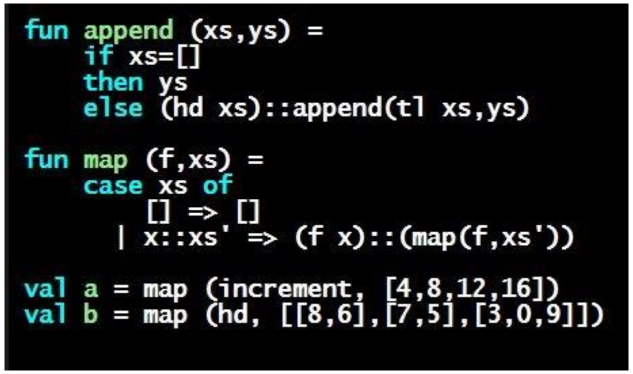

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

Introduction to First-Class Functions

What is functional programming?

"*Functional programming*" can mean a few different things:

1. Avoiding mutation in most/all cases (done and ongoing)

2. Using functions as values (this section)

…

- Style encouraging recursion and recursive data structures
- Style closer to mathematical definitions
- Programming idioms using *laziness* (later topic, briefly)
- Anything not OOP or C? (not a good definition)

Not sure a definition of "*functional language*" exists beyond "makes functional programming easy / the default / required"

  – No clear yes/no for a particular language

First-class functions

- *First-class functions*: Can use them *wherever* we use values
  - Functions are values too
  - Arguments, results, parts of tuples, bound to variables, carried by datatype constructors or exceptions, …

```
fun double x = 2*x
fun incr x = x+1
val a_tuple = (double, incr, double(incr 7))
```

- Most common use is as an argument / result of another function
  - Other function is called a *higher-order function*
  - Powerful way to *factor out* common functionality

Function Closures

- *Function closure*: Functions can use bindings from outside the function definition (in scope where function is defined)
  - Makes first-class functions *much* more powerful
  - Will get to this feature in a bit, after simpler examples

- Distinction between terms *first-class functions* and *function closures* is not universally understood
  - Important conceptual distinction even if terms get muddled

Onward

Most of this section of course:

- How to use first-class functions and closures
- The precise semantics
- Multiple powerful idioms