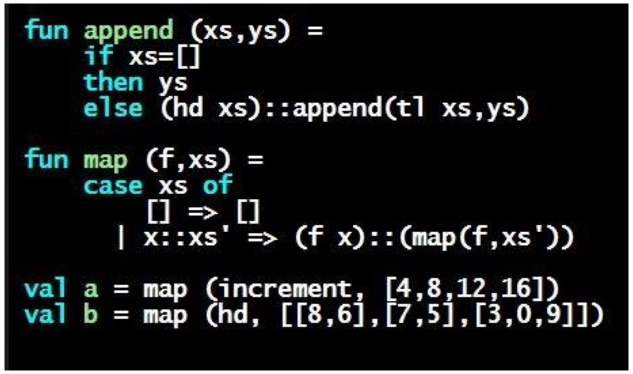

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

Equivalent Functions

Last Topic of Section

More careful look at what "two pieces of code are equivalent" means

- Fundamental software-engineering idea
- Made easier with
  - Abstraction (hiding things)
  - Fewer side effects

Not about any "new ways to code something up"

Equivalence

Must reason about “are these equivalent” *all the time*

- The more precisely you think about it the better

- *Code maintenance:* Can I simplify this code?
- *Backward compatibility:* Can I add new features without changing how any old features work?
- *Optimization:* Can I make this code faster?
- *Abstraction:* Can an external client tell I made this change?

To focus discussion: When can we say two functions are equivalent, even without looking at all calls to them?

- May not know all the calls (e.g., we are editing a library)

A definition

Two functions are equivalent if they have the same "observable behavior" no matter how they are used anywhere in any program

Given equivalent arguments, they:

- Produce equivalent results
- Have the same (non-)termination behavior
- Mutate (non-local) memory in the same way
- Do the same input/output
- Raise the same exceptions

Notice it is much easier to be equivalent if:

- There are fewer possible arguments, e.g., with a type system and abstraction
- We avoid *side-effects*: mutation, input/output, and exceptions

Example

Since looking up variables in ML has no side effects, these two functions are equivalent:

```
fun f x = x + x
```

$=$

```
val y = 2
fun f x = y * x
```

But these next two are not equivalent in general: it depends on what is passed for **`f`**

– Are equivalent *if* argument for **`f`** has no side-effects

```
fun g (f,x) =
    (f x) + (f x)
```

$\neq$

```
val y = 2
fun g (f,x) =
   y * (f x)
```

– Example: **`g (fn i => (print "hi" ; i), 7)`**

– Great reason for "pure" functional programming

Another example

These are equivalent *only if* functions bound to **`g`** and **`h`** do not raise exceptions or have side effects (printing, updating state, etc.)

- Again: pure functions make more things equivalent

```
fun f x =
    let
      val y = g x
      val z = h x
    in
      (y,z)
    end
```

$\neq$

```
fun f x =
    let
      val z = h x
      val y = g x
    in
      (y,z)
    end
```

- Example: **`g`** divides by **`0`** and **`h`** mutates a top-level reference
- Example: **`g`** writes to a reference that **`h`** reads from