

```
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

Procs

Blocks are “second-class”

All a method can do with a block is **yield** to it

- Cannot return it, store it in an object (e.g., for a callback), ...
- But can also turn blocks into real closures
- Closures are instances of class **Proc**
 - Called with method **call**

This is Ruby, so there are several ways to make **Proc** objects ☺

- One way: method **lambda** of **Object** takes a block and returns the corresponding **Proc**

Example

```
a = [3,5,7,9]
```

- Blocks are fine for applying to array elements

```
b = a.map {|x| x+1 }  
i = b.count {|x| x>=6 }
```

- But for an array of closures, need **Proc** objects
 - More common use is callbacks

```
c = a.map {|x| lambda {|y| x>=y}}  
c[2].call 17  
j = c.count {|x| x.call(5) }
```

Moral

- First-class (“can be passed/stored anywhere”) makes closures more powerful than blocks
- But blocks are (a little) more convenient and cover most uses
- This helps us understand what first-class means
- Language design question: When is convenience worth making something less general and powerful?