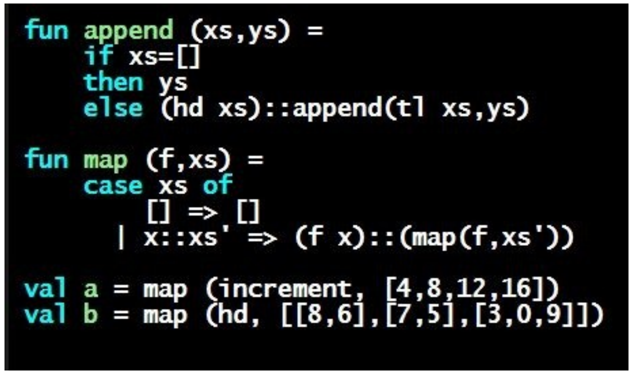
```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

Method-Lookup Rules, Precisely

Dynamic dispatch

Dynamic dispatch

- Also known as *late binding* or *virtual methods*
- Call **`self.m2()`** in method **`m1`** defined in class **`C`** can *resolve to* a method **`m2`** defined in a subclass of **`C`**
- Most unique characteristic of OOP

Need to define the semantics of *method lookup* as carefully as we defined *variable lookup* for our PLs

Review: variable lookup

Rules for "looking things up" is a key part of PL semantics

- ML: Look up *variables* in the appropriate environment
  - Lexical scope for closures
  - *Field names* (for records) are different: not variables

- Racket: Like ML plus **`let`**, **`letrec`**

- Ruby:
  - Local variables and blocks mostly like ML and Racket
  - But also have instance variables, class variables, methods (all more like record fields)
    - Look up in terms of **`self`**, which is special

Using `self`

- `self` maps to some "current" object
- Look up instance variable `@x` using object bound to `self`
- Look up class variables `@@x` using object bound to `self.class`
- Look up methods…

Ruby method lookup

The semantics for method calls also known as message sends

`e0.m(e1,…,en)`

1. Evaluate **`e0`**, **`e1`**, …, **`en`** to objects **`obj0`**, **`obj1`**, …, **`objn`**
   - As usual, may involve looking up **`self`**, variables, fields, etc.
2. Let **`C`** be the class of **`obj0`** (every object has a class)
3. If **`m`** is defined in **`C`**, pick that method, else recur with the superclass of **`C`** unless **`C`** is already **`Object`**
   - If no **`m`** is found, call **`method_missing`** instead
     - Definition of **`method_missing`** in **`Object`** raises an error
4. Evaluate body of method picked:
   - With formal arguments bound to **`obj1`**, …, **`objn`**
   - With **`self`** bound to **`obj0`** -- this implements dynamic dispatch!

Note: Step (3) complicated by *mixins*: will revise definition later

Punch-line again

`e0.m(e1,…,en)`

To implement dynamic dispatch, evaluate the method body with **`self`** mapping to the *receiver* (result of **`e0`**)

- That way, any **`self`** calls in body of **`m`** use the receiver's class,
  - Not necessarily the class that defined **`m`**

- This much is the same in Ruby, Java, C#, Smalltalk, etc.

Comments on dynamic dispatch

- This is why **`distFromOrigin2`** worked in **`PolarPoint`**
- More complicated than the rules for closures
  - Have to treat **`self`** specially
  - May seem simpler only if you learned it first
  - Complicated does not necessarily mean inferior or superior

Optional: static overloading

In Java/C#/C++, method-lookup rules are similar, but more complicated because > 1 methods in a class can have same name

- Java/C/C++: Overriding only when number/types of arguments the same
- Ruby: same-method-name always overriding

Pick the “best one” using the (static) types of the arguments

- Complicated rules for “best”
- Type-checking error if there is no “best”

Relies fundamentally on type-checking rules

- Ruby has none