

Lezione 1: Un'introduzione a OCaml

Fausto Spoto

13 gennaio 2005

Valori, espressioni, costanti, funzioni

```
[spoto@resistenza]> ocaml
Objective Caml version 3.06

# 5;;
- : int = 5
# 5 * 2;;
- : int = 10
# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265359
# let square x = x *. x;;
val square : float -> float = <fun>
# square(sin pi) +. square(cos pi);;
- : float = 1.
#
```

Funzioni ricorsive: il fattoriale

```
# let fact n =  
    if n = 0 then 1 else n * fact n - 1;;
```

Unbound value fact

```
# let rec fact n =  
    if n = 0 then 1 else n * fact n - 1;;
```

```
val fact : int -> int = <fun>
```

```
# fact 10;;
```

Stack overflow during evaluation
(looping recursion?).

```
# let rec fact n =  
    if n = 0 then 1 else n * fact (n - 1);;
```

```
val fact : int -> int = <fun>
```

```
# fact 10;;
```

```
- : int = 3628800
```

```
#
```

Errori di tipo

```
# 5 + 4.0;;
```

This expression has type float
but is here used with type int

```
# 5 +. 4.0;;
```

This expression has type int
but is here used with type float

```
# square 3;;
```

This expression has type int
but is here used with type float

```
# square 3.0 4;;
```

This function is applied to too many arguments

```
#
```

Altri tipi predefiniti

```
# 'f';;  
- : char = 'f'  
# "ciao";;  
- : string = ciao  
# (1 = 2 - 1) = true;;  
- : bool = true  
# "ciao" = "ciao";;  
- : bool = true  
#
```

Tipo predefinito list

```
# [2; 1; 2];;
```

```
- : int list = [2; 1; 2]
```

```
# [2; 3.4];;
```

This expression has type float
but is here used with type int

```
# [2; 3; -5];;
```

```
- : int list = [2; 3; -5]
```

```
# 12 :: [2; 3; -5];;
```

```
- : int list = [12; 2; 3; -5]
```

```
# [12] :: [2; 3; -5];;
```

This expression has type int
but is here used with type int list

La lista vuota

```
# 12 :: [];;  
- : int list = [12]  
# [12] :: [];;  
- : int list list = [[12]]  
# [];;  
- : 'a list = []  
#
```

Pattern matching: la funzione length

```
# let rec length lst =  
    match lst with  
        [] -> 0  
        | head :: tail -> 1 + length tail;;  
val length : 'a list -> int = <fun>  
# length [3; 4; -2; 0];;  
- : int = 4  
# length ["ciao"; "bel"; "micio"];;  
- : int = 3  
#
```

Pattern matching: la funzione ordered

```
# let rec ordered lst =
  match lst with
  | [] -> true
  | first :: [] -> true
  | first :: second :: tail ->
    if first <= second
    then ordered (second :: tail)
    else false;;

val ordered : 'a list -> bool = <fun>
# ordered [2; 3; 5];;
- : bool = true
# ordered [3.0; 4.5; 4.55];;
- : bool = true
# ordered [2; 1; 0];;
- : bool = false
#
```

Definizioni locali

```
# let area r =  
    let pi = 3.1415  
    in r *. r *. pi;;  
val area : float -> float = <fun>  
# area 12.5;;  
- : float = 490.859375  
# pi;;  
Unbound value pi
```

Funzioni come valori

```
# let deriv f =  
    function x -> (f (x +. 1e-6) -. f x) /. 1e-6;;  
val deriv : (float -> float) -> float -> float  
          = <fun>  
  
# deriv sin;;  
- : float -> float = <fun>  
  
# deriv sin 1.0;;  
- : float = 0.540301885121  
  
# deriv (deriv sin);;  
- : float -> float = <fun>  
  
# deriv (deriv sin) 1.0;;  
- : float = -0.841549052666  
  
#
```

Funzioni come valori: *mappature*

```
# let rec map f lst =
  match lst with
  [] -> []
  | head :: tail -> f head :: map f tail;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map (function n -> n + 1) [3; 5; -3; 0];;
- : int list = [4; 6; -2; 1]
# let rec fold f id lst =
  match lst with
  [] -> id
  | head :: tail -> f head (fold f id tail);;
val fold : ('a -> 'b -> 'b) -> 'b -> 'a list
          -> 'b = <fun>
# fold (+) 0 [2; -3; 4; 7; 1; 3; 2];;
- : int = 16
```

Tipi algebrici o varianti: alberi binari di interi

```
# type intbtree = Empty
    | Node of int * intbtree * intbtree;;
type intbtree = Empty
    | Node of int * intbtree * intbtree
# let t = Node (13, Node(17,Empty,Empty),
    Node(23,Empty,Empty));;
val t : intbtree =
  Node (13, Node (17, Empty, Empty),
    Node (23, Empty, Empty))
# let rec count tree =
  match tree with
  Empty -> 0
  | Node (value, left, right) -> 1 + (count left)
    + (count right);;
val count : intbtree -> int = <fun>
# count t;;
- : int = 3
```

Polimorfismo parametrico dei tipi: alberi binari

```
# type 'a btree = Empty |  
    Node of 'a * 'a btree * 'a btree;;  
type 'a btree = Empty |  
    Node of 'a * 'a btree * 'a btree  
# let t = Node (13, Node(17,Empty,Empty),  
    Node(23,Empty,Empty));;  
val t : int btree =  
    Node (13, Node (17, Empty, Empty),  
        Node (23, Empty, Empty))  
# let rec member x btree =  
    match btree with  
        Empty -> false  
    | Node(y,left,right) -> if y = x then true  
        else member x left or member x right;;  
val member : 'a -> 'a btree -> bool = <fun>  
# member 13 t;;  
- : bool = true
```

Eccezioni

```
# exception Empty_tree;;
exception Empty_tree
# let rec largest tree =
  match tree with
  | Empty -> raise Empty_tree
  | Node(v,Empty,Empty) -> v
  | Node(v,Node(vl,ll,rl),Empty) ->
    max v (largest (Node(vl,ll,rl)))
  | Node(v,Empty,Node(vr,lr,rr)) ->
    max v (largest (Node(vr,lr,rr)))
  | Node(v,left,right) ->
    max v (max (largest left) (largest right));;
val largest : 'a btree -> 'a = <fun>
# largest t;;
- : int = 23
# largest Empty;;
Exception: Empty_tree.
#
```

Intercettare le eccezioni

```
# let nice_largest tree =  
  try  
    largest tree  
  with Empty_tree -> 0;;  
val nice_largest : int btree -> int = <fun>  
# nice_largest Empty;;  
- : int = 0  
# exit 0;;  
[spoto@resistenza]>
```

Contenuto del file fib.ml

```
let rec fib n =
  if n < 2 then 1 else fib (n - 1)
    + fib (n - 2);;

let main () =
  let arg = int_of_string Sys.argv.(1) in
  print_int (fib arg);
  print_newline();
  exit 0;;

main ();;
```

```
[spoto@resistenza]> ocamlc -o fib fib.ml
```

```
[spoto@resistenza]> ./fib 5
```