

```
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
2013

Nested Functions

Any binding

According to our rules for let-expressions, we can define functions inside any let-expression

```
let  b1 b2 ... bn  in  e  end
```

This is a natural idea, and often good style

(Inferior) Example

```
fun countup_from1 (x : int) =  
  let fun count (from : int, to : int) =  
  
        if from = to  
        then to :: []  
        else from :: count(from+1,to)  
      in  
        count (1,x)  
      end
```

- This shows how to use a local function binding, but:
 - Better version on next slide
 - **count** might be useful elsewhere

Better:

```
fun countup_from1_better (x : int) =  
  let fun count (from : int) =  
        if from = x  
        then x :: []  
        else from :: count(from+1)  
      in  
        count 1  
      end
```

- Functions can use bindings in the environment where they are defined:
 - Bindings from “outer” environments
 - Such as parameters to the outer function
 - Earlier bindings in the let-expression
- Unnecessary parameters are usually bad style
 - Like `to` in previous example

Nested functions: style

- Good style to define helper functions inside the functions they help if they are:
 - Unlikely to be useful elsewhere
 - Likely to be misused if available elsewhere
 - Likely to be changed or removed later
- A fundamental trade-off in code design: reusing code saves effort and avoids bugs, but makes the reused code harder to change later