# Imperative Programming

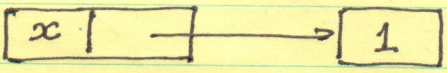
A command changes the state of the system.

A new data type : mutable storage.
We can now issue a command to update a storage cell.

Commands are special expressions.
We need a special "dummy" value

```
() : unit
↓      ↓
value  type
```

```
let x = ref 1
```

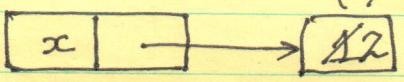



```
val x : int ref = {contents = 1}
```

A record with one mutable field = cell
or a location

ref plays 2 roles
(1) it is a type constructor
(2) it is also a value constructor
e.g. ref 1

To update `x := 2;;`
`() : unit`




To see what is inside a cell

```
!x
- : int = 2
```

$x := exp$ $\quad$ $x$ $\alpha$-ref
$exp : \alpha$

(I) evaluate $x$ & make sure it is a location
(II) evaluate exp to produce a value $v$
(III) store $v$ in the location pointed to by $x$

Information has been thrown away.
The programmer now has responsibility for the lifetime of data.
Time has appeared in the picture
Sequence of commands ;

—x—

NOT in OCAML but in other languages:

$x := x+1$

The symbol $x$ means 2 different things
on the LHS it stands for a location
on the RHS it stands for a value

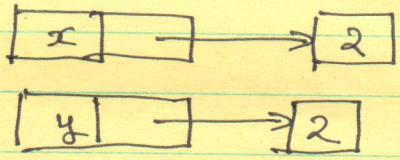
In OCAML $x$ always means a location

$x := !x+1$

—x—

Equality

let $y$ = ref 2
$x = y$ → do these cells store
true the same value?



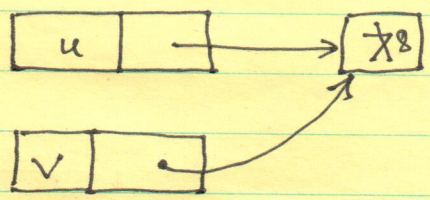
```
y := 3
x = y
- false
```

== do these names stand for the same location?

```
let u = ref 7 ;;
let v = ref 7;;
!u;;
7
!v
7
u=v
true
v:= 8
u = v
false
```

---

```
let u = ref 7
let v = u
```

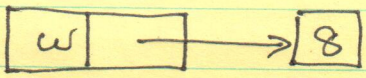



aliasing

2 names for one cell

```
!u
7
!v
7
u=v
true
v:=8
```

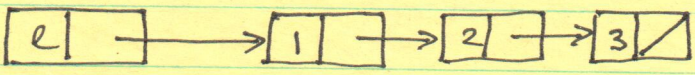
```
!u
8
u=v
true
let w= ref 8
u=w
true
```




```
u == w
false
u == v
true
```

```
u := u+1
type error
```

```
let l = ref [1;2;3]
```




```
l := 0::(!l)
```

! : the dereference operator

Records with multiple mutable fields

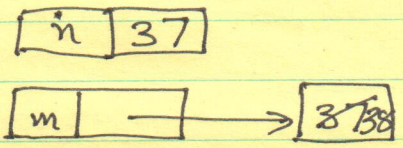
```
type point = {mutable x:int, mutable y:int
let p = {x=3; y=4}
  p.x   automatic ~~def~~ deref
let move (p:point) (a,b) =
    (p.x <- p.x+a);(p.y <- p.y+b)
      <- alternative syntax
        for updating
```

---

Interaction between mutation & functions

```
let modify n = (n := !n+1)
    int ref -> unit
 !x
 17
 modify x;;
 ()
 !x;;
 18
```

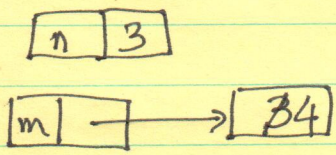
```
let incUpdate (n:int) =
    let m = ref n in (m:= !m+1);;
        int -> unit

let q = 37;;
incUpdate q;;
();
q;;
37
```




incUpdate is absolutely useless!

To see the intermediate changes we need to insert print statements.

```
let mash (n: int) =
    let m = ref n in
      (Printf.printf "m is %i " !m);
      (m:= !m+1);
      (Print . . . .  n);
      (Print - - - . .  !m)
mash 3;;
```




```
m is 3
n is 3
m is 4
```

on exit both these frames are discarded.

```
let foo n =
    let x = ref n in
    while (!x <10) do
        (Print... !x);
        (x := !x+1)
    done
```

foo 1

; is sequencing

- Memory can now be updated
- The programmer is responsible for lifetime of data
- Time enters the computation picture
- commands appear as a new semantic category