# OCAML
# Programming

```
# let rec removeDuplicate list =
    match list with
    | [] -> []
    | [hd] -> [hd]
    | hd :: hd' :: tl ->
      if hd = hd' then removeDuplicate (hd' :: tl)
      else hd :: removeDuplicate (hd' :: tl)
  ;;
val removeDuplicate: 'a list -> 'a list = <fun>

# removeDuplicate [1;1;2;3;4;5;5;5];;
- : int list = [1; 2; 3; 4; 5]
```

```
# let rec removeDuplicate list =
    match list with
    | [] -> []
    | [hd] -> [hd]
    | hd :: hd' :: tl ->
      if hd = hd' then removeDuplicate (hd' :: tl)
      else hd :: removeDuplicate (hd' :: tl)
  ;;
val removeDuplicate: 'a list -> 'a list = <fun>

# removeDuplicate [1;1;2;3;4;5;5;5];;
- : int list = [1; 2; 3; 4; 5]
```

*a new list element*
*Is allocated*

```
 let rec removeDuplicate = function
   | [] as l -> l
   | [_] as l -> l
   | hd :: (hd' :: _ as tl) ->
     if hd = hd' then removeDuplicate tl
     else hd :: removeDuplicate tl
  ;;
val removeDuplicate: 'a list -> 'a list = <fun>
```

the **as** pattern allows us to declare
a name for the thing matched

```
 let rec removeDuplicate = function
   | [] | [_] as l -> l
   | hd :: (hd' :: _ as tl) ->
     if hd = hd' then removeDuplicate tl
     else hd :: removeDuplicate tl
  ;;
val removeDuplicate: 'a list -> 'a list = <fun>
```

the **or** pattern allows us to combine the first two cases into one

```
 let rec removeDuplicate = function
   | [] | [_] as l -> l
   | hd :: (hd' :: _ as tl) when hd = hd' -> removeDuplicate tl
   | hd :: tl -> hd :: removeDuplicate tl  ;;
val removeDuplicate: 'a list -> 'a list = <fun>
```

**when** clause allows us to add an
extra precondition to a pattern

# Equality

```
# 3 = 4;;
- : bool = false
# [3;4;5] = [3;4;5];;
- : bool = true
# [Some 3; None] = [None; Some 3];;
- : bool = false
```

# Polymorphic equality

```
# (=);;
- : 'a -> 'a -> bool = <fun>
```

**OCaml's polymorphic comparison functions are built into the runtime to a low level**
**paying attention only to the structure of the values as they're laid out in memory.**

# But .....

```
# (fun x -> x + 1) = (fun x -> x + 1);;
Exception: (Invalid_argument "equal: functional value").
```

# Example

Given two indices, i and k, the slice is the list containing the elements between the i'th and k'th element of the original list (both limits included).

# Solution

```
# let slice list i k =
    let rec take n = function
      | [] -> []
      | h :: t -> if n = 0 then [] else h :: take (n-1) t
    in
    let rec drop n = function
      | [] -> []
      | h :: t as l -> if n = 0 then l else drop (n-1) t
    in
    take (k - i + 1) (drop i list);;
val slice : 'a list -> int -> int -> 'a list = <fun>
```

# Example

Drop every n'th element from a list.

# Solution

```
# let drop list n =
    let rec aux i = function
      | [] -> []
      | h :: t -> if i = n then aux 1 t else h :: aux (i+1) t  in
    aux 1 list;;
val drop : 'a list -> int -> 'a list = <fun>
```