```
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

Subclassing

Next major topic

- Subclasses, inheritance, and overriding
  - The essence of OOP
  - Not unlike you may have seen in Java/C#/C++/Python, but worth studying from PL perspective and in a more dynamic language

Subclassing

- A class definition has a *superclass* (`Object` if not specified)

  `class ColorPoint < Point` …

- The superclass affects the class definition:
  - Class *inherits* all method definitions from superclass
  - But class can *override* method definitions as desired

- Unlike Java/C#/C++:
  - No such thing as "inheriting fields" since all objects create instance variables by assigning to them
  - Subclassing has nothing to do with a (non-existent) type system: can still (try to) call any method on any object

Example (to be continued)

```ruby
class Point
  attr_accessor  :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    # direct field access
    Math.sqrt(@x*@x
              + @y*@y)
  end
  def distFromOrigin2
    # use getters
    Math.sqrt(x*x
              + y*y)
  end
end
```

```ruby
class ColorPoint < Point
  attr_accessor  :color
  def initialize(x,y,c)
    super(x,y)
    @color = c
  end
end
```

An object has a class

```
p  = Point.new(0,0)
cp = ColorPoint.new(0,0,"red")
p.class                             # Point
p.class.superclass                  # Object
cp.class                            # ColorPoint
cp.class.superclass                 # Point
cp.class.superclass.superclass      # Object
cp.is_a? Point                      # true
cp.instance_of? Point               # false
cp.is_a? ColorPoint                 # true
cp.instance_of? ColorPoint          # true
```

- Using these methods is usually non-OOP style
  - Disallows other things that "act like a duck"
  - Nonetheless semantics is that an instance of `ColorPoint` "is a" `Point` but is not an "instance of" `Point`
  - [Java note: `instanceof` is like Ruby's `is_a?`]