Programming Languages and Environments (Lecture 8)

LEI - Licenciatura em Engenharia Informática

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

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

Syllabus

- Lists
- Recursive functions over lists

Primitive lists in OCaml

- Lists are homogeneous and immutable collections of values.
- They can also be heterogeneous with sum types.
- The can be created by literals

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

[51]  []
      ✓ 0.0s
...   - : 'a list = []
```

- Or by the operator **::**

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

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

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

- The list type is parametric on the type of its elements.

Primitive lists in OCaml

- Lists are homogeneous and immutable collections of values.

```
[[[[];[]];[[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.)]]
```

The list type

- The list type is parametric and inductive with two cases
- `Nil` or empty list (`[]`) where the type of elements is still to be defined.
- `Cons` or (`h::t`) with a head (`h`) and tail (`t`). The head has same type as the elements of tail.

```
type 'a list =
| []
| (::) of 'a * 'a list
```

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

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

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

Deconstruction of the List type

- The list type is an *inductive* and parametric type, and its cases can be analyzed with 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>
```

Deconstruction of the List type

- The list type is an *inductive* and parametric type, and its cases can be analyzed with 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>
```

Deconstruction of the List type

- The list type is an *inductive* and parametric type, and its cases can be analyzed with 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>
```

Deconstruction of the List type

- The list type is an *inductive* and parametric type, and its cases can be analyzed with 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>
```

To make writing these definitions easier, the syntactic construct `function` allows pattern matching of a parameter.

Lists are immutable values

- List modifications are done by creating new lists.

```
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
```

Lists are immutable values

- List modifications are done by creating new lists.

```
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>
```

Lists are immutable values

- List modifications are done by creating new lists.

```
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 = ()
```

Lists are immutable values

- List modifications are done by creating new lists.

```
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]
```

Higher-order iteration over Lists

- The function map creates a new list (`'b list`) applying a function `f` that is a transformation (`'a -> 'b`) over a list of input (`'a list`).
- This function is already defined within the `List` module.

```
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]
```

Higher-order iteration over Lists

- The function `fold_left` folds every element of a list (`l`), applying a function `f` that accumulates the result into a single value.
- This function is also defined in the List module.
- It is conceived as follows:

```
f (...(f (f acc l[0]) l[1])...) l[n]
```

```
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"
```

Higher-order iteration over Lists

- The function `fold_right` folds every element of a list (`l`), applying a function `f` that accumulates the result into a single value.
- This function is also defined in the List module.
- It differs from the previous fold by the directed of the iteration.
- It is conceived as follows:

```
f l[0] (f l[1] (...(f l[n] acc)...)))
```

```
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"]
```

Exercise

```
[75]  let count_by_group l =
      ✓ 0.0s                                                    OCaml
...   val count_by_group : 'a list → ('a * int) list = <fun>
```

```
[77]  count_by_group ["a"; "a"; "b"; "c"; "c"; "c"; "d"; "e"; "e"; "f"; "f"; "f"; "f"; "g"; "h"; "i"; "i"; "i"; "i"; "i"];;
      ✓ 0.0s                                                    OCaml
...   - : (string * int) list =
      [("a", 2); ("b", 1); ("c", 3); ("d", 1); ("e", 2); ("f", 4); ("g", 1);
       ("h", 1); ("i", 5)]
```

Exercise

```
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
```

[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

Exercise

```
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)]
```

Exercise

```
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]]
```

Exercise

```
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]]
```