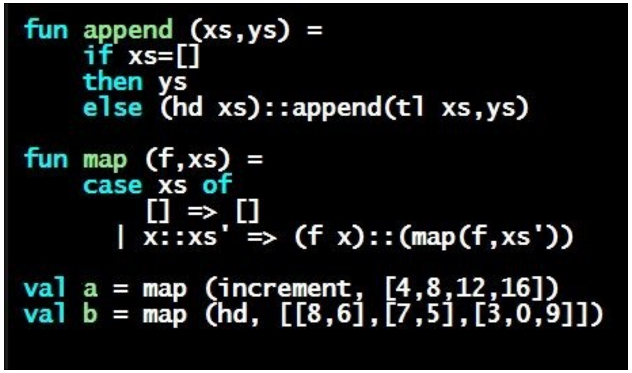

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

Implementing Variables and Environments

Dealing with variables

- Interpreters so far have been for languages without variables
  - No let-expressions, functions-with-arguments, etc.
  - Language in homework has all these things

- This segment describes in English what to do
  - Up to you to translate this to code

- Fortunately, what you have to implement is what we have been stressing since the very, very beginning of the course

Dealing with variables

- An environment is a mapping from variables (Racket strings) to values (as defined by the language)
  - Only ever put pairs of strings and values in the environment

- Evaluation takes place in an environment
  - Environment passed as argument to interpreter helper function
  - A variable expression looks up the variable in the environment
  - Most subexpressions use same environment as outer expression
  - A let-expression evaluates its body in a larger environment

The Set-up

So now a recursive helper function has all the interesting stuff:

```
(define (eval-under-env e env)
   (cond … ; case for each kind of

    ))     ; expression
```

- Recursive calls must “pass down” correct environment

Then **`eval-exp`** just calls **`eval-under-env`** with same expression and the *empty environment*

On homework, environments themselves are just Racket lists containing Racket pairs of a string (the MUPL variable name, e.g., **`"x"`**) and a MUPL value (e.g., **`(int 17)`**)

A grading detail

- Stylistically **`eval-under-env`** would be a helper function one could define locally inside **`eval-exp`**

- But do not do this on your homework
  - We have grading tests that call **`eval-under-env`** directly, so we need it at top-level