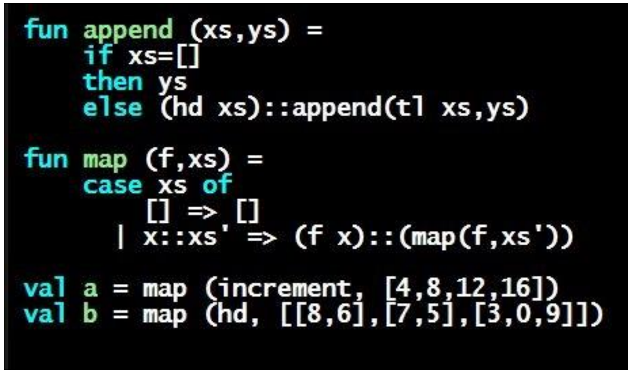

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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
University of Washington

Wrap-Up: What We Have Learned

From “About the Course”

Successful course participants will:

- Internalize an accurate understanding of what functional and object-oriented programs mean
- Develop the skills necessary to learn new programming languages quickly
- Master specific language concepts such that they can recognize them in strange guises
- Learn to evaluate the power and elegance of programming languages and their constructs
- Attain reasonable proficiency in the ML, Racket, and Ruby languages and, as a by-product, become more proficient in languages they already know

Where we’ve been…

1. Basics, functions, recursion, scope, variables, tuples, lists, …
2. Datatypes, pattern-matching, tail recursion
3. First-class functions, closures [and course motivation!]
4. Type inference, modules, equivalence

5. Dynamic types, parentheses, delayed evaluation, streams, macros
6. Structs, interpreters, closures
7. Static checking, static vs. dynamic

8. Dynamically-typed Object-Oriented Programming in Ruby
9. OOP vs. Functional decomposition, multiple inheritance, mixins, …
10. Subtyping; generics vs. subtyping

The grid, one last time

SML, Racket, and Ruby are a useful *combination* for us

	dynamically typed	statically typed
functional	Racket	SML
object-oriented	Ruby	Java/C#/Scala

ML: polymorphic types, pattern-matching, abstract types & modules

Racket: dynamic typing, “good” macros, minimalist syntax, eval

Ruby: classes but not types, very OOP, mixins

[and much more]

There is more out there, remember:

Haskell: laziness, purity, type classes, monads

Prolog: unification and backtracking

[and much more]

Benefits of No Mutation

[An incomplete list?]

1. Can freely alias or copy values/objects: Section 1
2. More functions/modules are equivalent: Section 4
3. No need to make local copies of data: Section 5
4. Depth subtyping is sound: Section 10

State updates are appropriate when you are modeling a phenomenon that is inherently state-based

Some other highlights

- Function closures are *really* powerful and convenient…
  - … and implementing them is not magic

- Datatypes and pattern-matching are really convenient…
  - … and exactly the opposite of OOP decomposition

- Sound static typing prevents certain errors…
  - … and is inherently approximate

- Subtyping and generics allow different kinds of code reuse…
  - … and combine synergistically

- Modularity is really important; languages can help

More highlights

- Programs themselves can be thought of as trees
  - Implemented via recursive interpreter

- From the very beginning: Each language construct has syntax, typing rules, and evaluation rules

- From a small set of elegant primitives with precise semantics, we have built a world that runs on software
  - Truly awe-inspiring!

What now?

- I hope to have provided a *framework* that will *remain accurate* as you continue to learn about programming languages
  - Terminology and details may change, but these concepts have all *stood the test of time* [so far??]