

```
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

*Optional: eval and quote*

# *Eval*

Racket, Scheme, LISP, Javascript, Ruby, ... have **eval**

- At run-time create some data (in Racket a nested list, in Javascript a string) however you want
- Then treat the data as a program and run it
- Since we do not know ahead of time what data will be created, we need a language implementation at run-time to support **eval**
  - Could be interpreter, compiler, combination
  - But do need to “ship a language implementation” in any program containing **eval**

# `eval` in Racket

Appropriate idioms for `eval` are a matter of contention

- Often but not always there is a better way
- Programs with `eval` are harder to analyze

We will not use `eval`, but no point in leaving it mysterious

- It works on nested lists of symbols and other values
- Get advantage from concrete/abstract syntax similarity

```
(define (make-some-code y) ; just returns a list
  (if y
      (list 'begin (list 'print "hi") (list '+ 4 2))
      (list '+ 5 3)))

(eval (make-some-code #t)) ; prints "hi", result 6
```

# Quote

- Quoting (`quote ...`) or `' (...)` is a special form that makes “everything underneath” atoms and lists, not variables and calls

```
(list 'begin  
      (list 'print "hi")  
      (list '+ 4 2)) = (quote (begin  
                               (print "hi")  
                               (+ 4 2)))
```

- But then calling `eval` on it looks up symbols as code
- So `quote` and `eval` are *inverses*
- There is also *quasiquote*
  - Everything underneath is atoms and lists except if *unquoted*
  - Languages like Ruby, Python, Perl eval strings and support putting expressions inside strings, which is quasiquote