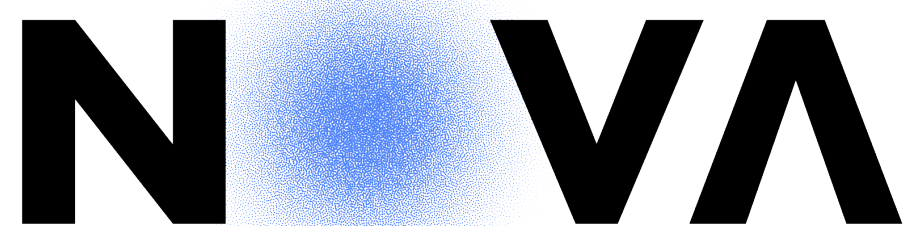


# Linguagens e Ambientes de Programação (Aula Teórica 8)

**LEI - Licenciatura em Engenharia Informática**

**João Costa Seco ([joao.seco@fct.unl.pt](mailto:joao.seco@fct.unl.pt))**



NOVA SCHOOL OF  
SCIENCE & TECHNOLOGY

# Agenda

---

- Tipos compostos: Listas
- Funções recursivas sobre listas

# Listas primitivas em OCaml

- Listas são coleções homogêneas e imutáveis de valores.
- São coleções heterogêneas seguras, com tipos soma.
- Podem ser criadas por literais
- Podem ser criadas pelo operador `::`

```
[50] ✓ 0.0s
... - : int list = [1; 2; 3; 4]

[51] ✓ 0.0s
... - : 'a list = []
```

```
[52] ✓ 0.0s
... - : int list = [1; 2; 3; 4]

[53] ✓ 0.0s
... - : int list = [1; 2; 3; 4]

[54] ✓ 0.0s
... - : int list = [1; 2; 3; 4]
```

- O tipo lista é paramétrico no tipo dos seus elementos.

# Listas primitivas em OCaml

- Listas são coleções homogêneas e imutáveis de valores de qualquer tipo.

```
▷ [[[]; []]; [[1; 2]; [3; 4]]; [[]]]  
[60] ✓ 0.0s  
... - : int list list list list = [[[]; []]; [[1; 2]; [3; 4]]; [[]]]
```

```
let points = [(1.0, 2.0); (3.0, 4.0); (5.0, 6.0)] in Polygon points  
[57] ✓ 0.0s  
... - : figure = Polygon [(1., 2.); (3., 4.); (5., 6.)]
```

```
▷ let polygons = [Polygon [(1.0, 2.0); (3.0, 4.0); (5.0, 6.0)]; Polygon [(1.0, 2.0); (3.0, 4.0); (5.0, 6.0); (7.0, 8.0)]]  
[58] ✓ 0.0s OCaml  
... val polygons : figure list =  
  Polygon [(1., 2.); (3., 4.); (5., 6.)];  
  Polygon [(1., 2.); (3., 4.); (5., 6.); (7., 8.)]
```

# Tipo lista

- O tipo lista é um tipo indutivo e paramétrico com dois casos
- **Nil** ou lista vazia (`[]`) onde o tipo dos elementos ainda não está determinado.
- **Cons** com cabeça e cauda (`x :: l`) onde o elemento `x` é do tipo dos elementos da lista `l`

```
[62]  ✓  0.0s  
      []  
...   - : 'a list = []
```

```
[63]  ✓  0.0s  
      1 :: []  
...   - : int list = [1]
```

```
[65]  ✓  0.0s  
      fun x -> x :: []  
...   - : 'a -> 'a list = <fun>
```

# Eliminação do Tipo lista

- O tipo lista é um tipo *indutivo* e paramétrico com dois casos, que podem ser analisados com pattern matching.

```
let is_empty l =  
  match l with  
  | [] -> true  
  | _ -> false
```

[77] ✓ 0.0s

... val is\_empty : 'a list -> bool = <fun>

```
let rec sum l =  
  match l with  
  | [] -> 0  
  | h::t -> h + sum t
```

[75] ✓ 0.0s

... val sum : int list -> int = <fun>

```
let rec length l =  
  match l with  
  | [] -> 0  
  | _::t -> 1 + length t
```

[76] ✓ 0.0s

... val length : 'a list -> int = <fun>

# Eliminação do Tipo lista

- O tipo lista é um tipo *indutivo* e paramétrico com dois casos, que podem ser analisados com pattern matching.

```
▷ (** [max l] is the maximum of the numbers in the list
   | | pre: [length l > 0] *)
let rec max l =
  match l with
  | [] →
  | h::t →

[11] ✓ 0.0s

... val max : 'a list → 'a = <fun>
```

# Eliminação do Tipo lista

- O tipo lista é um tipo *indutivo* e paramétrico com dois casos, que podem ser analisados com pattern matching.

```
▷ (** [max l] is the maximum of the numbers in the list
   | | pre: [length l > 0] *)
let rec max l =
  match l with
  | [] → assert false
  | [ x ] → x
  | head::tail → let m = max tail in if head > m then head else m

[11] ✓ 0.0s

... val max : 'a list → 'a = <fun>
```



# Eliminação do Tipo lista

- O tipo lista é um tipo *indutivo* e paramétrico com dois casos, que podem ser analisados com pattern matching.

```
▷ (** [max l] is the maximum of the numbers in the list
   | | pre: [length l > 0] *)
let rec max = function
  | [] → assert false
  | [ x ] → x
  | head::tail → let m = max tail in if head > m then head else m

[10] ✓ 0.0s

... val max : 'a list → 'a = <fun>
```

Neste caso, estamos a usar um acelerador que facilita a declaração de funções recursivas com pattern matching.

# Listas são valores imutáveis

- “Modificar” listas faz-se criando novas listas.

```
let l = [1; 2; 3; 4; 5];;  
let l0 = 0::l;;  
let n = length l;;
```

[85] ✓ 0.0s

```
... val l : int list = [1; 2; 3; 4; 5]
```

```
... val l0 : int list = [0; 1; 2; 3; 4; 5]
```

```
... val n : int = 5
```

# Listas são valores imutáveis

- “Modificar” listas faz-se criando novas listas.

```
▷ let rec string_of_list f l =  
  match l with  
  | [] → "[]"  
  | head::tail → f head ^ " :: " ^ string_of_list f tail;;  
  
[18] ✓ 0.0s  
... val string_of_list : ('a → string) → 'a list → string = <fun>
```

# Listas são valores imutáveis

- “Modificar” listas faz-se criando novas listas.

```
▷ let rec string_of_list f l =
  match l with
  | [] → "[]"
  | head::tail → f head ^ " :: " ^ string_of_list f tail;;

let l = [1; 2; 3; 4; 5] in
let l0 = 0::l in
print_string (string_of_list string_of_int l0); print_newline ();
print_string (string_of_list string_of_int l); print_newline ()

[18] ✓ 0.0s

... val string_of_list : ('a → string) → 'a list → string = <fun>

... 0 :: 1 :: 2 :: 3 :: 4 :: 5 :: []
... 1 :: 2 :: 3 :: 4 :: 5 :: []

... - : unit = ()
```

# Listas são valores imutáveis

- “Modificar” listas faz-se criando novas listas.

```
let rec append l1 l2 =  
  match l1 with  
  | [] → l2  
  | h::t → h::(append t l2);;
```

```
let l1 = [1; 2; 3] in  
let l2 = [4; 5; 6] in  
append l1 l2
```

[21] ✓ 0.0s

... val append : 'a list → 'a list → 'a list = <fun>

... - : int list = [1; 2; 3; 4; 5; 6]

# Iteração em listas

- Quando queremos transformar uma lista através de uma função, usamos a função `map`.
- a função `map` já está definida no módulo `List`.

```
let rec map f l =  
  match l with  
  | [] → []  
  | h::t → f h :: map f t;;
```

[35] ✓ 0.0s

... val map : ('a → 'b) → 'a list → 'b list = <fun>

```
▷ map (fun x → x + 1) [1; 2; 3; 4; 5];;
```

[37] ✓ 0.0s

... - : int list = [2; 3; 4; 5; 6]

```
▷ map (fun x → x * 2) [1; 2; 3; 4; 5];;
```

[38] ✓ 0.0s

... - : int list = [2; 4; 6; 8; 10]

# Iteração em listas

- Quando queremos analisar uma lista e produzir um único resultado, usamos a função `fold`. (`fold_left`)
- a função `fold_left` também já está definida no módulo `List`.

```
let rec fold_left f acc l =
  match l with
  | [] → acc
  | h::t → fold_left f (f acc h) t;;
[43] ✓ 0.0s
... val fold_left : ('a → 'b → 'a) → 'a → 'b list → 'a = <fun>

▷ fold_left (fun x y → x + y) 0 [1; 2; 3; 4; 5];;
[45] ✓ 0.0s
... - : int = 15

▷ fold_left (fun acc n → acc ^ string_of_int n) "" [1; 2; 3; 4; 5];;
[47] ✓ 0.0s
... - : string = "12345"
```

# Iteração em listas

- Quando queremos analisar uma lista e produzir um único resultado, usamos a função `fold`. (`fold_right`)
- a função `fold_right` também já está definida no módulo `List`.

```
let rec fold_right f l acc =  
  match l with  
  | [] → acc  
  | h::t → f h (fold_right f t acc);;  
[51] ✓ 0.0s  
... val fold_right : ('a → 'b → 'b) → 'a list → 'b → 'b = <fun>  
  
▷ fold_right (fun n acc → string_of_int n :: acc) [1; 2; 3; 4; 5] [];;  
[54] ✓ 0.0s  
... - : string list = ["1"; "2"; "3"; "4"; "5"]
```



# Exercício

```
let count_by_group l =
```

[75] ✓ 0.0s

OCaml

```
... val count_by_group : 'a list → ('a * int) list = <fun>
```

```
count_by_group ["a"; "a"; "b"; "c"; "c"; "c"; "d"; "e"; "e"; "f"; "f"; "f"; "f"; "g"; "h"; "i"; "i"; "i"; "i"; "i"];;
```

[77] ✓ 0.0s

OCaml

```
... - : (string * int) list =  
[("a", 2); ("b", 1); ("c", 3); ("d", 1); ("e", 2); ("f", 4); ("g", 1);  
 ("h", 1); ("i", 5)]
```

# Exercício

```
let rec count_by_group l =  
  match l with  
  | [] → []  
  | h::t →  
    let tail = count_by_group t in  
    begin match tail with  
    | [] → [(h, 1)]  
    | (x, n)::rest → if h = x then (x, n+1)::rest else (h, 1)::tail  
    end  
end
```

[79] ✓ 0.0s

OCaml

... val count\_by\_group : 'a list → ('a \* int) list = <fun>



▶ count\_by\_group ["a"; "a"; "b"; "c"; "c"; "c"; "d"; "e"; "e"; "f"; "f"; "f"; "f"; "g"; "h"; "i"; "i"; "i"; "i"; "i"];;

[80] ✓ 0.0s

OCaml

# Exercício

```
let count_by_group l =  
  let rec f x acc =  
    match acc with  
    | [] → [(x, 1)]  
    | (y, n)::gs → if x = y then (y,n+1)::gs else (x,1)::acc  
  in List.fold_right f l []
```

[75] ✓ 0.0s

OCaml

... val count\_by\_group : 'a list → ('a \* int) list = <fun>

```
count_by_group ["a"; "a"; "b"; "c"; "c"; "c"; "d"; "e"; "e"; "f"; "f"; "f"; "f"; "g"; "h"; "i"; "i"; "i"; "i"; "i"];;
```

[77] ✓ 0.0s

OCaml

... - : (string \* int) list =  
[("a", 2); ("b", 1); ("c", 3); ("d", 1); ("e", 2); ("f", 4); ("g", 1);  
("h", 1); ("i", 5)]



# Exercício

```
let group_by l =
  let rec f x acc =
    match acc with
    | [] → [[ x ]]
    | g::gs →
      begin match g with
      | [] → assert false (* all lists have elements, see above *)
      | y::ys → if x = y then (x::y::ys)::gs else [ x ]::acc
      end
    in List.fold_right f l []
```

[72] ✓ 0.0s

... val group\_by : 'a list → 'a list list = <fun>



```
group_by [1; 1; 2; 3; 3; 3; 4; 5; 5; 6; 6; 6; 6; 7; 8; 9; 9; 9; 9; 9];;
```

[73] ✓ 0.0s

... - : int list list =  
[[1; 1]; [2]; [3; 3; 3]; [4]; [5; 5]; [6; 6; 6; 6]; [7]; [8];  
[9; 9; 9; 9; 9]]