

OCAML

☞ nucleo funzionale puro

- funzioni (ricorsive)
- tipi e pattern matching
- primitive utili: liste
- trascriviamo pezzi della semantica denotazionale

☞ componente imperativo

- variabili e assegnamento
- primitive utili: arrays

☞ moduli

Espressioni pure

Objective Caml version 2.00+3/Mac1.0a1

```
# 25;;
```

```
- : int = 25
```

```
# true;;
```

```
- : bool = true
```

```
# 23 * 17;;
```

```
- : int = 391
```

```
# true & false;;
```

```
- : bool = false
```

```
# 23 * true;;
```

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

```
# if 2 = 3 then 23 * 17 else 15;;
```

```
- : int = 15
```

```
# if 2 = 3 then 23 else true;;
```

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

Funzioni

```
# function x -> x + 1;;  
- : int -> int = <fun>  
# (function x -> x + 1) 3;;  
- : int = 4  
# (function x -> x + 1) true;;  
This expression has type bool but is here used with type int  
# function x -> x;;  
- : 'a -> 'a = <fun>  
# function x -> function y -> x y;;  
- : ('a -> 'b) -> 'a -> 'b = <fun>  
# (function x -> x) 2;;  
- : int = 2  
# (function x -> x) (function x -> x + 1);;  
- : int -> int = <fun>  
# function (x, y) -> x + y;;  
- : int * int -> int = <fun>  
# (function (x, y) -> x + y) (2, 33);;  
- : int = 35
```


Let binding

```
# let x = 3;;  
val x : int = 3  
# x;;  
- : int = 3  
# let y = 5 in x + y;;  
- : int = 8  
# y;;  
Unbound value y  
# let f = function x -> x + 1;;  
val f : int -> int = <fun>  
# f 3;;  
- : int = 4  
# let f x = x + 1;;  
val f : int -> int = <fun>  
# f 3;;  
- : int = 4  
# let fact x = if x = 0 then 1 else x * fact(x - 1) ;;  
Unbound value fact
```


Let rec binding

```
# let rec fact x = if x = 0 then 1 else x * fact(x - 1) ;;  
val fact : int -> int = <fun>  
# fact (x + 1);;  
- : int = 24
```


Tipi 1

```
# type ide = string;;  
type ide = string  
# type expr = | Den of ide | Val of ide | Fun of ide * expr  
| Plus of expr * expr | Apply of expr * expr;;  
type expr =  
| Den of ide  
| Val of ide  
| Fun of ide * expr  
| Plus of expr * expr  
| Apply of expr * expr  
  
# Apply(Fun("x",Plus(Den "x", Den "x")), Val "y");;  
- : expr = Apply (Fun ("x", Plus (Den "x", Den "x")), Val "y")
```


Tipi 2

```
# type eval = Int of int | Bool of bool | Efun of expr
  | Unbound;;
type eval = | Int of int | Bool of bool | Efun of expr |
  Unbound
# type env = ide -> eval;;
type env = ide -> eval
# let bind (rho, i, d) =
    function x -> if x = i then d else rho x;;
val bind : (ide -> eval) * ide * eval -> (ide -> eval) = <fun>
```


Tipi 3

```
# type com = Assign of ide * expr | Ifthenelse of expr *  
  com list * com list | While of expr * com list;;  
type com =  
  | Assign of ide * expr  
  | Ifthenelse of expr * com list * com list  
  | While of expr * com list  
  
# While(Den "x", [Assign("y", Plus(Val "y", Val "x"))]);;  
- : com = While (Den "x", [Assign ("y", Plus (Val "y", Val  
  "x"))])
```


Un tipo primitivo utile: le liste

```
# let l1 = [1; 2; 1];;  
val l1 : int list = [1; 2; 1]  
# let l2 = 3 :: l1;;  
val l2 : int list = [3; 1; 2; 1]  
# let l3 = l1 @ l2;;  
val l3 : int list = [1; 2; 1; 3; 1; 2; 1]  
# List.hd l3;;  
- : int = 1  
# List.tl l3;;  
- : int list = [2; 1; 3; 1; 2; 1]  
# List.length l3;;  
- : int = 7
```


Tipi e pattern matching

```
type expr =  
  | Den of ide  
  | Fun of ide * expr  
  | Plus of expr * expr  
  | Apply of expr * expr  
type eval = | Int of int | Bool of bool | Efun of expr | Unbound  
type env = ide -> eval  
  
# let rec sem (e, rho) = match e with  
  | Den i -> rho i  
  | Plus(e1, e2) -> plus(sem (e1, rho), sem (e2, rho))  
  | Fun(i, e) -> Efun(Fun(i, e))  
  | Apply(e1, e2) -> match sem(e1, rho) with  
    | Efun(Fun(i, e)) -> sem(e, bind(rho, i, sem(e2, rho)))  
    | _ -> failwith("wrong application");;  
val sem : expr * env -> eval = <fun>
```


Variabili e frammento imperativo

```
# let x = ref(3);;
val x : int ref = {contents=3}
# !x;;
- : int = 3
# x := 25;;
- : unit = ()
# !x;;
- : int = 25
# x := !x + 2; !x;;
- : int = 27
```


Un tipo primitivo mutabile: l'array

```
# let a = [| 1; 2; 3 |];;
val a : int array = [|1; 2; 3|]
# let b = Array.make 12 1;;
val b : int array = [|1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1|]
# Array.length b;;
- : int = 12
# Array.get a 0;;
- : int = 1
# Array.get b 12;;
Uncaught exception: Invalid_argument("Array.get")
# Array.set b 3 131;;
- : unit = ()
# b;;
- : int array = [|1; 1; 1; 131; 1; 1; 1; 1; 1; 1; 1; 1|]
```


Moduli: interfaccie

```
# module type PILA =  
  sig  
    type 'a stack                (* abstract *)  
    val emptystack : 'a stack  
    val push : 'a stack -> 'a -> 'a stack  
    val pop : 'a stack -> 'a stack  
    val top : 'a stack -> 'a  
  end;;  
module type PILA =  
  sig  
    type 'a stack  
    val emptystack : 'a stack  
    val push : 'a stack -> 'a -> 'a stack  
    val pop : 'a stack -> 'a stack  
    val top : 'a stack -> 'a  
  end
```


Moduli: implementazione

```
# module SpecPila: PILA =  
  struct  
    type 'a stack = Empty | Push of 'a stack * 'a  
    let emptystack = Empty  
    let push p a = Push(p,a)  
    let pop p = match p with  
      | Push(p1, _) -> p1  
    let top p = match p with  
      | Push(_, a) -> a  
  end;;  
module SpecPila : PILA
```