Restricted Mutable State
More idioms

• Pass functions with private data to iterators: Done
• Combine functions (e.g., composition): Done
• Currying (multi-arg functions and partial application): Done
• Callbacks (e.g., in reactive programming)
ML has (restricted) mutation

• Mutable data structures are okay in some situations
  – When “update to state of world” is appropriate model
  – But want most language constructs truly immutable

• ML does this with a separate construct: references

• Do not use references on your homework.
References

• New types: \( t \, \text{ref} \) where \( t \) is a type

• New expressions:
  – \( \text{ref e} \) to create a reference with initial contents from result of \( e \)
  – \( e1 := e2 \) to update contents
  – \( !e \) to retrieve contents (not negation)
References example

```haskell
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!y) + (!z) (* 85 *)
(* x + 1 does not type-check *)
```

- A **variable** bound to a reference (e.g., x) is still **immutable**: it will always refer to the same reference
- **Contents** of the reference may change via :=
- There may be **aliases** to the reference, which matter a lot
- References are **first-class** values
- Like a one-field mutable object. := and ! don’t specify field
Callback idiom

Library takes function to apply later, when an event occurs.
Library interface:

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

Other 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 need different private data with different types
- Callback function’s type does not include the types of bindings in its environment!
Library implementation

Mutable state not absolutely necessary, but is reasonably appropriate.

```ocaml
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
```

Create new ref cell with initial contents []

Get contents of ref cell.

Set contents of ref cell.

Sequencing expression;
Evaluate left side and throw away result, then evaluate right side and use result.
Clients

Closure’s environment captures any necessary context, possibly including mutable state for "remembering" history.

```plaintext
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 ()
fun makeCounterCallback k =
  let count = ref 0 in
  onKeyEvent (fn i => if i=k
              then count := !count + 1
              else ());
  count
end
```