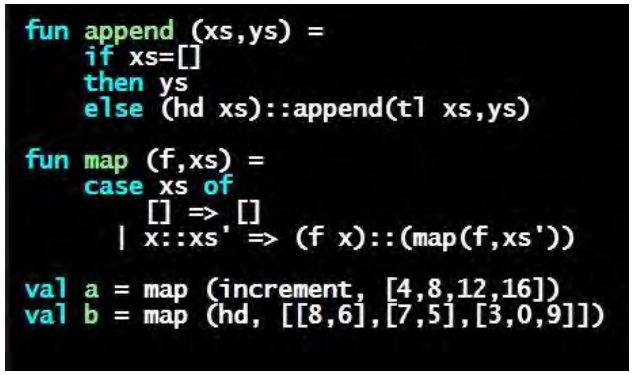


Programming Languages

Dan Grossman

Introduction to Ruby

Ruby logistics

- Next two sections use the Ruby language
  - http://www.ruby-lang.org/
  - Installation / basic usage instructions on course website
    - Emacs in lectures, but many editors support Ruby well
    - We support a few versions; differences not so relevant

- Excellent documentation available, much of it free
  - So may not cover every language detail in course materials
  - http://ruby-doc.org/
  - http://www.ruby-lang.org/en/documentation/
  - Particularly recommend “Programming Ruby 1.9 & 2.0, The Pragmatic Programmers’ Guide”
    - Not required and not free

Ruby: Our focus

- *Pure object-oriented*: *all* values are *objects* (even numbers)
- *Class-based*: Every object has a class that determines behavior
  - Like Java, unlike Javascript
  - *Mixins* (neither Java interfaces nor C++ multiple inheritance)
- *Dynamically typed*
- Convenient *reflection*: Run-time inspection of objects
- Very *dynamic*: Can change classes during execution
- *Blocks* and libraries encourage lots of closure idioms
- Syntax, scoping rules, semantics of a "*scripting language*"
  - Variables "spring to life" on use
  - Very flexible arrays

Ruby: Not our focus

- Lots of support for string manipulation and regular expressions
- Popular for server-side web applications
  - Ruby on Rails
- Often many ways to do the same thing
  - More of a “why not add that too?” approach

Where Ruby fits

	dynamically typed	statically typed
functional	Racket	SML
object-oriented (OOP)	Ruby	Java, etc.

Note: Racket also has classes and objects when you want them

- In Ruby everything uses them (at least implicitly)

Historical note: *Smalltalk* also a dynamically typed, class-based, pure OOP language with blocks and convenient reflection

- Smaller just-as-powerful language
- Ruby less simple, more “modern and useful”

Dynamically typed OOP helps identify OOP's essence by not having to discuss types

A note on the homework

Next homework is about understanding and extending an *existing* program in an *unfamiliar* language

- Good practice
- Quite different feel than previous homeworks
- *Read* code: determine what you do and do not (!) need to understand

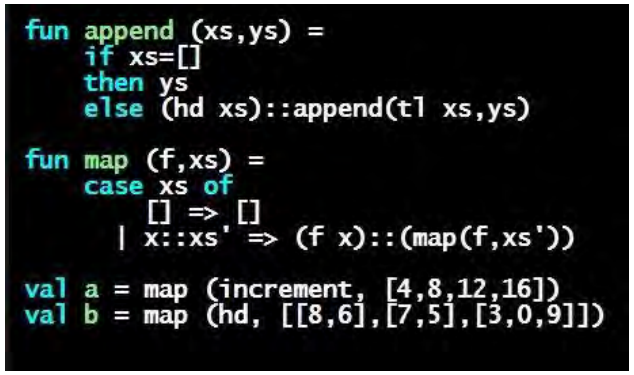
Homework requires the Tk graphics library to be installed such that the provided Ruby code can use it

Getting started

- See **`.rb`** file for our first program
  - (There are much shorter ways to write the same thing)

- Can run file **`foo.rb`** at the command-line with **`ruby foo.rb`**

- Or can use **`irb`**, which is a REPL
  - Run file **`foo.rb`** with **`load "foo.rb"`**



Programming Languages

Dan Grossman

Classes and Objects

The rules of class-based OOP

In Ruby:

1. All values are references to *objects*
2. Objects communicate via *method calls*, also known as *messages*
3. Each object has its own (private) *state*
4. Every object is an instance of a *class*
5. An object's class determines the object's *behavior*
   - How it handles method calls
   - Class contains method definitions

Java/C#/etc. similar but do not follow (1) (e.g., numbers, null) and allow objects to have non-private state

Defining classes and methods

```
class Name
  def method_name1 method_args1
    expression1
  end
  def method_name2 method_args2
    expression2
  end
  …
end
```

- Define a new class called with methods as defined
- Method returns its last expression
  - Ruby also has explicit **`return`** statement
- Syntax note: Line breaks often required (else need more syntax), but indentation always only style

Creating and using an object

- **`ClassName.new`** creates a new object whose class is **`ClassName`**

- **`e.m`** evaluates **`e`** to an object and then calls its **`m`** method
  - Also known as "sends the **`m`** message"
  - Can also write **`e.m()`**

- Methods can take arguments, called like **`e.m(e1,…,en)`**
  - Parentheses optional in some places, but recommended

Variables

- Methods can use local variables
  - Syntax: starts with letter
  - Scope is method body
- No declaring them, just assign to them anywhere in method body (!)
- Variables are mutable, **`x=e`**
- Variables also allowed at “top-level” or in REPL
- Contents of variables are always references to objects because all values are objects

Self

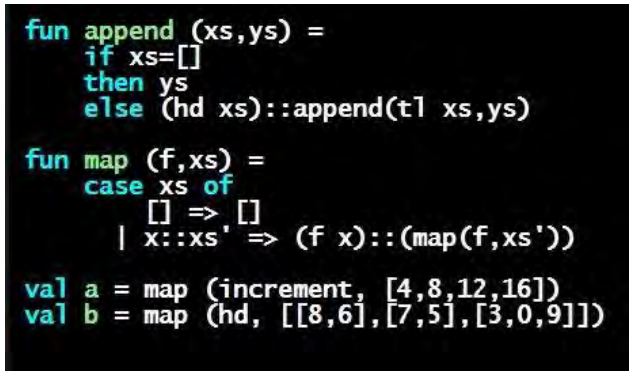
- **`self`** is a special keyword/variable in Ruby

- Refers to “the current object”
  - The object whose method is executing

- So call another method on “same object” with **`self.m(…)`**
  - Syntactic sugar: can just write **`m(…)`**

- Also can pass/return/store “the whole object” with just **`self`**

- (Same as **`this`** in Java/C#/C++)



Programming Languages

Dan Grossman

Object State

Objects have state

- An object's state persists
  - Can grow and change from time object is created

- State only directly accessible from object's methods
  - Can read, write, extend the state
  - Effects persist for next method call

- State consists of *instance variables* (also known as fields)
  - Syntax: starts with an `@`, e.g., **`@foo`**
  - "Spring into being" with assignment
    - So mis-spellings silently add new state (!)
  - Using one not in state not an error; produces **`nil`** object

Aliasing

- Creating an object returns a reference to a new object
  - Different state from every other object

- Variable assignment (e.g., **`x=y`**) creates an alias
  - Aliasing means same object means same state

Initialization

- A method named **`initialize`** is special
  - Is called on a new object before **`new`** returns
  - Arguments to **`new`** are passed on to **`initialize`**
  - Excellent for creating object invariants
  - (Like constructors in Java/C#/etc.)

- Usually good *style* to create instance variables in **`initialize`**
  - Just a convention
  - Unlike OOP languages that make "what fields an object has" a (fixed) part of the class definition
    - In Ruby, different instances of same class can have different instance variables

Class variables

- There is also state shared by the entire class
- Shared by (and only accessible to) all instances of the class
- Called *class variables*
  - Syntax: starts with an `@@`, e.g., `@@foo`
- Less common, but sometimes useful
  - And helps explain via contrast that each object has its own instance variables

Class constants and methods

- *Class constants*
  - Syntax: start with capital letter, e.g., `Foo`
  - Should not be mutated
  - Visible outside class `C` as `C::Foo` (unlike class variables)

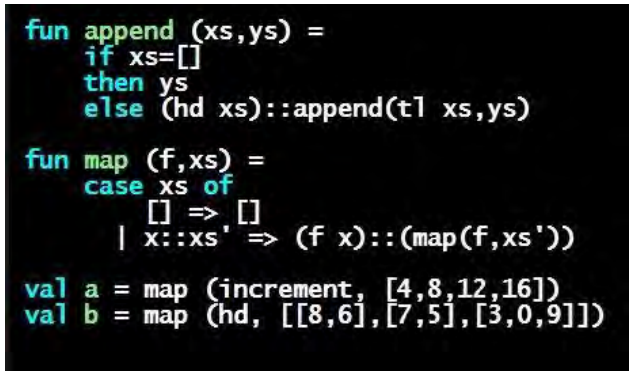
- *Class methods* (cf. Java/C# static methods)
  - Syntax (in some class `C`):

```
def self.method_name (args)
  …
end
```

  - Use (of class method in class `C`):

```
C.method_name(args)
```

  - Part of the class, not a particular instance of it



Programming Languages

Dan Grossman

Visibility

Who can access what

- We know “hiding things” is essential for modularity and abstraction
- OOP languages generally have various ways to hide (or not) instance variables, methods, classes, etc.
  - Ruby is no exception
- Some basic Ruby rules here as an example…

Object state is private

- In Ruby, object state is always private
  - Only an object's methods can access its instance variables
  - Not even another instance of the same class
  - So can write `@foo`, but not `e.@foo`

- To make object-state publicly visible, define "getters" / "setters"
  - Better/shorter style coming next

```
def get_foo
  @foo
end
def set_foo x
  @foo = x
end
```

Conventions and sugar

- Actually, for field `@foo` the convention is to name the methods

```
def foo
  @foo
end
```

```
def foo= x
  @foo = x
end
```

- Cute sugar: When *using* a method ending in `=`, can have space before the `=`

```
e.foo = 42
```

- Because defining getters/setters is so common, there is shorthand for it in class definitions
  - Define just getters: `attr_reader :foo, :bar, …`
  - Define getters and setters: `attr_accessor :foo, :bar, …`

- Despite sugar: getters/setters are just methods

Why private object state

- This is “more OOP” than public instance variables
- Can later change class implementation without changing clients
  - Like we did with ML modules that hid representation
  - And like we will soon do with subclasses
- Can have methods that “seem like” setters even if they are not

```
def celsius_temp= x
  @kelvin_temp = x + 273.15
end
```

- Can have an unrelated class that implements the same methods and use it with same clients
  - See later discussion of “duck typing”

Method visibility

- Three *visibilities* for methods in Ruby:
  - **`private`**: only available to object itself
  - **`protected`**: available only to code in the class or subclasses
  - **`public`**: available to all code

- Methods are **`public`** by default
  - Multiple ways to change a method's visibility
  - Here is one way…

Method visibilities

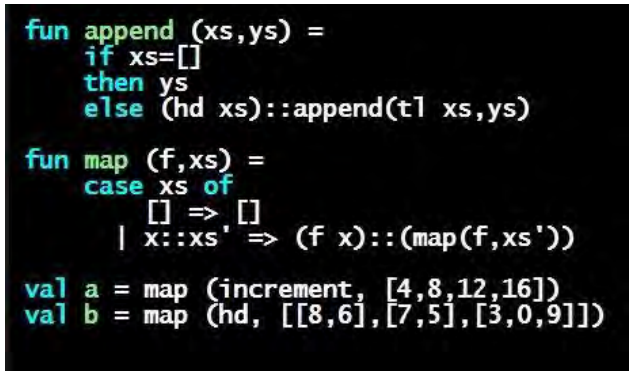
```
class Foo =
# by default methods public
    …
 protected
# now methods will be protected until
# next visibility keyword
    …
 public
    …

 private
    …
end
```

One detail

If **`m`** is private, then you can only call it via **`m`** or **`m(args)`**

- As usual, this is shorthand for **`self.m`** …
- But for private methods, only the shorthand is allowed



```
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

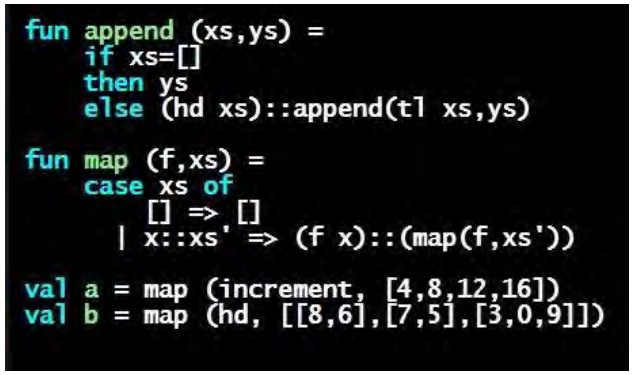
A Longer Example

Now

- Put together much of what we have learned to define and use a small class for rational numbers
  - Called **`MyRational`** because Ruby 1.9 has great built-in support for fractions using a class **`Rational`**

- Will also use several new and useful expression forms
  - Ruby is too big to show everything; see the documentation

- Way our class works: Keeps fractions in reduced form with a positive denominator
  - Like an ML-module example earlier in course



Programming Languages

Dan Grossman

Everything is an Object

Pure OOP

- Ruby is fully committed to OOP:

  Every value is a reference to an object

- Simpler, smaller semantics

- Can call methods on anything
  - May just get a dynamic "undefined method" error

- Almost everything is a method call
  - Example: `3 + 4`

Some examples

- Numbers have methods like **`+`**, **`abs`**, **`nonzero?`**, etc.

- **`nil`** is an object used as a "nothing" object
  - Like **`null`** in Java/C#/C++ except it is an object
  - Every object has a **`nil?`** method, where **`nil`** returns **`true`** for it
  - Note: **`nil`** and **`false`** are "false", everything else is "true"

- Strings also have a **`+`** method
  - String concatenation
  - Example: **`"hello" + 3.to_s`**

All code is methods

- All methods you define are part of a class
- Top-level methods (in file or REPL) just added to **`Object`** class
- Subclassing discussion coming later, but:
  - Since all classes you define are *subclasses* of **`Object`**, all *inherit* the top-level methods
  - So you can call these methods anywhere in the program
  - Unless a class overrides (*roughly-not-exactly*, shadows) it by defining a method with the same name

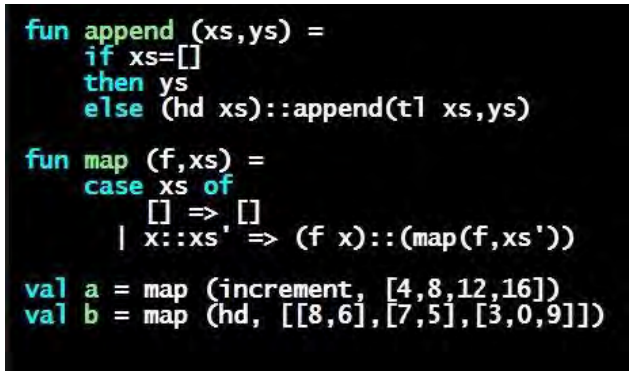
Reflection and exploratory programming

- All objects also have methods like:
  - **`methods`**
  - **`class`**

- Can use at run-time to query "what an object can do" and respond accordingly
  - Called *reflection*

- Also useful in the REPL to explore what methods are available
  - May be quicker than consulting full documentation

- Another example of "just objects and method calls"



Programming Languages

Dan Grossman

Class Definitions are Dynamic

Changing classes

- Ruby programs (or the REPL) can add/change/replace methods while a program is running

- Breaks abstractions and makes programs very difficult to analyze, but it does have plausible uses
  - Simple example: Add a useful helper method to a class you did not define
    - Controversial in large programs, but may be useful

- For us: Helps re-enforce “the rules of OOP”
  - Every object has a class
  - A class determines its instances’ behavior

Examples

- Add a **`double`** method to our **`MyRational`** class
- Add a **`double`** method to the built-in **`FixNum`** class
- Defining top-level methods adds to the built-in **`Object`** class
  - Or replaces methods
- Replace the **`+`** method in the built-in **`FixNum`** class
  - Oops: watch **`irb`** crash

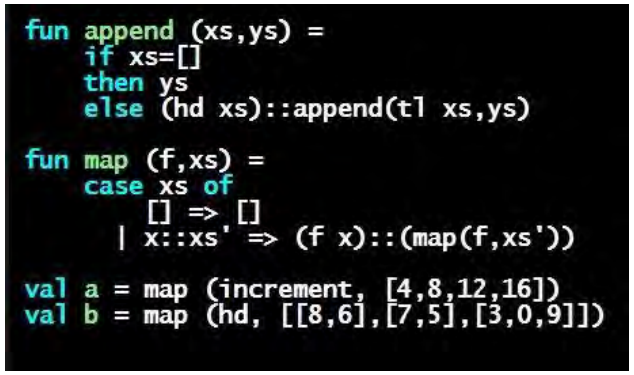
The moral

- Dynamic features cause interesting semantic questions

- Example:
  - First create an instance of class `C`, e.g., `x = C.new`
  - Now replace method method `m` in `C`
  - Now call `x.m`

  Old method or new method? In Ruby, new method

  The point is Java/C#/C++ do not have to ask the question
  - May allow more optimized method-call implementations as a result



Programming Languages

Dan Grossman

Duck Typing

Duck Typing

"If it walks like a duck and quacks like a duck, it's a duck"

- Or don't worry that it may not be a duck

When writing a method you might think, "I need a `Foo` argument" but really you need an object with enough methods similar to `Foo`'s methods that your method works

- Embracing duck typing is always making method calls rather than assuming/testing the class of arguments

Plus: More code reuse; very OOP approach

- What messages an object receive is "all that matters"

Minus: Almost nothing is equivalent

- `x+x` versus `x*2` versus `2*x`
- Callers may assume a lot about how callees are implemented

Duck Typing Example

```
def mirror_update pt
  pt.x = pt.x * (-1)
end
```

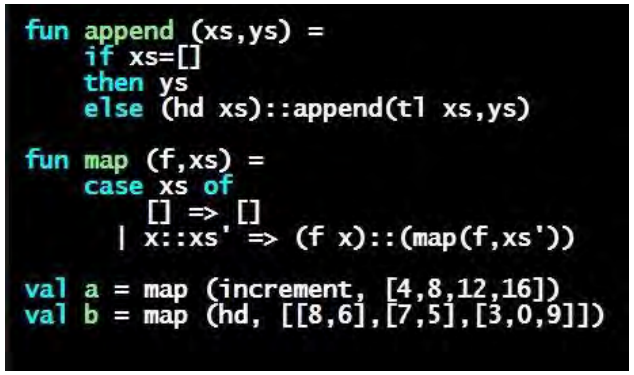
- Natural thought: "Takes a **`Point`** object (definition not shown here), negates the **`x`** value"
  - Makes sense, though a **`Point`** instance method more OOP
- Closer: "Takes anything with getter and setter methods for **`@x`** instance variable and multiplies the **`x`** field by **`-1`**"
- Closer: "Takes anything with methods **`x=`** and **`x`** and calls **`x=`** with the result of multiplying result of **`x`** and **`-1`**"
- Duck typing: "Takes anything with method **`x=`** and **`x`** where result of **`x`** has a **`*`** method that can take **`-1`**. Sends result of calling **`x`** the **`*`** message with **`-1`** and sends that result to **`x=`**"

With our example

```
def mirror_update pt
  pt.x = pt.x * (-1)
end
```

- Plus: Maybe **`mirror_update`** is useful for classes we did not anticipate
- Minus: If someone does use (abuse?) duck typing here, then we cannot change the implementation of **`mirror_update`**
  - For example, to **`- pt.x`**
- Better (?) example: Can pass this method a number, a string, or a **`MyRational`**

```
def double x
  x + x
end
```



Programming Languages

Dan Grossman

Arrays

Ruby Arrays

- Lots of special syntax and many provided methods for the **`Array`** class

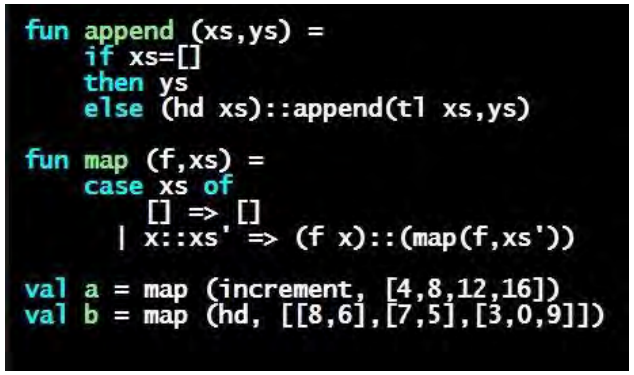
- Can hold any number of other objects, *indexed* by number
  - Get via **`a[i]`**
  - Set via **`a[i] = e`**

- Compared to arrays in many other languages
  - More flexible and dynamic
  - Fewer operations are errors
  - Less efficient

- "The standard collection" (like lists were in ML and Racket)

Using Arrays

- See many examples, some demonstrated here
- Consult the documentation/tutorials
  - If seems sensible and general, probably a method for it
- Arrays make good tuples, lists, stacks, queues, sets, …
- Iterating over arrays typically done with methods taking blocks
  - Next topic…



Programming Languages

Dan Grossman

Passing Blocks

Blocks

Blocks are probably Ruby's strangest feature compared to other PLs

But *almost* just closures

- Normal: easy way to pass anonymous functions to methods for all the usual reasons
- Normal: Blocks can take 0 or more arguments
- Normal: Blocks use lexical scope: block body uses environment where block was defined

Examples:

```
3.times { puts "hi" }
[4,6,8].each { puts "hi" }
i = 7
[4,6,8].each {|x| if i > x then puts (x+1) end }
```

Some strange things

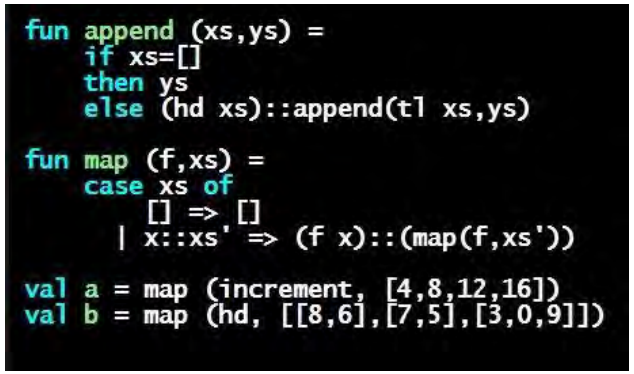
- Can pass 0 or 1 block with *any* message
  - Callee might ignore it
  - Callee might give an error if you do not send one
  - Callee might do different things if you do/don't send one
    - Also number-of-block-arguments can matter

- Just put the block "next to" the "other" arguments (if any)
  - Syntax: `{e}`, `{|x| e}`, `{|x,y| e}`, etc. (plus variations)
    - Can also replace `{` and `}` with **`do`** and **`end`**
      - Often preferred for blocks > 1 line

Blocks everywhere

- Rampant use of great block-taking methods in standard libraray
- Ruby has loops but very rarely used
  - Can write `(0..i).each {|j| e}`, but often better options
- Examples (consult documentation for many more)

```
a = Array.new(5) {|i| 4*(i+1)}
a.each { puts "hi" }
a.each {|x| puts (x * 2) }
a.map  {|x| x * 2 } #synonym: collect
a.any? {|x| x > 7 }
a.all? {|x| x > 7 }
a.inject(0) {|acc,elt| acc+elt }
a.select {|x| x > 7 } #non-synonym: filter
```



Programming Languages

Dan Grossman

Using Blocks

More strangeness

- Callee does not give a name to the (potential) block argument

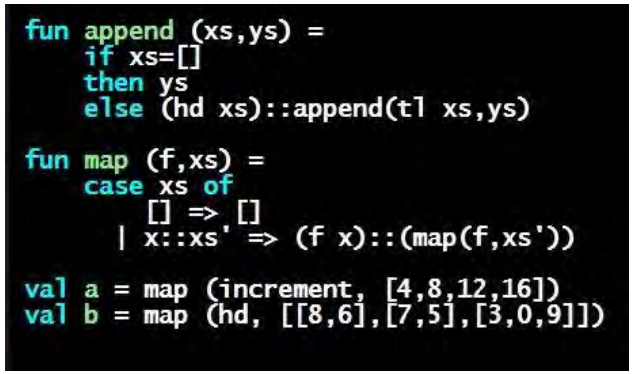
- Instead, just calls it with **`yield`** or **`yield(args)`**
  - Silly example:

```
def silly a
  (yield a) + (yield 42)
end
```

```
x.silly(5) {|b| b*2 }
```

  - See code for slightly less silly example

- Can ask **`block_given?`** but often just assume a block is given or that a block's presence is implied by other arguments



```
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

Procs

Blocks are "second-class"

All a method can do with a block is **`yield`** to it

- Cannot return it, store it in an object (e.g., for a callback), …
- But can also turn blocks into real closures
- Closures are instances of class **`Proc`**
  - Called with method **`call`**

This is Ruby, so there are several ways to make **`Proc`** objects ☺

- One way: method **`lambda`** of **`Object`** takes a block and returns the corresponding **`Proc`**

Example

```
a = [3,5,7,9]
```

- Blocks are fine for applying to array elements

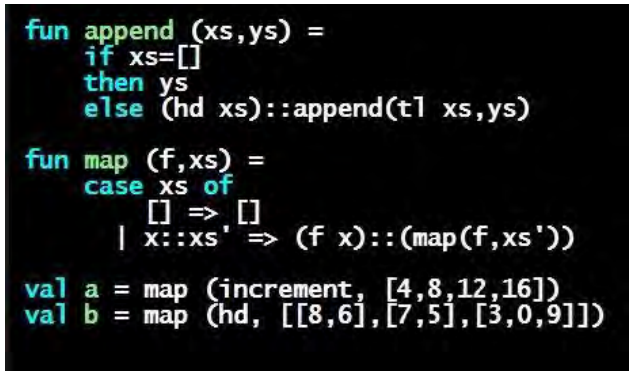
```
b = a.map {|x| x+1 }
i = b.count {|x| x>=6 }
```

- But for an array of closures, need `Proc` objects
  - More common use is callbacks

```
c = a.map {|x| lambda {|y| x>=y}}
c[2].call 17
j = c.count {|x| x.call(5) }
```

Moral

- First-class ("can be passed/stored anywhere") makes closures more powerful than blocks
- But blocks are (a little) more convenient and cover most uses
- This helps us understand what first-class means
- Language design question: When is convenience worth making something less general and powerful?



Programming Languages

Dan Grossman

Hashes and Ranges

More collections

- *Hashes* like arrays but:
  - *Keys* can be *anything*; strings and symbols common
  - No natural ordering like numeric indices
  - Different syntax to make them

  Like a dynamic record with anything for field names
  - Often pass a hash rather than many arguments

- *Ranges* like arrays of contiguous numbers but:
  - More efficiently represented, so large ranges fine

Good style to:

  - Use ranges when you can
  - Use hashes when non-numeric keys better represent data

Similar methods

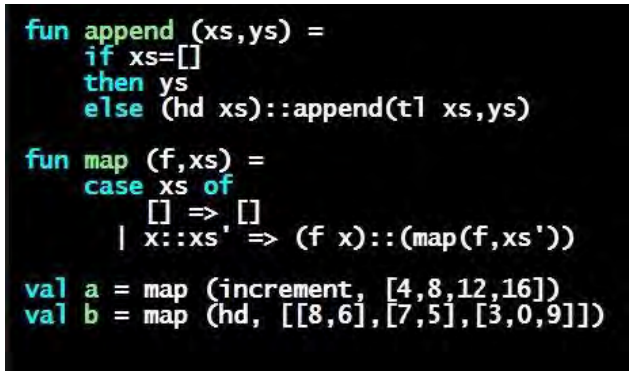
- Arrays, hashes, and ranges all have some methods other don't
  - E.g., **`keys`** and **`values`**

- But also have many of the same methods, particularly iterators
  - Great for duck typing
  - Example

```
def foo a
  a.count {|x| x*x < 50}
end

foo [3,5,7,9]
foo (3..9)
```

Once again separating "how to iterate" from "what to do"



Programming Languages

Dan Grossman

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)

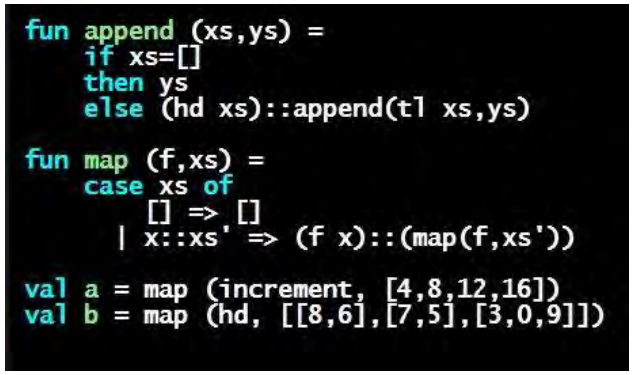
```
class Point
  attr_accessor  :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    # direct field access
    Math.sqrt(@x*@x
              + @y*@y)
  end
  def distFromOrigin2
    # use getters
    Math.sqrt(x*x
              + y*y)
  end
end
```

```
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?`**]



Programming Languages

Dan Grossman

Why Use Subclassing?

Example continued

- Consider alternatives to:

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

- Here subclassing is a good choice, but programmers often overuse subclassing in OOP languages

Why subclass

- Instead of creating **`ColorPoint`**, could add methods to **`Point`**
  - That could mess up other users and subclassers of **`Point`**

```
class Point
  attr_accessor :color
  def initialize(x,y,c="clear")
    @x = x
    @y = y
    @color = c
  end
end
```

Why subclass

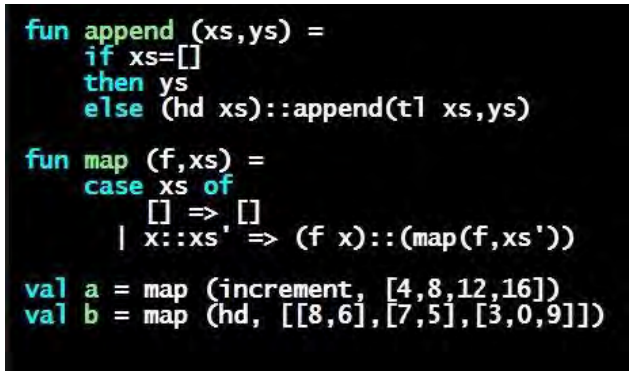
- Instead of subclassing **`Point`**, could copy/paste the methods
  - Means the same thing *if* you don't use methods like **`is_a?`** and **`superclass`**, but of course code reuse is nice

```
class ColorPoint
  attr_accessor :x, :y, :color
  def initialize(x,y,c="clear")
    …
  end
  def distFromOrigin
    Math.sqrt(@x*@x + @y*@y)
  end
  def distFromOrigin2
    Math.sqrt(x*x + y*y)
  end
end
```

Why subclass

- Instead of subclassing **`Point`**, could use a **`Point`** instance variable
  - Define methods to send same message to the **`Point`**
  - Often OOP programmers overuse subclassing
  - But for **`ColorPoint`**, subclassing makes sense: less work and can use a **`ColorPoint`** wherever code expects a **`Point`**

```
class ColorPoint
  attr_accessor :color
  def initialize(x,y,c="clear")
     @pt = Point.new(x,y)
     @color = c
  end
  def x
    @pt.x
  end
  … # similar "forwarding" methods
    # for y, x=, y=
end
```



Programming Languages

Dan Grossman

Overriding and Dynamic Dispatch

Overriding

- **`ThreeDPoint`** is more interesting than **`ColorPoint`** because it overrides **`distFromOrigin`** and **`distFromOrigin2`**
  - Gets code reuse, but *highly disputable* if it is appropriate to say a **`ThreeDPoint`** "is a" **`Point`**
  - Still just avoiding copy/paste

```
class ThreeDPoint < Point
  …
  def initialize(x,y,z)
     super(x,y)
     @z = z
  end
  def distFromOrigin # distFromOrigin2 similar
    d = super
    Math.sqrt(d*d + @z*@z)
  end
  …
end
```

So far…

- With examples so far, objects are not so different from closures
  - Multiple methods rather than just "call me"
  - Explicit instance variables rather than environment where function is defined
  - Inheritance avoids helper functions or code copying
  - “Simple” overriding just replaces methods

- But there is one big difference:

  Overriding can make a method defined in the superclass call a method in the subclass

  - *The* essential difference of OOP, studied carefully next lecture

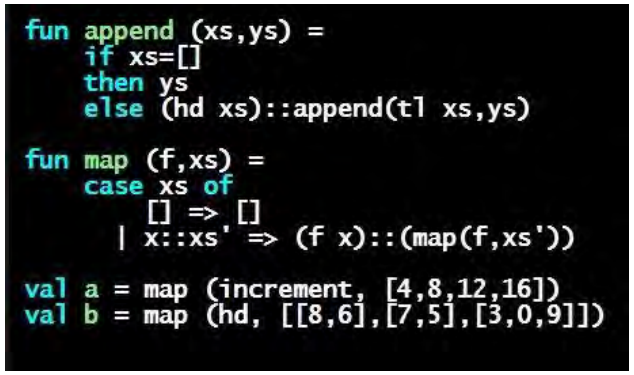
Example: Equivalent except constructor

```
class PolarPoint < Point
  def initialize(r,theta)
    @r = r
    @theta = theta
  end
  def x
    @r * Math.cos(@theta)
  end
  def y
    @r * Math.sin(@theta)
  end
  def distFromOrigin
    @r
  end
  …
end
```

- Also need to define **`x=`** and **`y=`** (see code file)
- Key punchline: **`distFromOrigin2`**, defined in **`Point`**, "already works"

```
def distFromOrigin2
  Math.sqrt(x*x+y*y)
end
```

  - Why: calls to **`self`** are resolved in terms of the object's class



Programming Languages

Dan Grossman

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

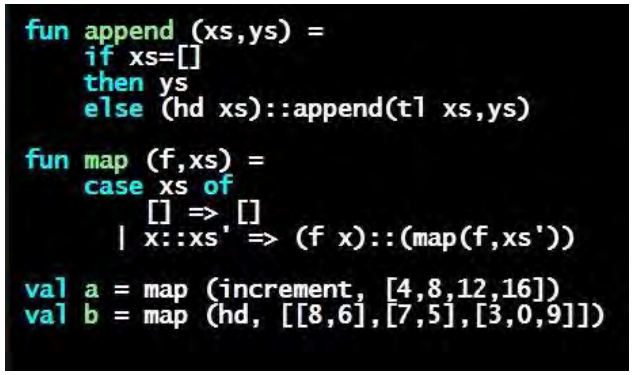
- Java/C/C++: Overriding only when number/types of arguments the same
- Ruby: same-method-name always overriding

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



Programming Languages

Dan Grossman

Dynamic Dispatch Versus Closures

A simple example, part 1

In ML (and other languages), closures are closed

```
fun even x = if x=0 then true  else odd  (x-1)
and odd  x = if x=0 then false else even (x-1)
```

So we can shadow **odd**, but any call to the closure bound to **odd** above will "do what we expect"

- Does not matter if we shadow **even** or not

```
(* does not change odd - too bad; this would
   improve it *)
fun even x = (x mod 2)=0
```

```
(* does not change odd - good thing; this would
   break it *)
fun even x = false
```

A simple example, part 2

In Ruby (and other OOP languages), subclasses can change the behavior of methods they do not override

```
class A
  def even x
    if x==0 then true  else odd  (x-1) end
  end
  def odd x
    if x==0 then false else even (x-1) end
  end
end
class B < A  # improves odd in B objects
  def even x ; x % 2 == 0 end
end
class C < A  # breaks odd in C objects
  def even x ; false end
end
```

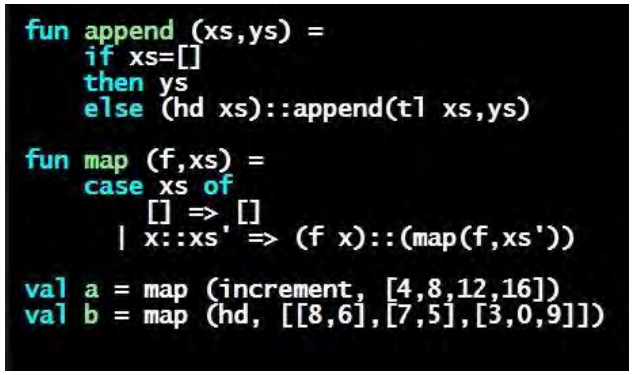
The OOP trade-off

Any method that makes calls to overridable methods can have its behavior changed in subclasses even if it is not overridden

- Maybe on purpose, maybe by mistake
- Observable behavior includes calls-to-overridable methods

- So *harder* to reason about "the code you're looking at"
  - Can avoid by disallowing overriding
    - "private" or "final" methods

- So *easier* for subclasses to affect behavior without copying code
  - Provided method in superclass is not modified later



Programming Languages

Dan Grossman

Optional: Dynamic Dispatch Manually in Racket

Manual dynamic dispatch

Now: Write Racket code with little more than pairs and functions that *acts like* objects with dynamic dispatch

Why do this?

- (Racket actually has classes and objects available)

- Demonstrates how one language's *semantics* is an idiom in another language
- Understand dynamic dispatch better by coding it up
  - Roughly how an interpreter/compiler might

Analogy: Earlier optional material encoding higher-order functions using objects and explicit environments

Our approach

Many ways to do it; our code does this:

- An "object" has a list of field pairs and a list of method pairs

```
(struct obj (fields methods))
```

- Field-list element example:

```
(mcons 'x 17)
```

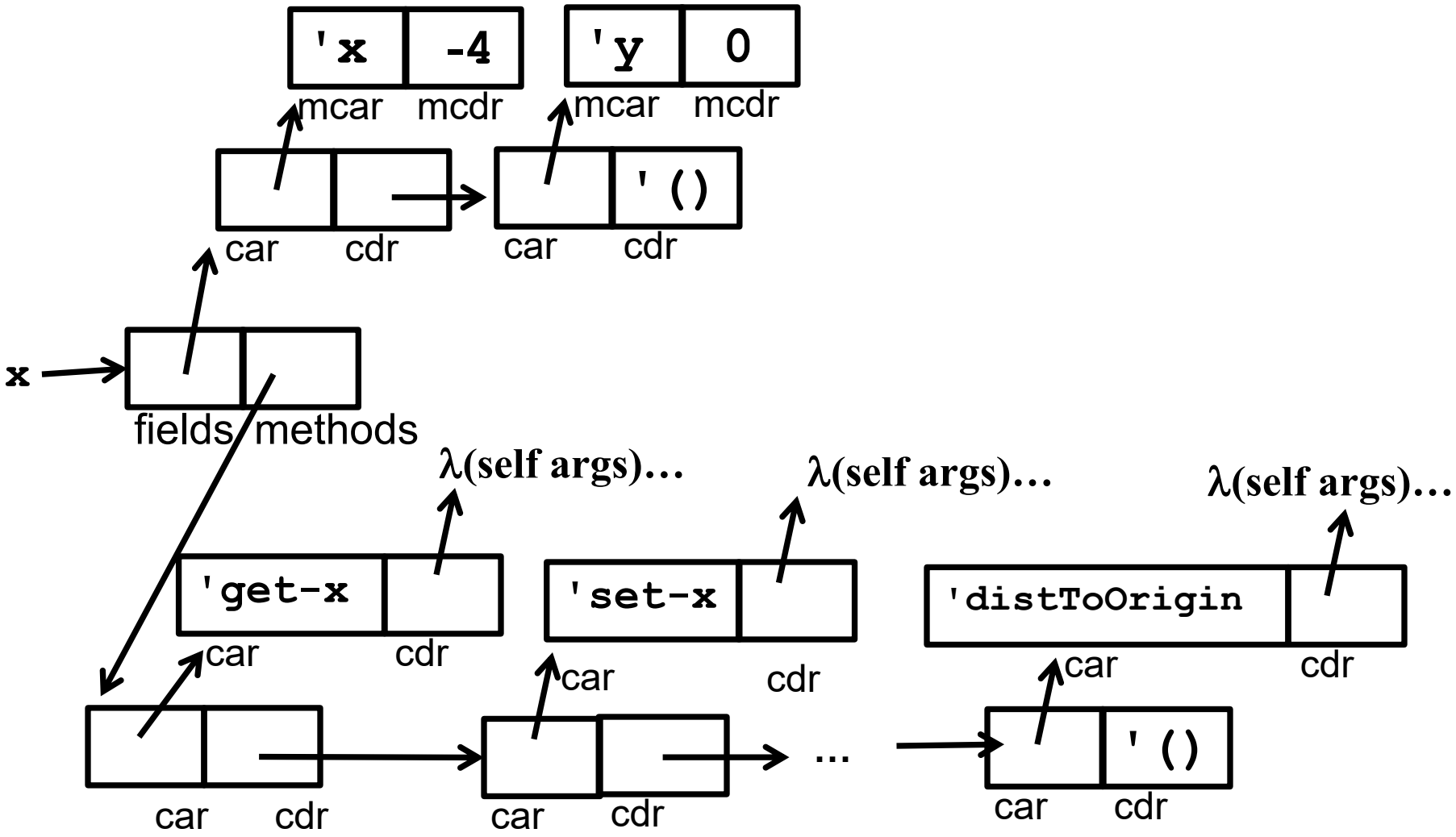
- Method-list element example:

```
(cons 'get-x (lambda (self args) …))
```

Notes:

- Lists sufficient but not efficient
- Not class-based: object has a list of methods, not a class that has a list of methods [could do it that way instead]
- Key trick is lambdas taking an extra **`self`** argument
  - All "regular" arguments put in a list **`args`** for simplicity

A point object bound to **x**



Key helper functions

Now define plain Racket functions to get field, set field, call method

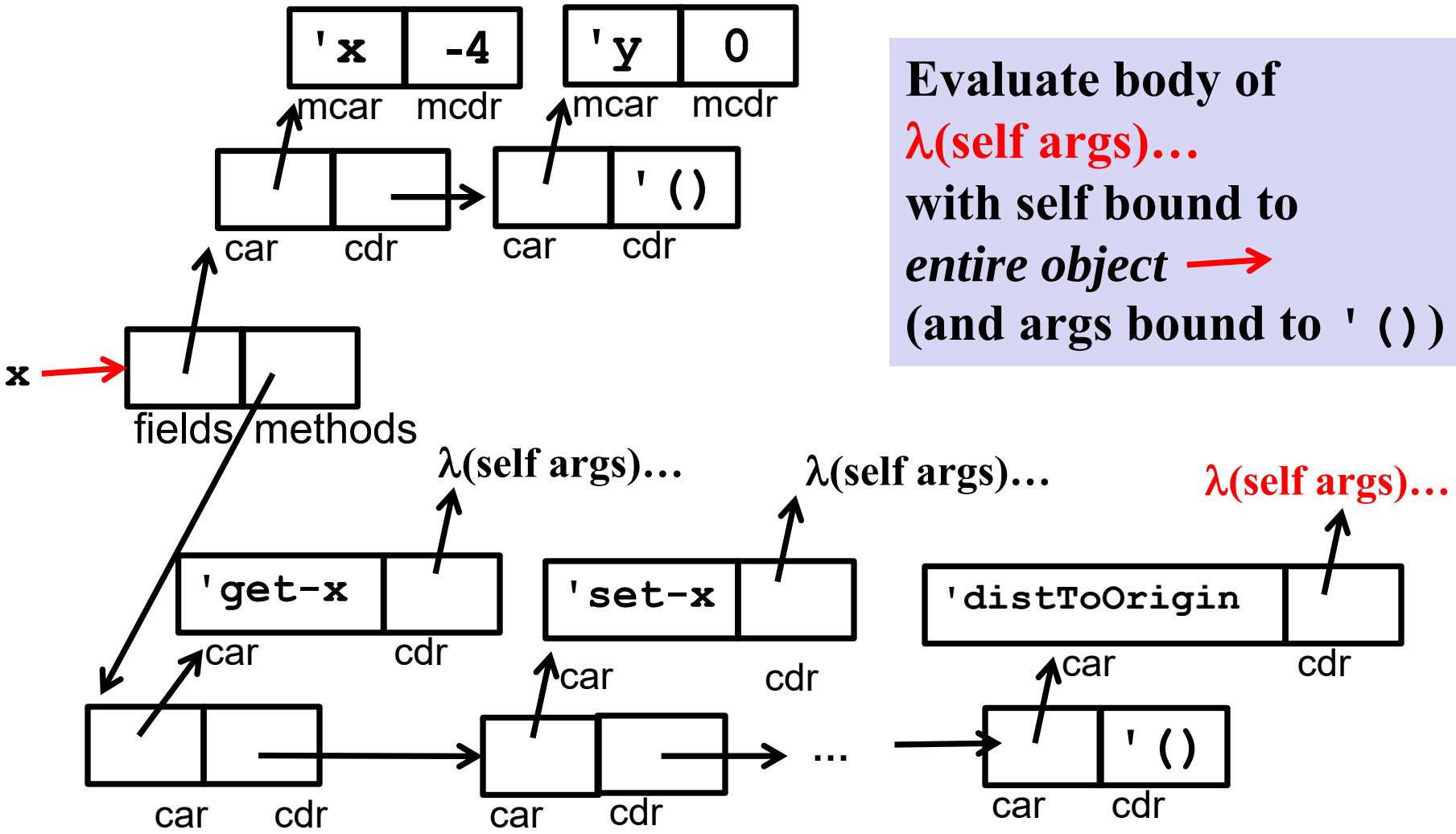
```
(define (assoc-m v xs)
   …) ; assoc for list of mutable pairs

(define (get obj fld)
   (let ([pr (assoc-m fld (obj-fields obj))]))
      (if pr (mcdr pr) (error …))))

(define (set obj fld v)
   (let ([pr (assoc-m fld (obj-fields obj))]))
      (if pr (set-mcdr! pr v) (error …))))

(define (send obj msg . args)
   (let ([pr (assoc msg (obj-methods obj))]))
      (if pr ((cdr pr) obj args) (error …))))
```

(send x 'distToOrigin)



Evaluate body of λ(self args)… with self bound to *entire object* → (and args bound to '())

Constructing points

- Plain-old Racket function can take initial field values and build a point object
  - Use functions **`get`**, **`set`**, and **`send`** on result and in "methods"
  - Call to self: **`(send self 'm …)`**
  - Method arguments in **`args`** list

```
(define (make-point _x _y)
  (obj
    (list (mcons 'x _x)
          (mcons 'y _y))
    (list (cons 'get-x (λ(self args)(get self 'x)))
          (cons 'get-y (λ(self args)(get self 'y)))
          (cons 'set-x (λ(self args)(…)))
          (cons 'set-y (λ(self args)(…)))
          (cons 'distToOrigin (λ(self args)(…))))))
```

"Subclassing"

- Can use **`make-point`** to write **`make-color-point`** or **`make-polar-point`** functions (see code)

- Build a new object using fields and methods from "super" "constructor"
  - Add new or overriding methods to the *beginning* of the list
    - **`send`** will find the first matching method
  - Since **`send`** passes the entire receiver for **`self`**, dynamic dispatch works as desired

Why not ML?

- We were wise not to try this in ML!
- ML's type system does not have subtyping for declaring a polar-point type that “is also a” point type
  - Workarounds possible (e.g., one type for all objects)
  - Still no good type for those **`self`** arguments to functions
    - Need quite sophisticated type systems to support dynamic dispatch if it is not *built into the language*
- In fairness, languages with subtyping but not generics make it analogously awkward to write generic code