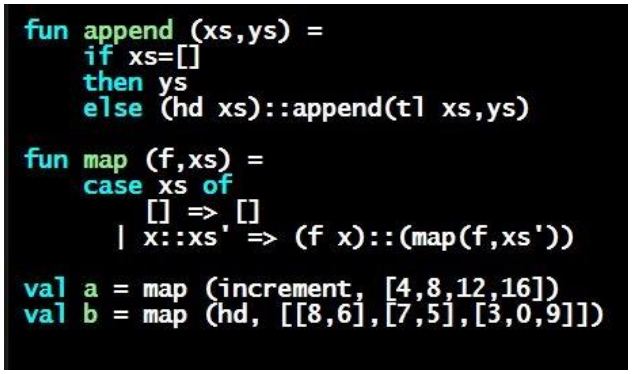

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

OOP vs. Functional Decomposition

Breaking things down

- In functional (and procedural) programming, break programs down into functions that perform some operation

- In object-oriented programming, break programs down into classes that give behavior to some kind of data

Beginning of this unit:

  - These two forms of *decomposition* are so exactly opposite that they are two ways of looking at the same “matrix”

  - Which form is “better” is somewhat personal taste, but also depends on how you expect to *change/extend software*

  - For some operations over two (multiple) arguments, functions and pattern-matching are straightforward, but with OOP we can do it with *double dispatch* (multiple dispatch)

The expression example

Well-known and compelling example of a common *pattern*:

- Expressions for a small language
- Different variants of expressions: ints, additions, negations, …
- Different operations to perform: **`eval`**, **`toString`**, **`hasZero`**, …

Leads to a matrix (2D-grid) of variants and operations

- Implementation will involve deciding what "should happen" for each entry in the grid *regardless of the PL*

	`eval`	**`toString`**	**`hasZero`**	…
`Int`				
`Add`				
`Negate`				
…				

Standard approach in ML

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Define a *datatype*, with one *constructor* for each variant
  - (No need to indicate datatypes if dynamically typed)
- "Fill out the grid" via one function per column
  - Each function has one branch for each column entry
  - Can combine cases (e.g., with wildcard patterns) if multiple entries in column are the same

[See the ML code]

Standard approach in OOP

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- Define a *class*, with one *abstract method* for each operation
  - (No need to indicate abstract methods if dynamically typed)
- Define a *subclass* for each variant
- So “fill out the grid” via one class per row with one method implementation for each grid position
  - Can use a method in the superclass if there is a default for multiple entries in a column

[See the Ruby code] [*Optional*: See the Java code]

A big course punchline

	`eval`	`toString`	`hasZero`	…
`Int`				
`Add`				
`Negate`				
…				

- FP and OOP often doing the same thing in *exact* opposite way
  - Organize the program “by rows” or “by columns”

- Which is “most natural” may depend on what you are doing (e.g., an interpreter vs. a GUI) or personal taste

- Code layout is important, but there is no perfect way since software has many dimensions of structure
  - Tools, IDEs can help with multiple “views” (e.g., rows / columns)