```
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

Subtyping From the Beginning

Last major topic

Build up key ideas from first principles

- In pseudocode because:
  - No time for another language
  - Simple to first show subtyping without objects

Then, a few segments from now:

- How does subtyping relate to types for OOP?
  - Brief sketch only
- What are the relative strengths of subtyping and generics?
- How can subtyping and generics combine synergistically?

A tiny language

- Can cover most core subtyping ideas by just considering *records with mutable fields*

- Will make up our own syntax
  - ML has records, but no subtyping or field-mutation
  - Racket and Ruby have no type system
  - Java uses class/interface names and rarely fits on a slide

Records (half like ML, half like Java)

Record creation (field names and contents):

`{f1=e1, f2=e2, …, fn=en}` ke a record

Record field access:

`e.f` aluate **`e`** to record **`v`** with an **`f`** field, get contents of **`f`** field

Record field update

`e1.f = e2` Evaluate **`e1`** to a record **`v1`** and **`e2`** to a value **`v2`**;

Change **`v1`**'s **`f`** field (which must exist) to **`v2`**;

Return **`v2`**

A Basic Type System

Record types: What fields a record has and type for each field

`{f1:t1, f2:t2, …, fn:tn}`

Type-checking expressions:

- If `e1` has type `t1`, …, `en` has type `tn`,
  then `{f1=e1, …, fn=en}` has type `{f1:t1, …, fn:tn}`
- If `e` has a record type containing `f : t`,
  then `e.f` has type `t`
- If `e1` has a record type containing `f : t` and `e2` has type `t`,
  then `e1.f = e2` has type `t`

This is safe

These evaluation rules and typing rules prevent ever trying to access a field of a record that does not exist

Example program that type-checks (in a made-up language):

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```

Motivating subtyping

But according to our typing rules, this program does not type-check

- It does nothing wrong and seems worth supporting

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val c : {x:real,y:real,color:string} =
   {x=3.0, y=4.0, color="green"}

val five : real = distToOrigin(c)
```

A good idea: allow extra fields

Natural idea: If an expression has type

`{f1:t1, f2:t2, …, fn:tn}`

Then it can *also* have a type with some fields removed

This is what we need to type-check these function calls:

```
fun distToOrigin (p:{x:real,y:real}) = …
fun makePurple (p:{color:string}) = …

val c :{x:real,y:real,color:string} =
    {x=3.0, y=4.0, color="green"}

val _ = distToOrigin(c)
val _ = makePurple(c)
```