k answer)
Design Problem: Missing Value

Provide a witness to existence:

\[(\text{witness } p\? \ x \ \text{xs}) \equiv x, \text{ where } (\text{member } x \ \text{xs}), \]
\[\text{provided } (\text{exists? } p\? \ x \ \text{xs})\]

Problem: What if there exists no such \(x\)?
Solution: A New Interface

Success and failure continuations!

Laws:

\[
\text{Witness-cps } p? \ x : \text{xs succ fail) } = \text{(succ x)} \\
\quad \text{; where } x \text{ is in xs and } (p? \ x)
\]

\[
\text{Witness-cps } p? \ x : \text{xs succ fail) } = \text{(fail)} \\
\quad \text{; where } (\neg (\exists x \in x : \text{xs}))
\]
From contract to laws

(witness-cps p? xs succ fail) = (succ x)
  ; where x is in xs and (p? x)
(witness-cps p? xs succ fail) = (fail)
  ; where (not (exists? p? xs))

(witness-cps p? '() succ fail) = ?

(witness-cps p? (cons z zs) succ fail) = ?
  ; when (p? z)

(witness-cps p? (cons z zs) succ fail) = ?
  ; when (not (p? z))
Coding \textbf{witness} with continuations

\begin{verbatim}
(define witness-cps (p? xs succ fail)
    (if (null? xs)
        (fail)
        (let ([x (car xs)])
            (if (p? x)
                (if (p? x)
                    (succ x)
                    (witness-cps p? (cdr xs) succ fail))))))
\end{verbatim}
“Continuation-Passing Style”

All tail positions are continuations or recursive calls

(define witness-cps (p? xs succ fail)
    (if (null? xs)
        (fail)
        (let ([x (car xs)])
            (if (p? x)
                (if (p? x)
                    (succ x)
                    (witness-cps p? (cdr xs) succ fail))))))

Compiles to tight code
Example Use: Instructor Lookup

-> (val 2016f '((Fisher 105) (Hescott 170) (Chow 116)))
-> (instructor-info 'Fisher 2016f)
  (Fisher teaches 105)
-> (instructor-info 'Chow 2016f)
  (Chow teaches 116)
-> (instructor-info 'Souvaine 2016f)
  (Souvaine is-not-on-the-list)
Instructor Lookup: The Code

; info has form: ' (Fisher 105)
; classes has form: ' (info_1 ... info_n)
(define instructor-info (instructor classes)
  (let ((
      [s ; success continuation
        ... ]
      [f ; failure continuation
        ... ]
  )
    (witness-cps pred
      classes s f)))
Instructor Lookup: The Code

; info has form: '(Fisher 105)
; classes has form: '(info_1 ... info_n)
(define instructor-info (instructor classes)
  (let (
    [s ; success continuation
      ...]
    [f ; failure continuation
      ...])
    (witness-cps (o ((curry =) instructor) car)
                 classes s f)))
Instructor Lookup: The Code

; info has form: '(Fisher 105)
; classes has form: '(info_1 ... info_n)
(define instructor-info (instructor classes)
  (let (  
      [s (lambda (info) ; success continuation
          (list3 instructor 'teaches (cadr info)))]
      [f ; failure continuation]
    (witness-cps (o ((curry =) instructor) car)
      classes s f))


Instructor Lookup: The Code

; info has form: '(Fisher 105)
; classes has form: '(info_1 ... info_n)
(define instructor-info (instructor classes)
  (let (
    [s (lambda (info) ; success continuation
      (list3 instructor 'teaches (cadr info)))]
    [f (lambda () ; failure continuation
      (list2 instructor 'is-not-on-the-list))])
  (witness-cps (o ((curry =) instructor) car)
    classes s f)))