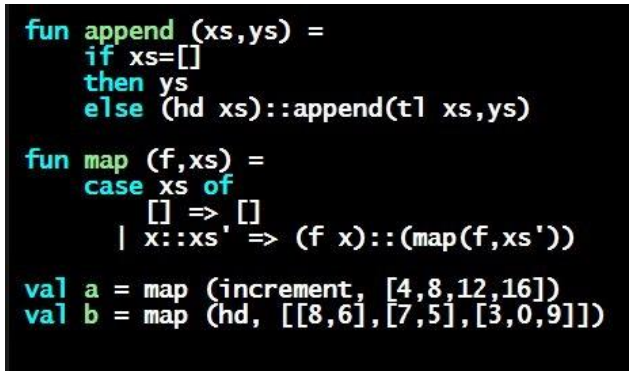

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

Modules for Namespace Management

Modules

For larger programs, one “top-level” sequence of bindings is poor

- Especially because a binding can use *all* earlier (non-shadowed) bindings

So ML has *structures* to define *modules*

```
structure MyModule = struct bindings end
```

Inside a module, can use earlier bindings as usual

- Can have any kind of binding (val, datatype, exception, ...)

Outside a module, refer to earlier modules’ bindings via **`ModuleName.bindingName`**

- Just like **`List.foldl`** and **`String.toUpper`**; now you can define your own modules

Example

```
structure MyMathLib =
struct

fun fact x =
    if x=0
    then 1
    else x * fact(x-1)

val half_pi = Math.pi / 2

fun doubler x = x * 2

end
```

Namespace management

- *So far*, this is just *namespace management*
  - Giving a hierarchy to names to avoid shadowing
  - Allows different modules to reuse names, e.g., **`map`**
  - Very important, but not very interesting

Optional: Open

- Can use **`open ModuleName`** to get "direct" access to a module's bindings
  - Never necessary; just a convenience; often bad style
  - Often better to create local val-bindings for just the bindings you use a lot, e.g., **`val map = List.map`**
    - But doesn't work for patterns
    - And **`open`** can be useful, e.g., for testing code