

```
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

Adding Operations or Variants

Extensibility

	eval	toString	hasZero	noNegConstants
Int				
Add				
Negate				
Mult				

- For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
- But beyond just style, this decision affects what (unexpected?) software *extensions* need not change old code
- **Functions** [see ML code]:
 - Easy to add a new operation, e.g., **noNegConstants**
 - Adding a new variant, e.g., **Mult** requires modifying old functions, but ML type-checker gives a to-do list if original code avoided wildcard patterns

Extensibility

	<code>eval</code>	<code>toString</code>	<code>hasZero</code>	<code>noNegConstants</code>
<code>Int</code>				
<code>Add</code>				
<code>Negate</code>				
<code>Mult</code>				

- For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
- But beyond just style, this decision affects what (unexpected?) software *extensions* are easy and/or do not change old code
- **Objects** [see Ruby code]:
 - Easy to add a new variant, e.g., `Mult`
 - Adding a new operation, e.g., `noNegConstants` requires modifying old classes, but **[optional:]** Java type-checker gives a to-do list if original code avoided default methods

The other way is possible

- Functions allow new operations and objects allow new variants without modifying existing code *even if they didn't plan for it*
 - Natural result of the decomposition

Optional:

- Functions can support new variants somewhat awkwardly “if they plan ahead”
 - *Not explained here: Can use type constructors to make datatypes extensible and have operations take function arguments to give results for the extensions*
- Objects can support new operations somewhat awkwardly “if they plan ahead”
 - *Not explained here: The popular Visitor Pattern uses the double-dispatch pattern to allow new operations “on the side”*

Thoughts on Extensibility

- Making software extensible is valuable and hard
 - If you know you want new operations, use FP
 - If you know you want new variants, use OOP
 - If both? Languages like Scala try; it's a hard problem
 - Reality: The future is often hard to predict!
- Extensibility is a double-edged sword
 - Code more reusable without being changed later
 - But makes original code more difficult to reason about locally or change later (could break extensions)
 - Often language mechanisms to make code *less* extensible (ML modules hide datatypes; Java's **final** prevents subclassing/overriding)