

```
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

ML Versus Racket

# *Key differences*

- Racket and ML have *much* in common
- Key differences
  - Syntax
  - Pattern-matching vs. struct-tests and accessor-functions
  - Semantics of various let-expressions
  - ...
- *Biggest* difference: ML's type system and Racket's lack thereof \*

\* There is Typed Racket, which interacts well with Racket so you can have typed and untyped modules, but we won't study it, and it differs in interesting ways from ML

# *Upcoming topics*

Coming soon:

- What is type-checking? Static typing? Dynamic typing? Etc.
- Why is type-checking approximate?
- What are the advantages and disadvantages of type-checking?

But first to better appreciate ML and Racket:

- How could a Racket programmer describe ML?
- How could an ML programmer describe Racket?

# *ML from a Racket perspective*

- Syntax, etc. aside, ML is like a well-defined **subset** of Racket
- Many of the programs it disallows have bugs ☺

```
(define (g x) (+ x x)) ; ok
(define (f y) (+ y (car y)))
(define (h z) (g (cons z 2)))
```

– In fact, in what ML allows, I never need primitives like **number**?

- But other programs it disallows I may actually want to write ☹

```
(define (f x) (if (> x 0) #t (list 1 2)))
(define xs (list 1 #t "hi"))
(define y (f (car xs)))
```

# Racket from an ML Perspective

One way to describe Racket is that it has “one big datatype”

- All values have this type

```
datatype theType = Int of int | String of string
                  | Pair of theType * theType
                  | Fun of theType -> theType
                  | ...
```

- Constructors are applied implicitly (values are *tagged*)

- 42 is really like `Int 42`

inttag	42
--------	----

- Primitives implicitly *check tags and extract data*, raising errors for wrong constructors

```
fun car v = case v of Pair(a,b) => a | _ => raise ...
fun pair? v = case v of Pair _ => true | _ => false
```

## *More on The One Type*

- Built-in constructors for “theType”: numbers, strings, booleans, pairs, symbols, procedures, etc.
- Each struct-definition creates a *new constructor*, dynamically adding to “theType”