- HW1

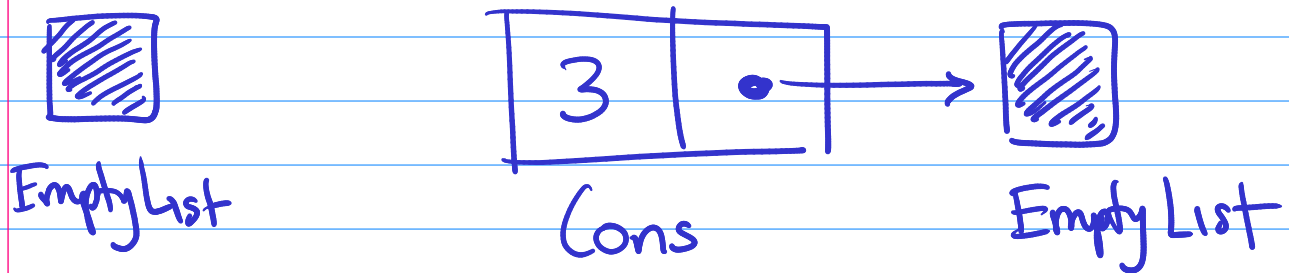
- Maps

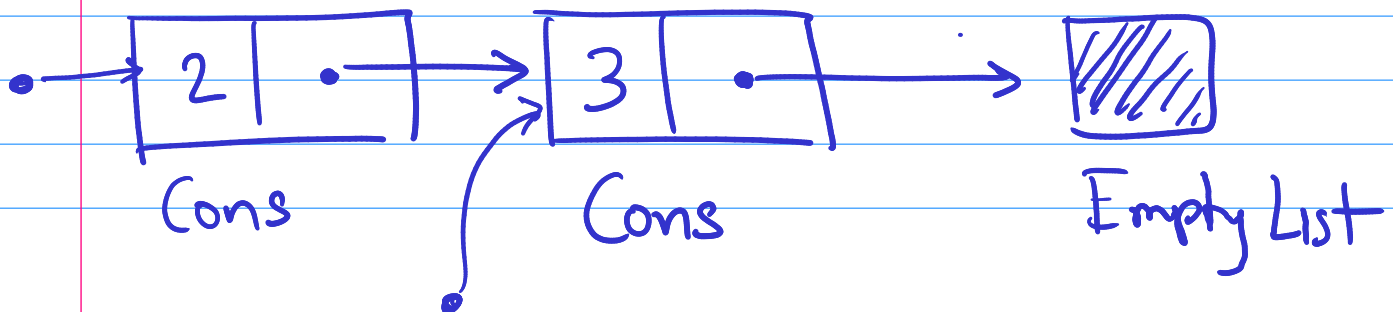
- Recursive Data Types

```
type status = Connected | Disconnected
```

"Closed-world assumption"

```
type ourlist = EmptyList
             | Cons of int * ourlist
```





Computing the length of a list

```
let rec ourlength l =
  match l with
  | EL -> 0
  | Cons(_, tl) -> 1 + ourlength tl
```

rec ← Adds "ourlength" to the scope during definition.

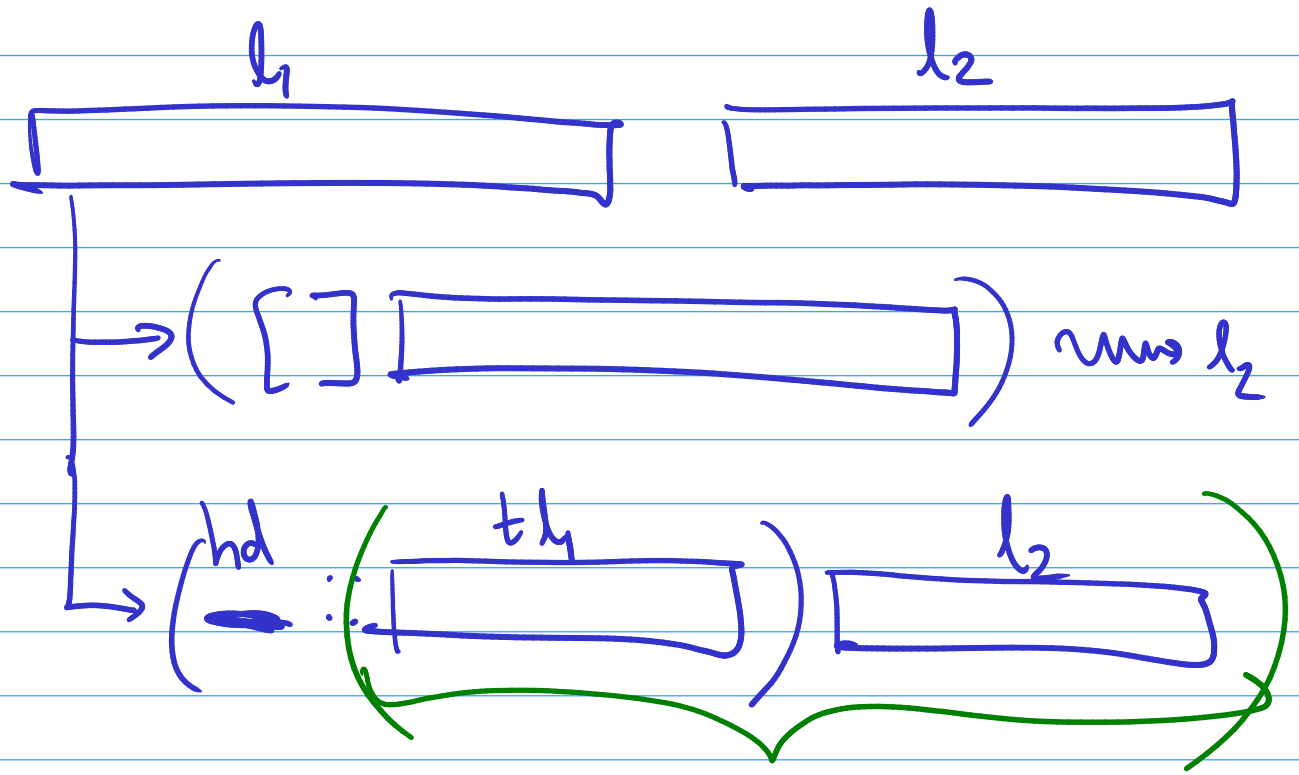
Finding if a number is even

```
let rec isEven n = (n = 0) || isOdd (n - 1)
and
isOdd n = if n = 0 then false else isEven (n - 1);;
```

"and" ← defines mutual recursion.

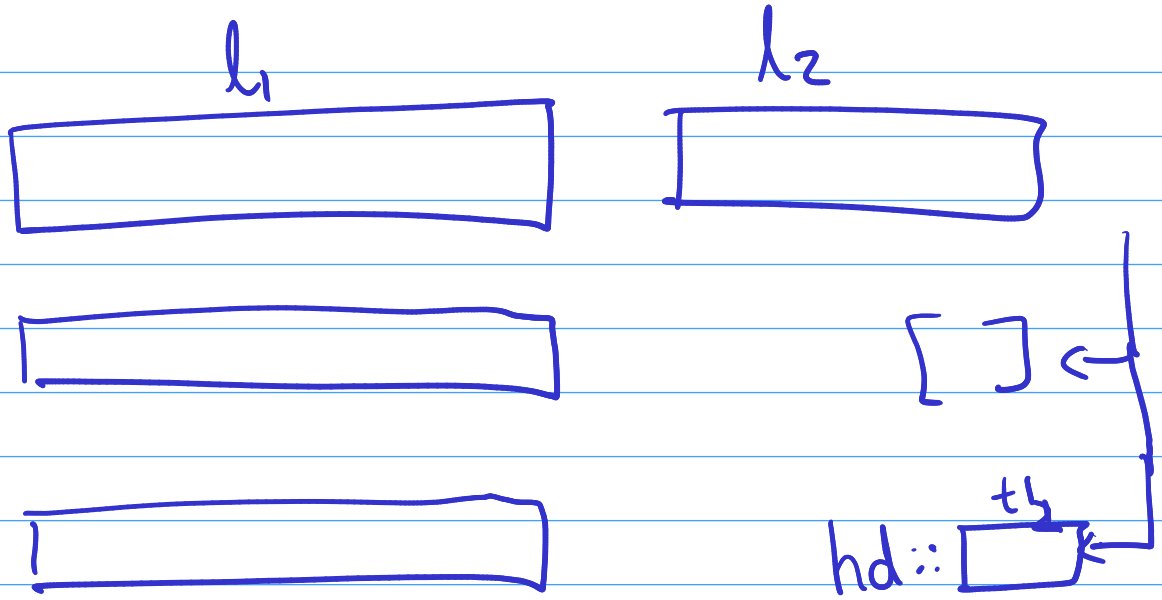
Careful about the base case.

Concatenating lists l_1 l_2




```
let rec concat (l1 : ourlist) (l2 : ourlist) : ourlist =
  match l1 with
  | EmptyList -> l2
  | Cons(hd1, tl1) -> Cons(hd1, concat tl1 l2);;
```

Terminates because first argument is structurally decreasing.


l1
l2
[ ]
hd ::
tl

Merge sorted arrays

```
let rec merge l1 l2 =
  match l1, l2 with
  | [], [] -> []
  | [], hd2 :: tl2 -> l2
  | hd1 :: tl1, [] -> l1
  | hd1 :: tl1, hd2 :: tl2 -> if hd1 < hd2
                              then hd1 :: merge tl1 l2
                              else hd2 :: merge l1 tl2
```

If one list is empty, then return the other.

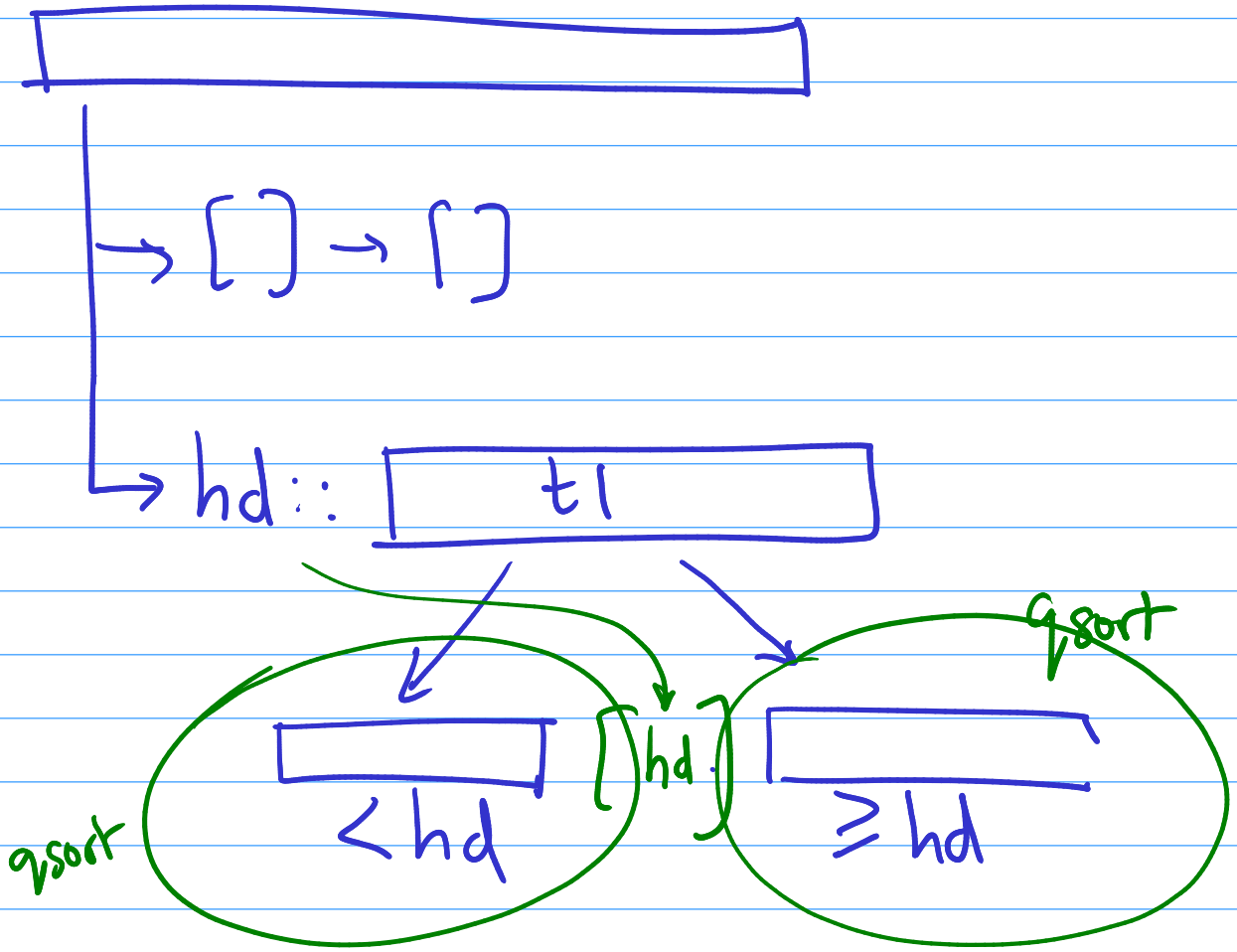
Compare hd_1 & hd_2.

Decide which goes first

Recursively merge the rest.

Total length of l_1 & l_2 goes down in each recursive call. So it terminates.

Sorting an array (QuickSort)




```
let rec qsort l =
  match l with
  | [] -> []
  | hd :: tl -> let lessThanHd = List.filter ((>) hd) tl in
                let biggerThanHd = List.filter ((<=) hd) tl in
                let sl = qsort lessThanHd in
                let sb = qsort biggerThanHd in
                sl @ [ hd ] @ sb;;
```