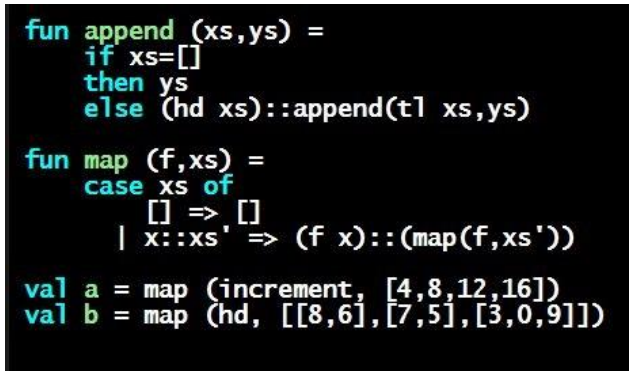


Programming Languages

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)
```

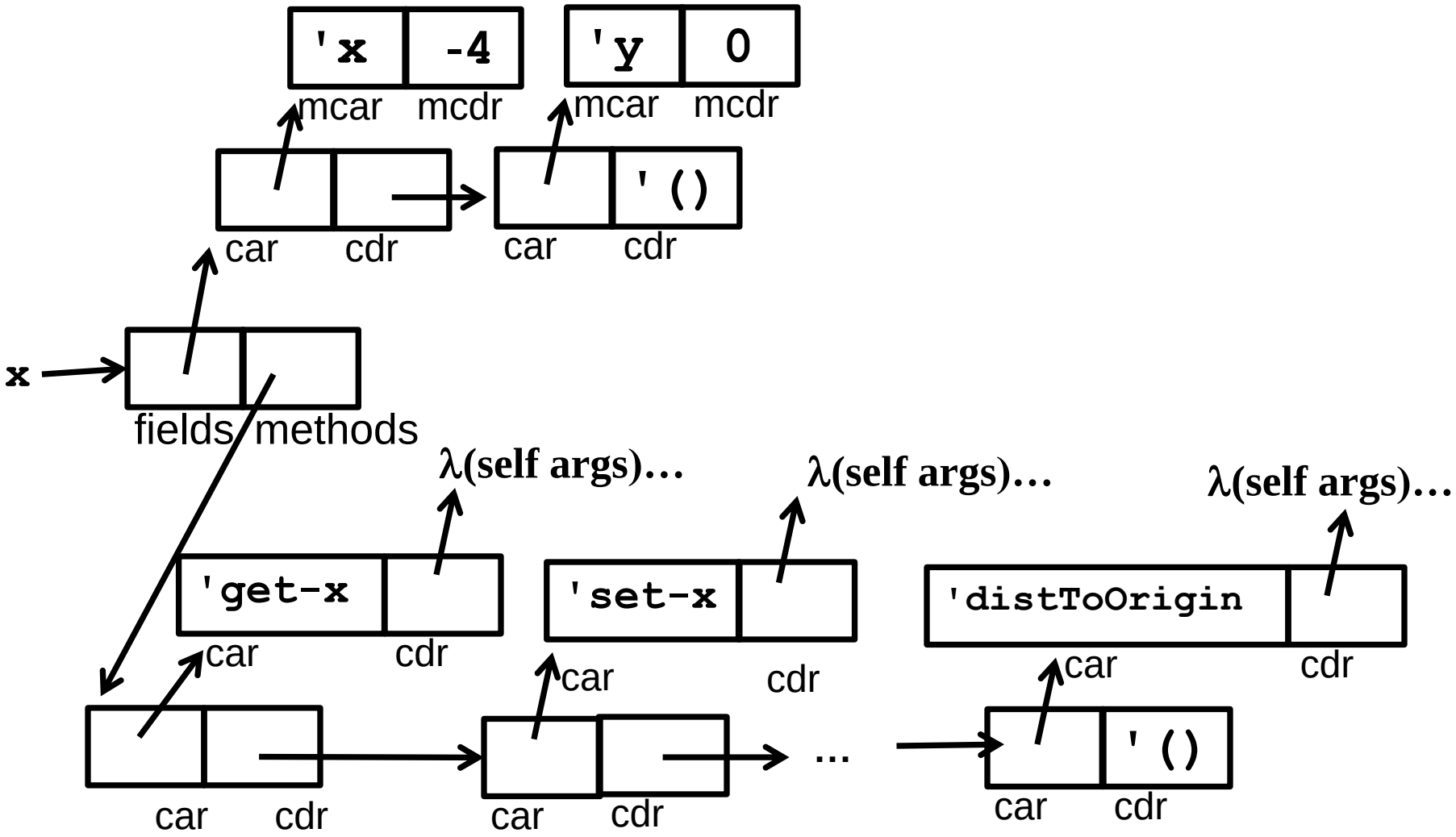
- Method-list element example:

```
(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

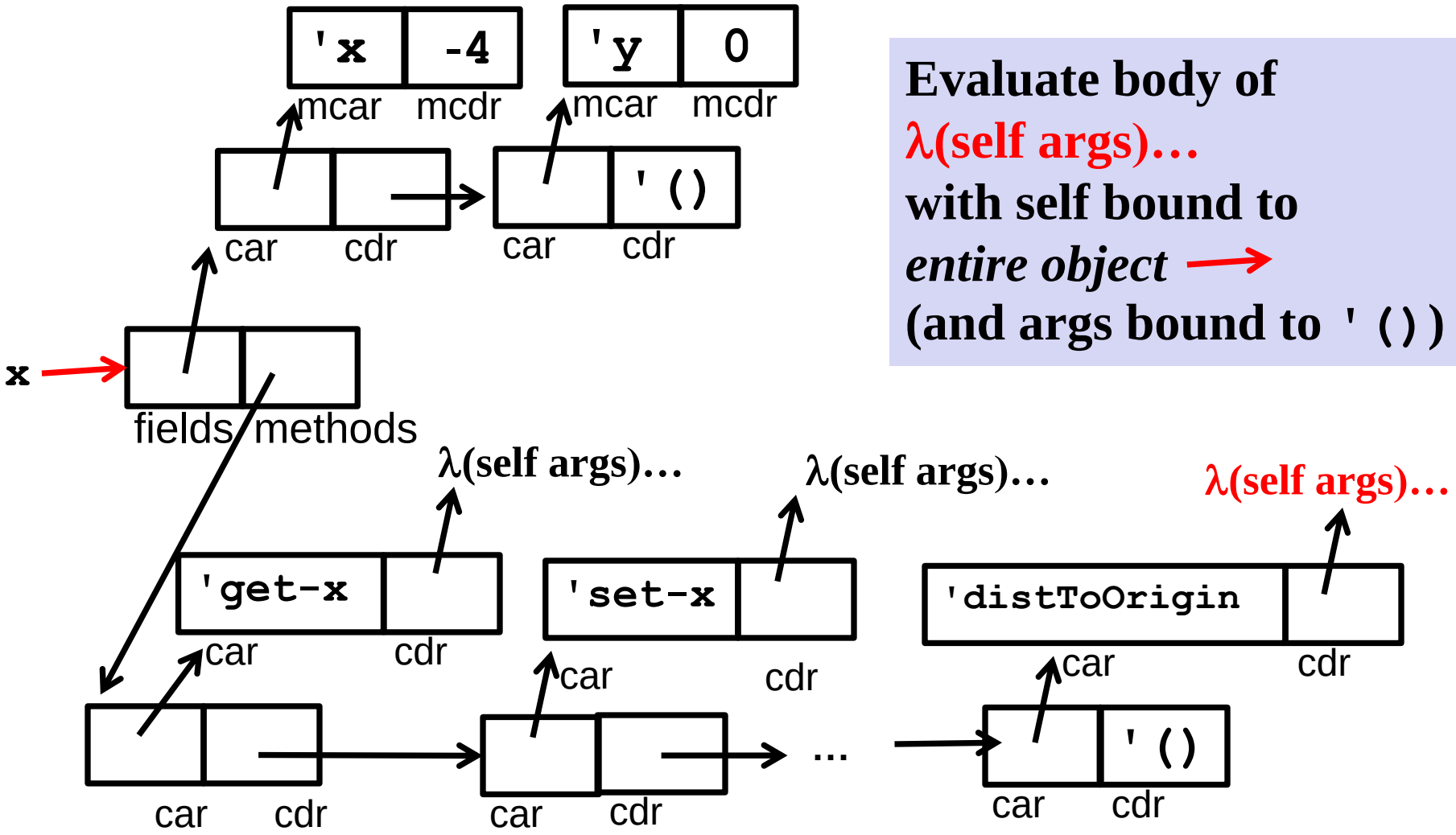
```
(define (assoc-m v xs)
   …) ; assoc for list of mutable pairs

(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