

Linguagens e Ambientes de Programação (Aula Teórica 10)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Agenda

- Exercício sobre listas.
- Tipos algébricos indutivos.
- Funções indutivas sobre tipos algébricos.

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

- Raciocínio:
 - resolver o caso base (lista vazia)
 - a contagem de uma lista vazia, é a lista vazia
 - assumir que a solução está resolvida para a lista com $(n-1)$ elementos (hipótese de indução)
 - construir a solução juntando o último elemento (o primeiro na lista)
 - se o resultado é a lista vazia, agora terá apenas um elemento
 - se o resultado tem elementos
 - e o elemento está repetido, conta mais uma vez
 - se não está repetido, conta a primeira vez.

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



Execício

- Raciocínio:
 - resolver o caso base (lista vazia)
 - a contagem de uma lista vazia, é a lista vazia
 - assumir que a solução está resolvida para a lista com $(n-1)$ elementos
 - construir a solução juntando o último elemento (o primeiro na lista)
 - se o resultado é a lista vazia, agora terá apenas um elemento
 - se o resultado tem elementos
 - e o elemento está repetido, conta mais uma vez
 - se não está repetido, conta a primeira vez.
- **Este tipo de indução, pode-se tratar com um `fold_right`**

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

[illegible]

[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]]
```


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]]

Exercício

```
let rec ungroup l =  
  match l with  
  |  
  |  
  |
```

[15] ✓ 0.0s

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



```
let _ = assert (ungroup [[1; 1]; [2]; [3; 3; 3]; [4]; [5; 5]; [6; 6; 6; 6]; [7]; [8]; [9; 9; 9; 9; 9]] = [1; 1; 2; 3; 3; 3; 4; 5; 5; 6; 6; 6; 6; 7; 8; 9; 9; 9; 9; 9])  
  
let l = [1; 1; 2; 3; 3; 3; 4; 5; 5; 6; 6; 6; 6; 7; 8; 9; 9; 9; 9; 9] in assert (ungroup (group_by l) = l)
```

[16] ✓ 0.0s

... - : unit = ()

... - : unit = ()

Exercício

```
let rec ungroup l =  
  match l with  
  | [] → []  
  | []::gs → ungroup gs  
  | (x::xs)::gs → x::ungroup (xs::gs)
```

[15] ✓ 0.0s

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



```
let _ = assert (ungroup [[1; 1]; [2]; [3; 3; 3]; [4]; [5; 5]; [6; 6; 6; 6]; [7]; [8]; [9; 9; 9; 9; 9]] = [1; 1; 2; 3; 3; 3; 4; 5; 5; 6; 6; 6; 6; 7; 8; 9; 9; 9; 9; 9])  
  
let l = [1; 1; 2; 3; 3; 3; 4; 5; 5; 6; 6; 6; 6; 7; 8; 9; 9; 9; 9; 9] in assert (ungroup (group_by l) = l)
```

[16] ✓ 0.0s

... - : unit = ()

... - : unit = ()

Exercício

```
[22] let ungroup gs = List.fold_right (fun xs ys → (List.fold_right (fun x ys → x :: ys) xs ys)) gs []  
✓ 0.0s
```

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

```
[17] let ungroup gs = List.fold_right (@) gs []  
✓ 0.0s
```

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

```
[4] let ungroup xs = List.concat xs  
✓ 0.0s
```

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

Tipos Algébricos Indutivos

Tipos algébricos

- São tipos compostos por outros tipos
 - Produtos (tuplos e registos)
 - Somas (variantes ou uniões, com e sem dados)
- São inspecionados (ou eliminados) por pattern matching
- A recursão na definição de tipos permite a construção de estruturas regulares de tamanho indeterminado (estaticamente)

Tipo lista

- O tipo lista (built-in) é um tipo indutivo que é tratado por pattern matching.

```
type 'a my_list = Nil | Cons of 'a * 'a my_list
```

[3] ✓ 0.0s

```
... type 'a my_list = Nil | Cons of 'a * 'a my_list
```

```
Cons (1, Nil);  
Cons (1, Cons (2, Nil));  
Cons (1, Cons (2, Cons (3, Nil)));
```

[4] ✓ 0.0s

```
... - : int my_list = Cons (1, Nil)
```

```
... - : int my_list = Cons (1, Cons (2, Nil))
```

```
... - : int my_list = Cons (1, Cons (2, Cons (3, Nil)))
```

Tipo lista

- O tipo lista (built-in) é um tipo indutivo que é tratado por pattern matching.

```
[3] type 'a my_list = Nil | Cons of 'a * 'a my_list
      ✓ 0.0s
```

```
[6] let rec my_map f l =
      match l with
      | Nil → Nil
      | Cons (x, xs) → Cons (f x, my_map f xs)
      ✓ 0.0s
```

```
... val my_map : ('a → 'b) → 'a my_list → 'b my_list = <fun>
```

```
▷ [7] let _ = assert (my_map (fun x → x+1) (Cons (1, Cons (2, Cons (3, Nil))))) = (Cons (2, Cons (3, Cons (4, Nil))))
      ✓ 0.0s
```

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


Tipo árvore

- Os tipos indutivos permitem qualquer tipo de estrutura regular de valores

```
type 'a tree =
| Leaf
| Node of 'a * 'a tree * 'a tree

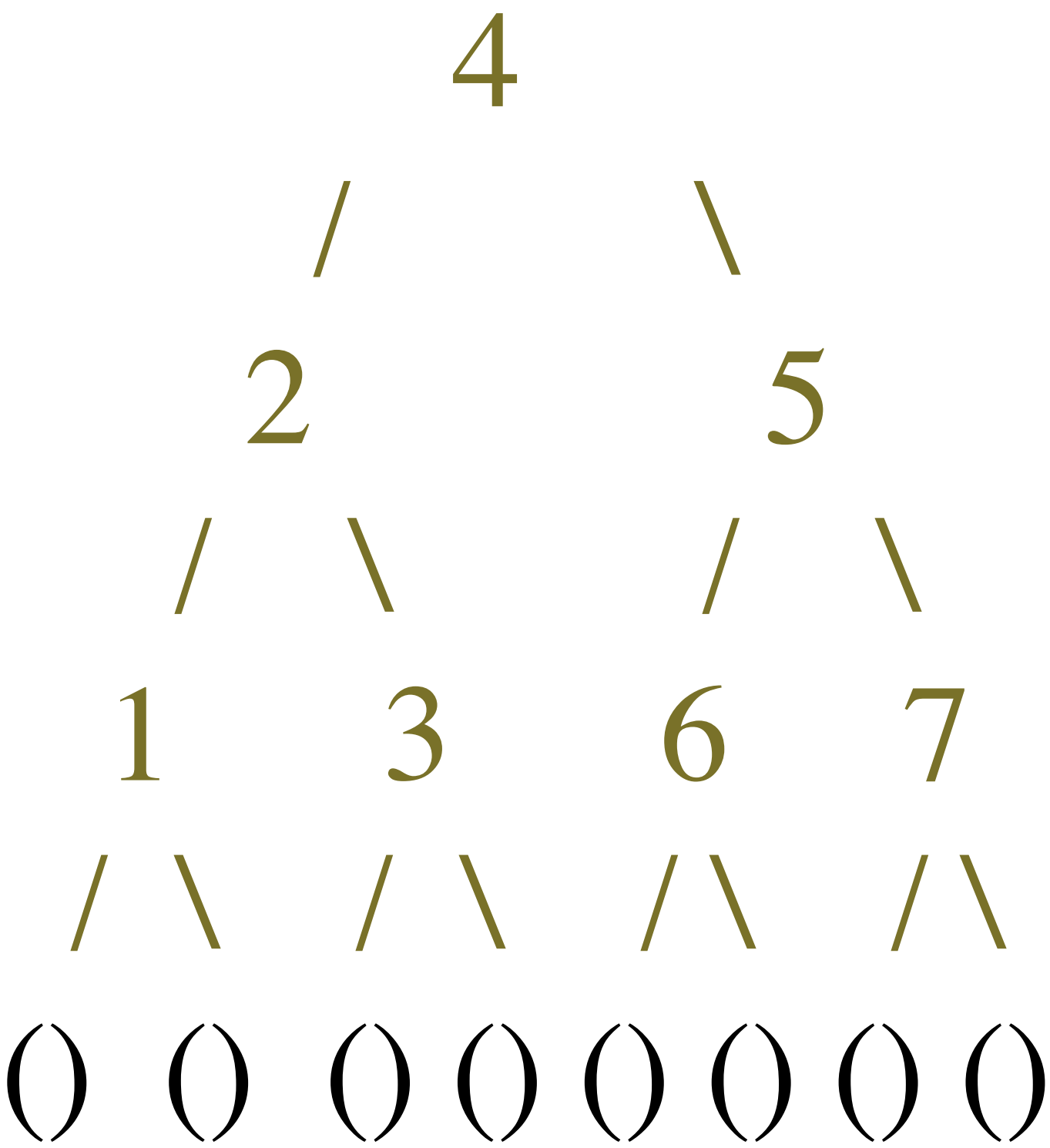
[14] ✓ 0.0s

... type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree

▷ let t =
  Node(4,
    Node(2,
      Node(1, Leaf, Leaf),
      Node(3, Leaf, Leaf)
    ),
    Node(5,
      Node(6, Leaf, Leaf),
      Node(7, Leaf, Leaf)
    )
  )

[15] ✓ 0.0s

... val t : int tree =
  Node (4, Node (2, Node (1, Leaf, Leaf), Node (3, Leaf, Leaf)),
    Node (5, Node (6, Leaf, Leaf), Node (7, Leaf, Leaf)))
```



Tipo árvore

```
let rec tree_size t =  
  match t with  
  | Leaf → 0  
  | Node (_, l, r) → 1 + tree_size l + tree_size r
```

[17] ✓ 0.0s

... val tree_size : 'a tree → int = <fun>



```
let _ = assert (tree_size t = 7)
```

[19] ✓ 0.0s

... - : unit = ()

Tipo árvore

```
let rec depth t =  
  match t with  
  | Leaf → 0  
  | Node (_, l, r) → 1 + max (depth l) (depth r)
```

[23] ✓ 0.0s

... val depth : 'a tree → int = <fun>



```
let _ = assert (depth t = 3)
```

[24] ✓ 0.0s

... - : unit = ()

Tipo árvore

```
let rec pre_order t =  
  match t with  
  | Leaf → []  
  | Node (v, l, r) → v :: (pre_order l) @ (pre_order r)
```

[26] ✓ 0.0s

... val pre_order : 'a tree → 'a list = <fun>



```
let _ = assert (pre_order t = [4; 2; 1; 3; 5; 6; 7])
```

[]

Tipo árvore

```
let rec in_order t =  
  match t with  
  | Leaf → []  
  | Node (v, l, r) → (in_order l) @ [v] @ (in_order r)
```

[28] ✓ 0.0s

... val in_order : 'a tree → 'a list = <fun>

```
let _ = assert (in_order t = [1; 2; 3; 4; 6; 5; 7])
```

[32] ✓ 0.0s

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

... - : unit = ()

Tipo árvore

```
let rec post_order t =  
  match t with  
  | Leaf → []  
  | Node (v, l, r) → (post_order l) @ (post_order r) @ [v]
```

[35] ✓ 0.0s

... val post_order : 'a tree → 'a list = <fun>



```
let _ = (assert (post_order t = [1; 3; 2; 6; 7; 5; 4]))
```

[36] ✓ 0.0s

... - : unit = ()

Tipo árvore

```
let rec map f t =  
  match t with  
  | Leaf → Leaf  
  | Node (v, l, r) → Node (f v, map f l, map f r)
```

[39] ✓ 0.0s

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



```
let _ = assert (map (fun x → x + 1) t =  
  Node(5,  
    Node(3,  
      Node(2, Leaf, Leaf),  
      Node(4, Leaf, Leaf)  
    ),  
    Node(6,  
      Node(7, Leaf, Leaf),  
      Node(8, Leaf, Leaf)  
    )  
  )  
)
```

[40] ✓ 0.0s

... - : unit = ()

Tipo árvore

```
let rec fold_tree_in_order f acc t =  
  match t with  
  | Leaf → acc  
  | Node (v, l, r) →  
    let acc_l = fold_tree f acc l in  
    let acc_n = f v acc_l in  
    let acc_r = fold_tree f acc_n r in  
    acc_r
```

[45] ✓ 0.0s

... val fold_tree_in_order : ('a → 'b → 'b) → 'b → 'a tree → 'b = <fun>

⏮ ⏭ E



```
let _ = assert (fold_tree_in_order (+) 0 t = 28)
```

[48] ✓ 0.0s

... - : unit = ()

Linguagens e Ambientes de Programação (Aula Teórica 11)

LEI - Licenciatura em Engenharia Informática

João Costa Seco (joao.seco@fct.unl.pt)



NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Agenda

- Merge sort.
- Tipos algébricos: Árvores
- Funções indutivas sobre tipos algébricos.

```
let rec fibonacci_cps n k =  
  if n ≤ 1 then k n  
  else fibonacci_cps (n - 1) (fun fib1 → fibonacci_cps (n - 2) (fun fib2 → k (fib1 + fib2)))  
  
let fibonacci n =  
  fibonacci_cps n (fun result → result)
```

[57] ✓ 0.0s

... val fibonacci_cps : int → (int → 'a) → 'a = <fun>

... val fibonacci : int → int = <fun>



```
fibonacci 10;;
```

[61] ✓ 0.0s

... - : int = 55