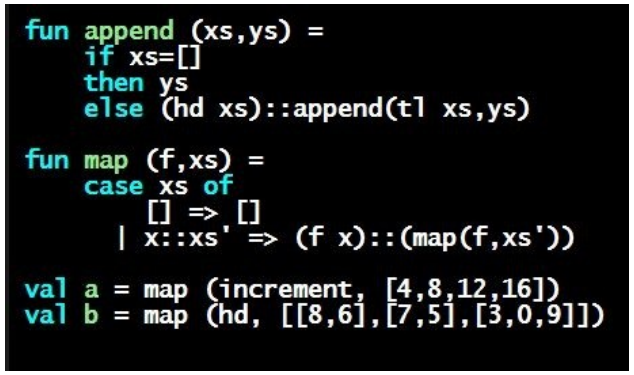

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

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

The Truth About **`cons`**

The truth about **`cons`**

`cons` just makes a pair

- Often called a *cons cell*
- By convention and standard library, lists are nested pairs that eventually end with **`null`**

```
(define pr (cons 1 (cons #t "hi"))) ; '(1 #t . "hi")
(define lst (cons 1 (cons #t (cons "hi" null))))
(define hi (cdr (cdr pr)))
(define hi-again (car (cdr (cdr lst))))
(define hi-another (caddr lst))
(define no (list? pr))
(define yes (pair? pr))
(define of-course (and (list? lst) (pair? lst)))
```

Passing an *improper list* to functions like **`length`** is a run-time error

The truth about **`cons`**

So why allow improper lists?

- Pairs are useful
- Without static types, why distinguish **`(e1,e2)`** and **`e1::e2`**

Style:

- Use proper lists for collections of unknown size
- But feel free to use **`cons`** to build a pair
  - Though structs (like records) may be better

Built-in primitives:

- **`list?`** returns true for proper lists, including the empty list
- **`pair?`** returns true for things made by cons
  - All improper and proper lists except the empty list