OCaml: un veloce ripasso

Lo stile funzionale

- In Java (ma anche in C) l'effetto osservabile di un programma è la **modifica** dello stato

```
temp = pair.x;
pair.x = pair.y;
pair.y = temp;
```

- In Ocaml il risultato della computazione è la ***produzione*** di un nuovo valore

```
let x = 5 in (x,x)
```

Value-oriented programming

- Programmazione funzionale: "**value-oriented programming**"
  - un programma Ocaml è una espressione
  - una espressione Ocaml denota un **valore**
- L'esecuzione di un programma OCaml può essere vista come una sequenza di passi di calcolo (semplificazioni di espressioni) che alla fine produce un valore

Espressioni

- Sintassi: regole di buona formazione
- Semantica
  - regole di **type checking** (tipo o errore)
  - **regole di esecuzione** che garantiscono che espressioni tipate producono un valore

Valori

- Un **valore** è una espressione che non deve essere valutata ulteriormente
  - **34** è un valore di tipo **int**
  - **34+17** è un'espressione di tipo int ma non è un valore

Valori

- La notazione **<exp> ⇒ <val>** indica che la espressione **<exp>** quando valutata calcola il valore **<val>**

  3 ⇒ 3 (valori di base)
  3+4 ⇒ 7
  2*(4+5) ⇒ 18

eval(e) = v metanotazione per **e ⇒ v**

let

- **Sintassi: let x = e1 in e2**
  - **x** *identificatore*
  - **e1, e2** *espressioni*
  - **let x = e1 in e2** espressione
  - **x = e1** *binding*
- **let** x = 2 **in** x + x
  let inc x = x + 1 **in** inc 10
  let y = "programmazione" **in** (**let** z = " 2" **in** y^z)

let

- **let x = e1 in e2**
- **Regola di valutazione**
  - **eval(e1) = v1**
  - sostituire il valore v1 per tutte le occorrenze di x in e2 ottenendo l'espressione e2'
  - **subst(e2, x, v1) = e2'**
  - **eval(e2') = v**
  - **eval(let x = e1 in e2**) = **v**

let

$$\frac{eval(e1) = v1 \;\; subst(e2, x, v1) = e2' \;\; eval(e2') = v}{eval(let \; x = e1 \; in \; e2) = v}$$

Esempio

- **eval(let x = 1 + 4 in x * 3)**
  - eval(1 + 4) = 5
- **eval(let x = 5 in x*3)**
  - subst(x * 3, x, 5) = 5 * 3
- **eval(5 * 3) = 15**
- **eval(let x = 1 + 4 in x * 3) = 15**

let binding = scope

```
let x = 42 in
    (* y non è visibile *)
    x + (let y = "3110" in
        (* y è visibile *)
            int_of_string y)
```

scope: overlapping

```
let x = 5 in ((let x = 6 in x) + x)
```

Due casi

```
((let x = 6 in x) + 5)
```

```
((let x = 6 in 5) + 5)
```

scope: overlapping

```
let x = 5 in ((let x = 6 in x) + x)
```

Due casi

```
((let x = 6 in x) + 5)
```

```
((let x = 6 in 5) + 5)
```

scope: overlapping

```
let x = 5 in ((let x = 6 in x) + x)
```

Due casi

```
((let x = 6 in x) + 5)
```

~~((let x = 6 in 5) + 5)~~

Alpha conversione

- L'identità puntuale delle variabili legate non ha alcun senso!!
- In matematica
  - f(x) = x * x
  - f(y) = y * y

    sono la medesima funzione!!
- In informatica
  - **let** x = 5 **in** ((**let** x = 6 **in** x) + x)
  - **let** x = 5 **in** ((**let** y = 6 **in** y) + x)

Dichiarazione di funzioni

```
let f (x : int) : int =
    let y = x * 10 in
        y * y;;
```

Funzioni ricorsive

```
let rec pow x y =
    if y = 0 then 1
    else x * pow x  (y - 1);;
```

Applicazione di funzioni

```
let f (x : int) : int =
    let y = x * 10 in
        y * y;;

f 5;;
- : int = 2500
```

Applicazione di funzioni

```
let rec pow x y =
   if y = 0 then 1
   else x * pow x  (y - 1);;

pow 2 3;;
-: int 8
```

Dichiarazione

- La valutazione di una dichiarazione di una funzione è la funzione stessa
  - le funzioni sono valori

Applicazione

- **eval(e0 e1 ... en) = v' se**
  - ***eval(e0) = let f x1 ... xn = e***
  - ***eval(e1) = v1 ... eval(en)= vn***
  - ***subst(e, x1, …, xn, v1, ..., vn) = e'***
  - ***eval(e') = v'***

Esempi

- let double (x : int) : int = 2 * x;;
- let square (x : int) : int = x * x;;
- let quad (x : int) : int = double (double x);;
- let fourth (x : int) : int = square (square x)

Esempi

- let twice ((f : int -> int), (x : int)) : int = f (f x)
- let quad (x : int) : int = twice (double, x)
- let fourth (x : int) : int = twice (square, x)
- *twice*
  - *higher-order function*: una funzione da funzioni ad altri valori

Funzioni higher-order

- let compose ((f, g) : (int -> int) *
  (int -> int)) (x : int) : int = f (g x)
- let rec ntimes ((f, n) : (int -> int) * int) =
  if n = 0 then (fun (x : int) -> x)
  else compose (f, ntimes (f, n - 1))

OCaml List

- **let** lst = [1; 2; 3];;
- **let** empty = [];;
- **let** longer = 5::lst;;
- **let** another = 5::1::2::3::[]

List: sintassi

- **[]** la lista vuota (nil derivato dal LISP)
- **e1::e2** inserisce l'elemento **e1** in testa alla lista **e2**
  - (:: = LISP cons)
- **[e1; e2; ...; en]** notazione sintattica per la lista **e1::e2::...::en::[]**

Accedere a una lista

- Strutturalmente una lista può essere
  - nil, []
  - la lista ottenuta mediante una operazione di cons di un elemento a una lista
- **Idea**: usare il **pattern matching** per accedere agli elementi della lista
- **let** empty lst = **match** lst **with**
  | [] -> **true**
  | h::t -> **false**

Esempi

- **let rec** sum xs = **match** xs **with**
  | [] -> 0
  | h::t -> h + sum t
- **let rec** concat ss = **match** ss **with**
  | [] -> ""
  | s::ss' -> s ^ (concat ss')
- **let rec** append lst1 lst2 = **match** lst1 **with**
  | [] -> lst2
  | h::t -> h::(append t lst2)

Pattern Matching

- **match e with**
  | p1 -> e1
  | p2 -> e2
  | ...
  | pn -> en
- Le espressioni **pi** si chiamano *pattern*

Pattern matching

- Il pattern **[]** "match" solamente il valore **[]**
- **match** [] **with**
  | [] ->0
  | h::t -> 1
  (* restituisce il valore 0 *)
- **match** [] **with**
  | h::t -> 0
  | [] -> 1
  (* restituisce il valore 1 *)

Pattern matching

- Il pattern **h::t** “match” una qualsiasi lista con almeno un elemento, e inoltre ha l’effetto di legare quell’elemento alla variabile **h** e la lista rimanente alla variabile **t**
- **match** [1; 2; 3] **with**
  | [] -> 0
  | h::t -> h (* restituisce il valore 1 *)
- **match** [1; 2; 3] **with**
  | [] -> []
  | h::t -> t (* restituisce il valore [2; 3] *)

Altri esempi

- Il pattern **a::[]** “match” tutte le liste con esattamente un elemento
- Il pattern **a::b** “match” tutte le liste con almeno un elemento
- Il pattern **a::b::[]** “match” tutte le liste con esattamente due elementi
- Il pattern **a::b::c::d** “match” tutte le liste con almeno tre elementi

Un esempio più complicato

- **let rec** drop_val v lst = **match** lst **with**
  | [] -> []
  | h::t -> **let** t' = drop_val v t **in**
  if h = v **then** t' **else** h::t'

Un altro esempio

- **let rec** max_list = **function**
  | [] -> ???
  | h::t -> max h (max_list t)
- Cosa mettiamo al posto di ??? ?
- **min_int** è una scelta possibile ...
- O sollevare una exception ...
- In Java, avremmo potuto restituire **null**...
- …ma siamo in Ocaml, che ci fornisce una altra soluzione

Option type

- **let rec** max_list = **function**
  | [] -> **None**
  | h::t -> **match** max_list t **with**
  | **None** -> **Some** h
  | **Some** x -> **Some** (max h x)

  (* max_list : 'a list -> 'a option *)

Iteratori

- **let rec** map f = **function**
  | [] -> []
  | x::xs -> (f x)::(map f xs)

Parametro impicito di tipo lista

- **map : ('a -> 'b) -> 'a list -> 'b list**

- **let is_even x = (x mod 2 = 0);;**
- **let lst = map is_even [1; 2; 3; 4];;**

Iteratori

- **let rec** map f = **function**
  | [] -> []
  | x::xs -> (f x)::(map f xs)
- **map : ('a -> 'b) -> 'a list -> 'b list**
- **let is_even x = (x mod 2 = 0);;**
- **let lst = map is_even [1; 2; 3; 4];;**
- Risultato **[false; true; false; true]**

Definire tipi di dato in OCaml

Ocaml permette al programmatore di definire *nuovi* tipi di dato

```
type giorno =
  | Lunedi
  | Martedi
  | Mercoledi
  | Giovedi
  | Venerdi
  | Sabato
  | Domenica
```

Dichiarazione di tipo

Nome del tipo

Costruttori

I costruttori definiscono i valori del tipo di dato
Sabato ha tipo *giorno*
[Venerdi, Sabato, Domenica] ha tipo *giorno list*

Pattern matching

Il pattern matching fornisce un modo efficiente per accedere ai valori di un tipo di dato

```
let string_of_g (g : giorno) : string =
    begin match g with
        | Lunedi -> "Lunedi"
        | Martedi -> "Martedi"
        | :
        | :
        | Domenica -> "Domenica"
 end
```

Il pattern matching **segue** la struttura sintattica dei valori del tipo di dato: i costruttori

Astrazioni sui dati

- Avremmo potuto rappresentare il tipo di dato *giorno* tramite dei semplici valori interi
  - Lunedi = 1, Martedi = 2, …, Domenica = 7
- Ma…
  - il tipo di dato primitivo *int* fornisce un insieme di operazioni differenti da quelle significative sul tipo di dato *giorno*, Mercoledi – Domenica *non* avrebbe alcun senso
  - esistono un numero maggiore di valori interi che di valori del tipo giorno
- Morale: I linguaggi di programmazione moderni (Java, C#, C++, Ocaml, …) forniscono strumenti per definire tipi di dato

Ocaml Type

- I costruttori possono trasportare "valori"

```
# type foo =
    | Nothing
    | Int of int
    | Pair of int * int
    | String of string;;

type foo = Nothing | Int of int | Pair of int * int | String of
string
```

Dichiarazione da valutare

Risultato

Valori del tipo *foo*

```
Nothing
Int 3
Pair (4, 5)
String "hello"...
```

Pattern matching

```
let get_count (f : foo) : int
=
    begin match f with
        | Nothing -> 0
        | Int(n) -> n
        | Pair(n,m) -> n + m
        | String(s) -> 0
    end
```

Tipi di dato ricorsivi

```
# type tree =
    | Leaf of int
    | Node of tree * int * tree;;

type tree = Leaf of int | Node of tree * int * tree
```

```
let t1 = Leaf 3
let t2 = Node(Leaf 3, 2,  Leaf 4)
let t3 = Node (Leaf 3, 2, Node (Leaf 5, 4,
 Leaf 6))
```

Tipi di dato ricorsivi

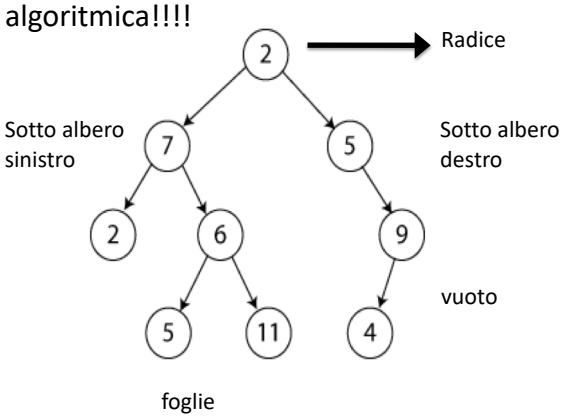
```
# type tree =
    | Leaf of int
    | Node of tree * int * tree;;

type tree = Leaf of int | Node of tree * int * tree
```

Quanti di voi hanno programmato con strutture dati del tipo *tree*?

Alberi binari

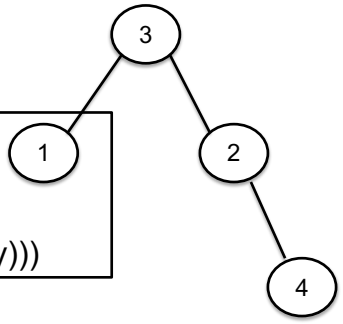
Li avete visti a algoritmica!!!!



Alberi binari in OCaml

```
type tree =
    | Empty
    | Node of tree * int * tree
```

```
let t : tree =
    Node(Node(Empty, 1, Empty), 3
            Node(Empty, 2,
                    Node(Empty, 4, Empty)))
```



Ricerca in un albero

```
let rec  contains (t : tree) (n : int) : bool =
   begin match t with
      | Empty -> false
      | Node(lt, x, rt) -> x = n ||
                           (contains lt n) ||
                           (contains rt n)
   end
```

La funzione contains effettua una ricerca del valore n sull'albero t
Caso peggiore: deve visitare tutto l'albero

Alberi binari di ricerca

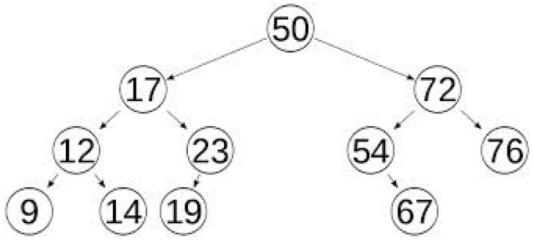
Idea: ordinare i dati sui quali viene fatta la ricerca

Un albero binario di ricerca (BST) è un albero binario
che deve soddisfare alcune proprietà invarianti addizionali

INVARIANTE DI RAPPRESENTAZIONE

- Node(lt, x, rt) è un BST se
  - lt e rt sono BST
  - tutti i nodi di lt contengono valori < x
  - tutti i nodi di rt contengono valori > x
- Empty (l'albero vuoto) è un BST

Esempio



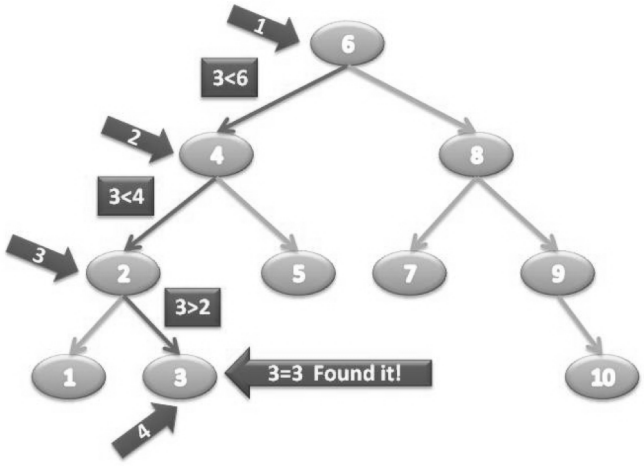
L'invariante di rappresentazione dei BST è soddisfatto

Ricerca su un BST

```
(* Ipotesi: t è un BST *)
let rec lookup (t : tree) (n : int) : bool =
   begin match t with
      | Empty -> false
      | Node(lt, x, rt) ->
         if x = n then true
         else if n < x then (lookup lt n)
               else (lookup rt n)
   end
```

Osservazione 1: L'invariante di rappresentazione guida la ricerca
Osservazione2: La ricerca può restituire valori non corretti se applicata a un albero che non soddisfa l'invariante di rappresentazione

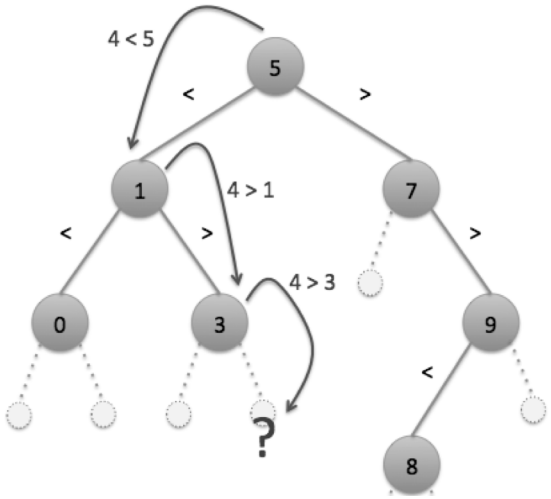
lookup(t, 3)



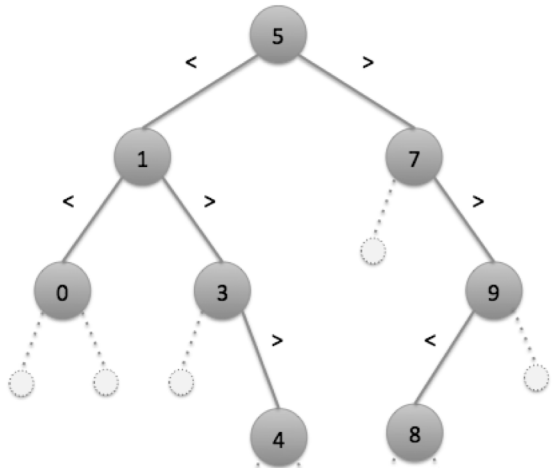
Come costruiamo un BST

- Opzione 1
  - costruiamo un albero e poi controlliamo (check) se vale l'invariante di rappresentazione
- Opzione 2
  - definire le funzioni che costruiscono BST a partire da BST (ad esempio, la funzione che inserisce un elemento in un BST e restituisce un BST)
  - definire una funzione che costruisce il BST vuoto
  - tutte queste funzioni soddisfano l'invariante di rappresentazione, pertanto "per costruzione" otteniamo un BST
  - non si deve effettuare nessun controllo a posteriori!!
  - questo passo mette in evidenza il ruolo della teoria in informatica (tipi algebrici): ne parleremo nel seguito del corso

insert(t, 4)



insert(t, 4)

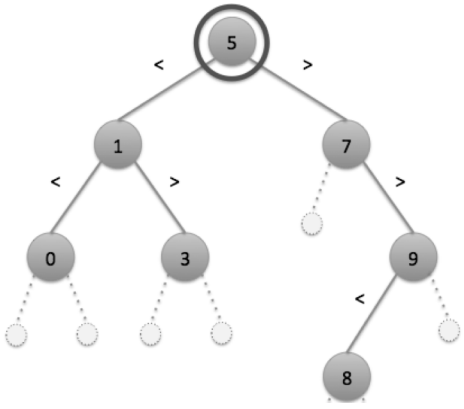


insert

```
(* Insert n nel BST t *)
let rec insert (t : tree) (n : int) : tree =
   begin match t with
      | Empty -> Node(Empty, n, Empty)
      | Node(lt, x, rt) ->
            if x = n then t
            else if n < x then Node(insert lt n, x, rt)
                          else Node(lt, x, insert rt n)
end
```

Per quale motivo l'albero costruito dalla funzione insert è un BST?

delete(t,5)



L'operazione di rimozione è più complicata: si deve promuovere la foglia 3 a radice dell'albero!!!

Funzione ausiliaria

```
let rec tree_max (t : tree) : int =
    begin match t with
      |Node(_, x, Empty) -> x
      |Node(_, _, rt) -> tree_max rt
      | _ -> failwith "tree_max called on Empty"
    end
```

L'invariante di rappresentazione garantisce che il valore max si trova nella parte più a destra dell'albero

delete

```
let rec delete (t : tree) (n : int) : tree =
   begin match t with
      | Empty -> Empty
      | Node(lt, x, rt) ->
         if x = n then
            begin match (lt, rt) with
               | (Empty, Empty) -> Empty
               | (Node _, Empty) -> lt
               | (Empty, Node _) -> rt
               | _ -> let m = tree_max lt in
                           Node(delete lt m, m, rt)
            end
         else if n < x then Node(delete lt n, x, rt)
                       else Node(lt, x, delete rt n)
end
```

Funzioni generiche

Analizziamo la funzione *length* applicata a *int list* e *string list*

```
let rec length (l : int list) : int =
   begin match l with
      | [] -> 0
      | _::tl -> 1 + length tl
   end
```

Le funzioni sono identiche, eccettuata l'annotazione di tipo

```
let rec length (l : string list) : int =
   begin match l with
      | [] -> 0
      | _::tl -> 1 + length tl
   end
```

Generici in OCaml

```
let rec length (l : 'a list) : int =
   begin match l with
      | [] -> 0
      | _::tl -> 1 + (length tl)
   end
```

La notazione 'a list indica una lista generica
length [1; 2; 3] applica la funzione a *int list*
length ["a"; "b"; "c"] applica la funzione a *string list*

Append generico

```
let rec append (l1 : 'a list) (l2 : 'a list) : 'a list =
  begin match l1 with
    | [] -> l2
    | h::tl -> h::(append tl l2)
  end
```

Pattern matching permette di operare su tipi generici
h ha tipo 'a
tl ha tipo 'a list

Generic zip

```
let rec zip (l1 : 'a list) (l2 : 'b list) : ('a * 'b) list =
  begin match (l1,l2) with
    | (h1::t1, h2::t2) -> (h1, h2)::(zip t1 t2)
    | _ -> []
  end
```

La funzione opera su tipi generici multipli
(da 'a list e 'b list verso ('a * 'b) list)

zip [1;2;3] ["a";"b";"c"] = [(1,"a");(2,"b");(3,"c")] : (int * string) list

Generic tree

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

Si noti l'utilizzo del parametro di tipo **'a**

Generic BST

```
let rec insert (t : 'a tree) (n : 'a) : 'a tree =
   begin match t with
     | Empty -> Node(Empty, n, Empty)
     | Node(lt, x, rt) ->
            if x = n then t
                  else if n < x then Node(insert lt n, x, rt)
                              else Node(lt, x, insert rt n)
   end
```

Gli operatori di relazione = e < operano su ogni tipo di dato

Collection (Set)

- Un insieme è una collezione di dati omogenei con operazioni di unione, intersezione, etc.
- Un Set è sostanzialmente una lista nella quale
  - la struttura d'ordine non è importante
  - non sono presenti duplicati

  ma non è un **tipo primitivo** in Ocaml
- Strutture dati come Set sono usate frequentemente in molte applicazioni
  - interrogazioni SQL (insieme degli studenti iscritti a Informatica, insieme dei risultati di una ricerca sul web con Google, insieme dei dati di un esperimento al CERN, …)
- Diversi modi per implementare Set

Set

- Un BST definisce una implementazione della struttura Set
  - *l'insieme vuoto (**bst empty**)*
  - *determinare tutti gli elementi che appartengono all'insieme (**visita dell'albero**)*
  - *definire una operazione per testare l'appartenenza di un elemento a un insieme (**lookup**)*
  - *definire unione e intersezione (tramite le operazioni di **insert e delete**)*

OCaml: Set Interface

```
module type Set = sig
  type 'a set
  val empty : 'a set
  val add : 'a -> 'a set -> ' a set
  val remove : 'a -> 'a set -> 'a set
  val list_to_set : 'a list -> 'a set
  val member : 'a -> 'a set -> bool
  val elements : 'a set -> 'a list
end
```

Idea (solita): fornire diverse funzionalità nascondendo la loro implementazione

Module type (in un file .mli) per dichiarare un TdA
sig ... end racchiudono una segnatura, che definisce il TdA e le operazioni
val: nome dei valori che devono essere definiti e dei loro tipi

Moduli in OCaml

Nome del modulo

Signature che deve essere implementata

```
module Myset : Set = struct …
  (* implementations of all the operations *)
  :
end
```

“dot notation”

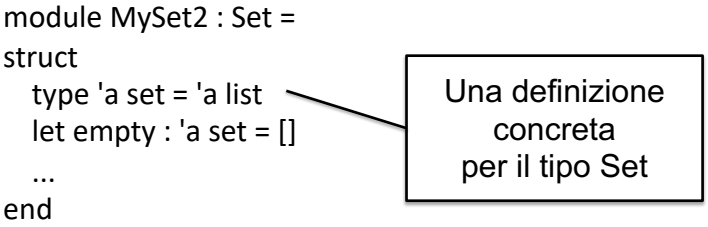
```
let s1 = Myset.add 3 Myset.empty
let s2 = Myset.add 4 Myset.empty
let s3 = Myset.add 4 s1
let test() : bool = (Myset.member 3 s1) = true
;; run_test "Myset.member 3 s1" test
let test() : bool = (Myset.member 4 s3) = true
 ;; run_test "Myset.member 4 s3" test
```

Open module

Alternativa: aprire lo scope del modulo (**open**) per portare i nomi nell’ambiente del programma in esecuzione

```
;; open Myset
let s1 = add 3 empty
let s2 = add 4 empty
let s3 = add 4 s1
let test() : bool = (member 3 s1) = true
;; run_test "Myset.member 3 s1" test
let test() : bool = (member 4 s3) = true
 ;; run_test "Myset.member 4 s3" test
```

Implementazione basata su list



Domande

open MySet
let s1 : int set = Empty

Supera la fase di controllo dei tipi?

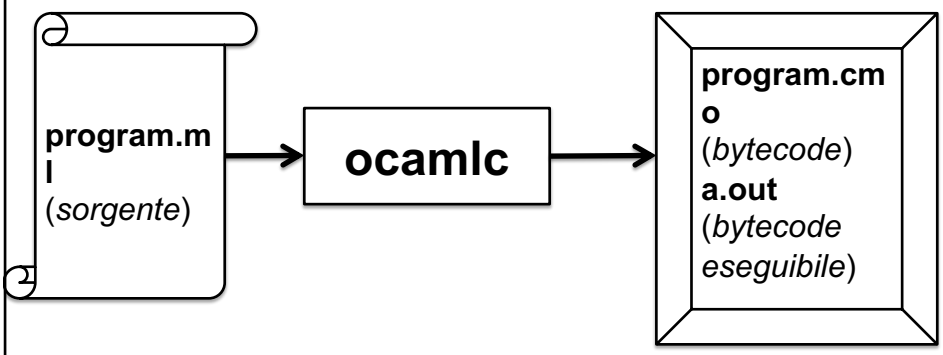
Domande

open MySet
let s1 : int set = Empty

Supera la fase di controllo dei tipi?

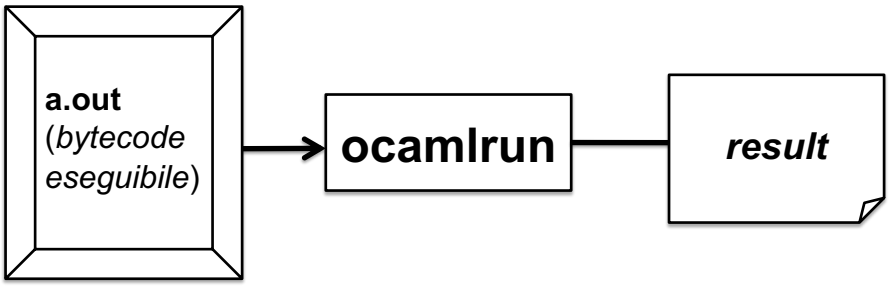
No: il costuttore Empty non è visibile esternamente al modulo!!!

Compilare programmi OCaml



http://caml.inria.fr/pub/docs/manual-ocaml/comp.html

Eseguire bytecode OCaml



http://caml.inria.fr/pub/docs/manual-ocaml-400/manual024.html