```
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

Another Closure Idiom: Callbacks

Callbacks

A common idiom: Library takes functions to apply later, when an *event* occurs – examples:

- – When a key is pressed, mouse moves, data arrives
- – When the program enters some state (e.g., turns in a game)

A library may accept multiple callbacks

- – Different callbacks may need different private data with different types
- – Fortunately, a function's type does not include the types of bindings in its environment
- – (In OOP, objects and private fields are used similarly, e.g., Java Swing's event-listeners)

Mutable state

While it's not absolutely necessary, mutable state is reasonably appropriate here

– We really do want the "callbacks registered" to *change* when a function to register a callback is called

Example call-back library

Library maintains mutable state for "what callbacks are there" and provides a function for accepting new ones

– A real library would all support removing them, etc.

– In example, callbacks have type **int->unit**

So the entire public library interface would be the function for registering new callbacks:

```
val onKeyEvent : (int -> unit) -> unit
```

(Because callbacks are executed for side-effect, they may also need mutable state)

Library implementation

```
val cbs : (int -> unit) list ref = ref []

fun onKeyEvent f =  cbs := f :: (!cbs)

fun onEvent i =
    let fun loop fs =
            case fs of
               []      => ()
             | f::fs' => (f i; loop fs')
    in loop (!cbs) end
```

Clients

Can only register an `int -> unit`, so if any other data is needed, must be in closure's environment

– And if need to "remember" something, need mutable state

Examples:

```
val timesPressed = ref 0
val _ = onKeyEvent (fn _ =>
            timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
    onKeyEvent (fn j =>
        if i=j
        then print ("pressed " ^ Int.toString i)
        else ())
```