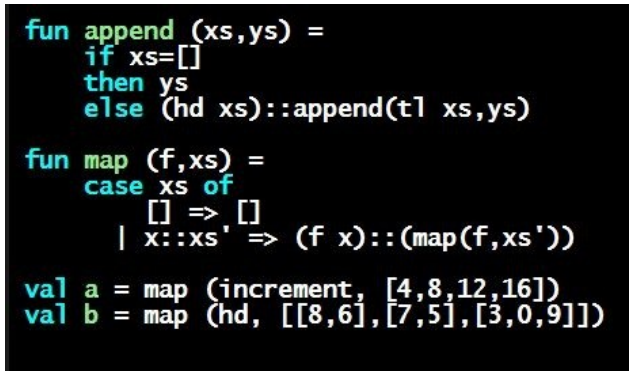

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

Accumulators

Moral of tail recursion

- Where reasonably elegant, feasible, and important, rewriting functions to be *tail-recursive* can be much more efficient
  - Tail-recursive: recursive calls are tail-calls

- There is a methodology that can often guide this transformation:
  - Create a helper function that takes an *accumulator*
  - Old base case becomes initial accumulator
  - New base case becomes final accumulator

Methodology already seen

```
fun fact n =
    let fun aux(n,acc) =
            if n=0
            then acc
            else aux(n-1,acc*n)
    in
        aux(n,1)
    end

val x = fact 3
```

fact 3	aux(3,1)	aux(2,3)	aux(1,6)	aux(0,6)

Another example

```
fun sum xs =
    case xs of
        [] => 0
         | x::xs' => x + sum xs'
```

```
fun sum xs =
    let fun aux(xs,acc) =
            case xs of
               [] => acc
             | x::xs' => aux(xs',x+acc)
    in
       aux(xs,0)
    end
```

And another

```
fun rev xs =
    case xs of
        [] => []
         | x::xs' => (rev xs') @ [x]
```

```
fun rev xs =
    let fun aux(xs,acc) =
            case xs of
               [] => acc
             | x::xs' => aux(xs',x::acc)
    in
       aux(xs,[])
    end
```

Actually much better

```
fun rev xs =
    case xs of
         [] => []
          | x::xs' => (rev xs') @
    [x]
```

- For **`fact`** and **`sum`**, tail-recursion is faster but both ways linear time
- Non-tail recursive **`rev`** is quadratic because each recursive call uses append, which must traverse the first list
  - And 1+2+…+(length-1) is almost length*length/2
  - Moral: beware list-append, especially within outer recursion
- Cons constant-time (and fast), so accumulator version much better