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

```
(** [pack l] is the list of pairs of elements that contain the number of times an element appears in a sequence [l].
    pre: [true]
    *)
let rec pack l =
  match l with
  |
  |
  |
  |
  |
  |
  end
```

[24] ✓ 0.0s OCaml

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

+ Code + Markdown

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

let _ = assert (pack ["a"; "a"; "b"; "c"; "c"; "c"; "d"; "e"; "e"; "f"; "f"; "f"; "f"; "g"; "h"; "i"; "i"; "i"; "i"; "i
```

[26] ✓ 0.0s OCaml

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

- : unit = ()
```

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

```
(** [pack l] is the list of pairs of elements that contain the number of times an element appears in a sequence [l].
    pre: [true]
    *)
let rec pack l =
  match l with
  | [] → []
  | x::xs →
    let tail = pack xs in
    begin match tail with
    | [] → [(x, 1)]
    | (g, n)::gs → if g = x then (g, n+1)::gs else (x, 1)::tail
  end
```

[24] ✓ 0.0s OCaml

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

+ Code + Markdown

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

let _ = assert (pack ["a"; "a"; "b"; "c"; "c"; "c"; "d"; "e"; "e"; "f"; "f"; "f"; "f"; "g"; "h"; "i"; "i"; "i"; "i"; "i
```

[26] ✓ 0.0s OCaml

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

- : unit = ()
```

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

```
(** [pack l] is the list of pairs of elements that contain the number of times an element appears in a sequence [l].
    pre: [true]
    *)
let pack l =
  let rec f x acc =
    match acc with
    | [] → [(x, 1)]
    | (g, n)::gs → if x = g then (g,n+1)::gs else (x,1)::acc
in List.fold_right f l []
```

[6] OCaml

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

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

let _ = assert (pack ["a"; "a"; "b"; "c"; "c"; "c"; "d"; "e"; "e"; "f"; "f"; "f"; "f"; "g"; "h"; "i"; "i"; "i"; "i"; "i
```

[27] ✓ 0.0s OCaml

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

- : unit = ()
```

Exercício

```
let rec unpack l =
  match l with
  | [] → []
  | (x, n)::xs →
    if n = 1 then x::unpack xs
    else x::unpack ((x, n-1)::xs)
```

[8] OCaml

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

```
let _ = assert (unpack [("a", 2); ("b", 1); ("c", 3); ("d", 1); ("e", 2); ("f", 4); ("g", 1); ("h", 1); ("i", 5)] =  ["

let l = ["a"; "a"; "b"; "c"; "c"; "c"; "d"; "e"; "e"; "f"; "f"; "f"; "f"; "g"; "h"; "i"; "i"; "i"; "i"; "i"] in
  assert (unpack (pack l) = l)
```

[9] OCaml

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

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

Exercício

```
let group_by l =
  let rec f x acc =
    
    
    
    
    
    
  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 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

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

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 ungroup gs = List.fold_right (fun xs ys → (List.fold_right (fun x ys → x :: ys) xs ys)) gs []
```

[22] ✓ 0.0s

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

```
let ungroup gs = List.fold_right (@) gs []
```

[17] ✓ 0.0s

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

```
let ungroup xs = List.concat xs
```

[4] ✓ 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.

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

[3] ✓ 0.0s

```
let rec my_map f l =
  match l with
  | Nil → Nil
  | Cons (x, xs) → Cons (f x, my_map f xs)
```

[6] ✓ 0.0s

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

```
let _ = assert (my_map (fun x → x+1) (Cons (1, Cons (2, Cons (3, Nil)))) = (Cons (2, Cons (3, Cons (4, Nil)))))
```

[7] ✓ 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
```

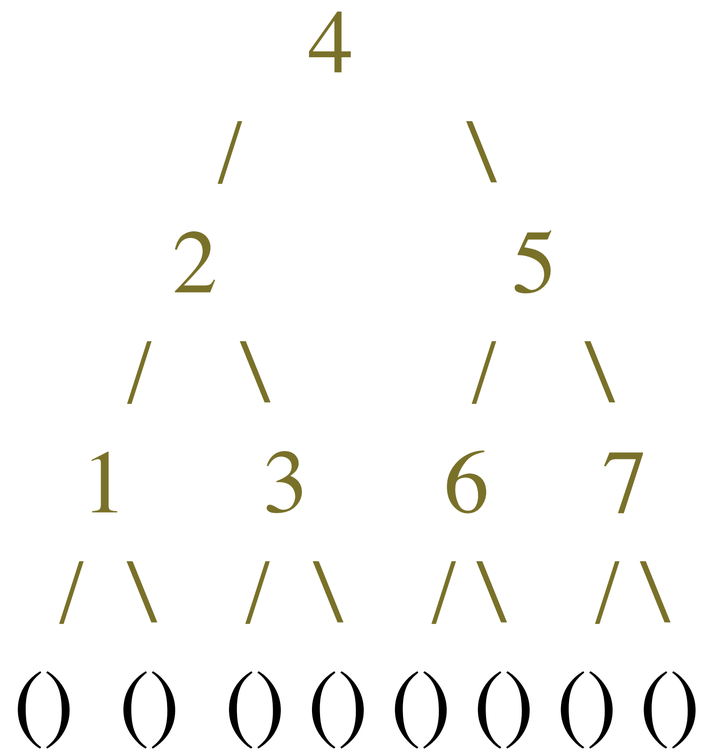
[14] ✓ 0.0s

```
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

```
[17]  ✓ 0.0s
```

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

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

```
[19]  ✓ 0.0s
```

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

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

Tipo árvore

```
[23]  ✓ 0.0s
```

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

```
val depth : 'a tree → int = <fun>
```

```
[24]  ✓ 0.0s
```

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

```
- : 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_in_order f acc l in
      let acc_n = f v acc_l in
      let acc_r = fold_tree_in_order f acc_n r in
      acc_r
```

[45] ✓ 0.0s OCaml

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

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

[49] ✓ 0.0s OCaml

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

Tipo árvore

```
let rec fold_tree_in_order f acc t =
  match t with
  | Leaf →
  | Node (v,
      let ac
      let ac
      let ac
      acc_r
```

[45] ✓ 0.0s

```
val fold_tree_i
```

```
let _ = asse
```

[49] ✓ 0.0s

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

```
fold_tree_in_order (fun x acc → x :: acc) [] t;;

fold_tree_in_order (fun x acc → acc @ [ x ]) [] t;;

List.rev (fold_tree_in_order (fun x acc → x :: acc) [] t)
```

[52] ✓ 0.0s

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

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

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