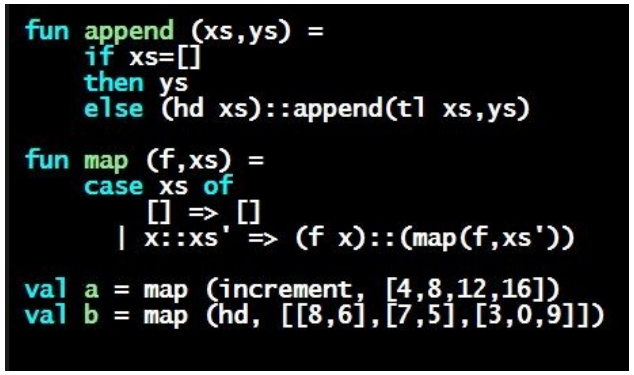

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

Polymorphic Datatypes

Finish the story

- Claimed built-in options and lists are not needed/special
  - Other than special syntax for list constructors
- But these datatype bindings are polymorphic type constructors
  - **`int list`** and **`string list`** and **`int list list`** are all types, not **`list`**
  - Functions might or might not be polymorphic
    - **`val sum_list : int list -> int`**
    - **`val append : 'a list * 'a list -> 'a list`**
- Good language design: Can define new polymorphic datatypes
- Semi-optional: Do *not* need to understand this for homework 2

Defining polymorphic datatypes

- Syntax: put one or more type variables before datatype name

```
datatype 'a option = NONE | SOME of 'a

datatype 'a mylist = Empty | Cons of 'a * 'a mylist

datatype ('a,'b) tree =
         Node of 'a * ('a,'b) tree * ('a,'b) tree
       | Leaf of 'b
```

- Can use these type variables in constructor definitions

- Binding then introduces a type constructor, not a type
  - Must say **`int mylist`** or **`string mylist`** or **`'a mylist`**
  - Not "plain" **`mylist`**

Nothing else changes

Use constructors and case expressions as usual

- No change to evaluation rules
- Type-checking will make sure types are used consistently
  - Example: cannot mix element types of list
- Functions will be polymorphic or not based on how data is used