```
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

Let Expressions

Review

Huge progress already on the core pieces of ML:

- Types: **`int bool unit t1*…*tn t list t1*…*tn->t`**
  - Types "nest" (each **`t`** above can be itself a compound type)
- Variables, environments, and basic expressions
- Functions
  - Build: **`fun x0 (x1:t1, …, xn:tn) = e`**
  - Use: **`e0 (e1, …, en)`**
- Tuples
  - Build: **`(e1, …, en)`**
  - Use: **`#1 e, #2 e, …`**
- Lists
  - Build: **`[] e1::e2`**
  - Use: **`null e hd e tl e`**

Now…

The big thing we need: local bindings

- For style and convenience

This segment:

- Basic let-expressions

Next segments:

- A big but natural idea: nested function bindings
- For efficiency (***not*** "just a little faster")

The construct to introduce local bindings is ***just an expression***, so we can use it anywhere an expression can go

Let-expressions

3 questions:

- Syntax: **`let b1 b2 … bn in e end`**
  - Each ***`bi`*** is any *binding* and ***`e`*** is any *expression*

- Type-checking: Type-check each ***`bi`*** and ***`e`*** in a static environment that includes the previous bindings.
  Type of whole let-expression is the type of ***`e`***.

- Evaluation: Evaluate each ***`bi`*** and ***`e`*** in a dynamic environment that includes the previous bindings.
  Result of whole let-expression is result of evaluating ***`e`***.

Silly examples

```
fun silly1 (z : int) =
    let val x = if z > 0 then z else 34
        val y = x+z+9
    in
        if x > y then x*2 else y*y
    end
fun silly2 () =
    let val x = 1
    in
        (let val x = 2 in x+1 end) +
        (let val y = x+2 in y+1 end)
    end
```

`silly2` is poor style but shows let-expressions are expressions

- Can also use them in function-call arguments, if branches, etc.
- Also notice shadowing

What's new

- What's new is ***scope***: where a binding is in the environment
  - *In* later bindings and body of the let-expression
    - (Unless a later or nested binding shadows it)
  - *Only in* later bindings and body of the let-expression

- *Nothing else is new:*
  - Can put any binding we want, even function bindings
  - Type-check and evaluate just like at "top-level"