```
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

The Subtype Relation

# *Keeping subtyping separate*

A programming language already has a lot of typing rules and we do not want to change them

- Example: The type of an actual function argument must ***equal*** the type of the function parameter

We can do this by adding "just two things to our language"

- *Subtyping*: Write **`t1 <: t2`** for **`t1`** is a subtype of t2
- One new typing rule that uses subtyping:

  If **`e`** has type **`t1`** and **`t1 <: t2`**,

  then **`e`** (also) has type **`t2`**

Now all we need to do is define **`t1 <: t2`**

# *Subtyping is not a matter of opinion*

- Misconception: If we are making a new language, we can have whatever typing and subtyping rules we want

- Not if you want to prevent what you claim to prevent [soundness]
  - Here: No accessing record fields that do not exist

- Our typing rules were *sound* before we added subtyping
  - We should keep it that way

- Principle of *substitutability*: If **`t1 <: t2`**, then any value of type **`t1`** must be usable in every way a **`t2`** is
  - Here: Any value of subtype needs all fields any value of supertype has

# *Four good rules*

For our record types, these rules all meet the substitutability test:

1. "Width" subtyping: A supertype can have a subset of fields with the same types
2. "Permutation" subtyping: A supertype can have the same set of fields with the same types in a different order
3. Transitivity: If **`t1 <: t2`** and **`t2 <: t3`**, then **`t1 <: t3`**
4. Reflexivity: Every type is a subtype of itself

(4) may seem unnecessary, but it composes well with other rules in a full language and "does no harm"