

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

# Programming Languages

Dan Grossman

Defining Streams

# Streams

Coding up a stream in your program is easy

- We will do functional streams using pairs and thunks

Let a stream be a thunk that *when called* returns a pair:

```
' (next-answer . next-thunk)
```

Saw how to use them, now how to make them...

- Admittedly mind-bending, but uses what we know

# Making streams

- How can one thunk create the right next thunk? Recursion!
  - Make a thunk that produces a pair where cdr is next thunk
  - A recursive function can return a thunk where recursive call does not happen until thunk is called

```
(define ones (lambda () (cons 1 ones)))

(define nats
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (+ x 1))))))]
    (lambda () (f 1))))

(define powers-of-two
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (* x 2))))))]
    (lambda () (f 2))))
```

# Getting it wrong

- This uses a variable before it is defined

```
(define ones-really-bad (cons 1 ones-really-bad))
```

- This goes into an infinite loop making an infinite-length list

```
(define ones-bad (lambda () cons 1 (ones-bad)))  
(define (ones-bad) (cons 1 (ones-bad)))
```

- This is a stream: thunk that returns a pair with cdr a thunk

```
(define ones (lambda () (cons 1 ones)))  
(define (ones) (cons 1 ones))
```