```
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

Polymorphic Types and Functions As Arguments

The key point

- Higher-order functions are often so “generic” and “reusable” that they have polymorphic types, i.e., types with type variables

- But there are higher-order functions that are not polymorphic

- And there are non-higher-order (first-order) functions that are polymorphic

- Always a good idea to understand the type of a function, especially a higher-order function

Types

```
fun n_times (f,n,x) =
    if n=0
    then x
    else f (n_times(f,n-1,x))
```

- **`val n_times : ('a -> 'a) * int * 'a -> 'a`**
  - Simpler but less useful: **`(int -> int) * int * int -> int`**

- Two of our examples *instantiated* **`'a`** with **`int`**
- One of our examples *instantiated* **`'a`** with **`int list`**
- This *polymorphism* makes **`n_times`** more useful

- Type is *inferred* based on how arguments are used (later lecture)
  - Describes which types must be exactly something (e.g., **`int`**) and which can be anything but the same (e.g., **`'a`**)

Polymorphism and higher-order functions

- Many higher-order functions are polymorphic because they are so reusable that some types, "can be anything"

- But some polymorphic functions are not higher-order
  - Example: **`len : 'a list -> int`**

- And some higher-order functions are not polymorphic
  - Example: **`times_until_0 : (int -> int) * int -> int`**

```
fun times_until_0 (f,x) =
   if x=0 then 0 else 1 + times_until_0(f, f x)
```

Note: Would be better with tail-recursion