



OCAML UN BREVE RIPASSO

Value-oriented programming



- ✦ “value-oriented programming”: una espressione Ocaml denota un valore.
- ✦ L'esecuzione di un programma OCaml puo' essere vista come una sequenza di passi di calcolo (semplificazioni di espressioni) che alla fine produce un valore.

Valutazione di espressioni



$3 \Rightarrow 3$ (valori di base)

$3+4 \Rightarrow 7$

$2*(4+5) \Rightarrow 18$

La notazione $\langle \text{exp} \rangle \Rightarrow \langle \text{val} \rangle$ indica che la espressione $\langle \text{exp} \rangle$ quando eseguita calcola il valore $\langle \text{val} \rangle$

E le funzioni?



Per determinare il valore calcolato dall'applicazione di una funzione a certi argomenti, si deve prima calcolare il valore degli argomenti, poi si deve sostituire il valore degli argomenti nei parametri presenti nel corpo della funzione e poi valutare il corpo così' ottenuto.

let succ (n: int) : int = n + 1 in succ 5 \Rightarrow 6

succ (2 + 3)

\Rightarrow succ(5)

\Rightarrow (n + 1){5 -> n}

\Rightarrow (5 + 1)

\Rightarrow 6

Sostituzione del valore dell'argomento
nel corpo della funzione =
passaggio dei parametri
(per valore)

Il Let?

```
let x = 2 + 1 in
let y = x + x in
X*y
```

```
==>          dato che 2 + 1 ==> 3
```

```
let x = 3 in
let y = x + x in
X*y
```

```
==>          dato che x + x ==> 3 + 3 = 6
```

```
let x = 3 in
let y = 6 in
x * y
```

```
==>          dato che x * y ==> 3 * 6 ==> 18
18
```

Binding: associazione di un nome con un valore

Ambiente: un insieme di binding

Programmazione I: interprete denotazionale



Binding e funzioni

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

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

Binding nell'ambiente tra il nome della funzione f e la sua definizione



In sintesi



Una dichiarazione inserisce un legame nell'ambiente tra il nome dichiarato e l'espressione

Una dichiarazione puo' essere una dichiarazione di variabile e una dichiarazione di funzione

```
let x = 100
(* ambiente contiene il legame tra x e 100 *)
```

```
let f (k:int) : int = k * 5 + x
(* estendo l'ambiente con il legame tra f e l'espressione che ne
costituisce il corpo *)
```

```
let x = 1
let y = f 42
```

Programmare con i tipi di dato

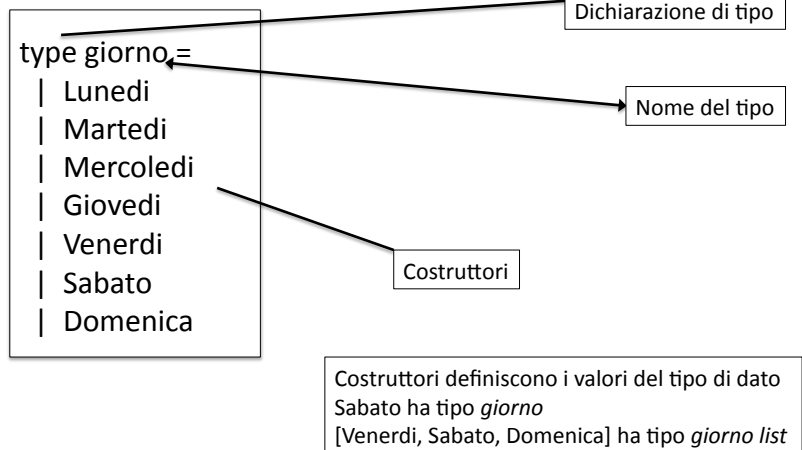


- ✎ I linguaggi di programmazione forniscono diversi costrutti primitivi per creare e manipolare dati strutturati.
- ✎ Cosa avete visto nel passato:
- ✎ Programmazione I:
 - *Tipi di dato primitivi* (int, string, bool, ...)
 - *Lists* (int list, string list, string list list, ...)
 - *Tuples* (int * int, int*string, ...)
- ✎ Algoritmica
 - tecniche algoritmiche (di base) per la risoluzione di problemi su vettori, liste, alberi e grafi, in modo efficiente in tempo e/o spazio

Definire tipi di dato in OCaml



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



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
| Lunedì -> "Lunedì"
| Martedì -> "Martedì"
| :
| :
| 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
 - Lunedì = 1, Martedì=2, ..., Domenica = 8
- ✎ Ma
 - Il tipo di dato primitivo *int* fornisce un insieme di operazioni differenti da quelle significative sul tipo di dato *giorno*, Mercoledì – 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
```

Valori del tipo *foo*

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

Pattern Matching



```
let get_count (f: foo) : int =
  begin match f with
  | Noting -> 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

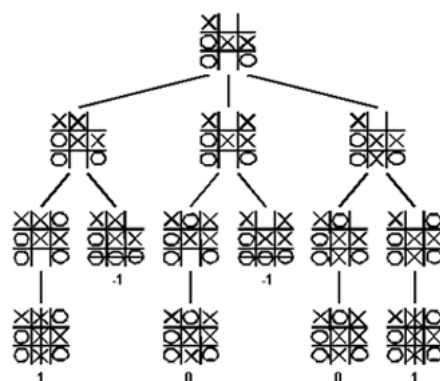


```
# type tree =
| Leaf of int
| Node of tree * int * tree;;
```

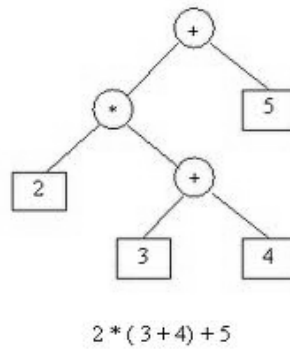
```
type tree = Leaf of int | Node of tree * int * tree
```

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

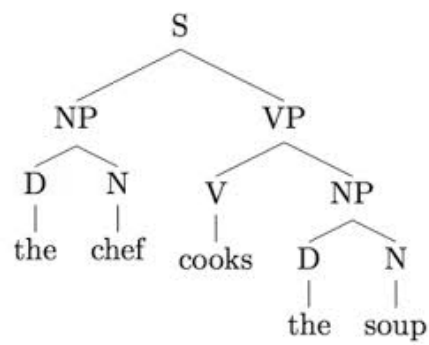
Game Tree



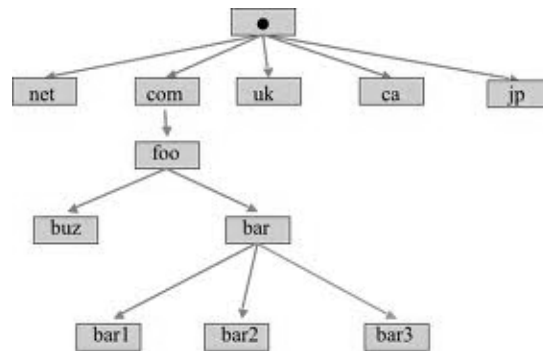
Expression Trees



(Natural Language) Parse Tree



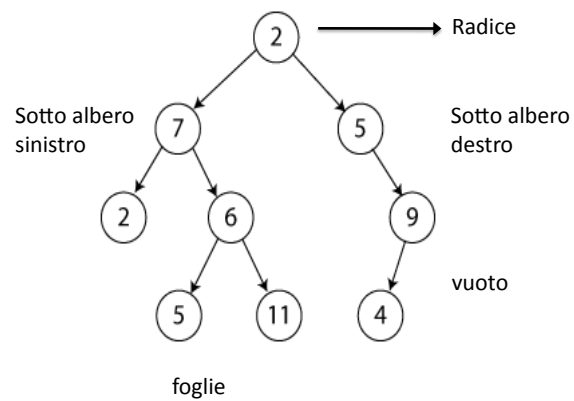
Domain Names (DNS)



Alberi Binari



Li avete visti a algoritmica!!!!

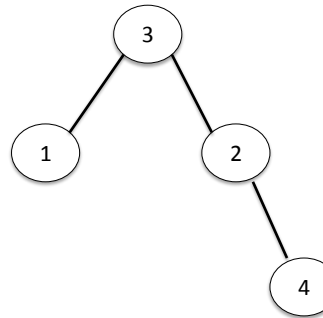


Alberi Binari in OCaml

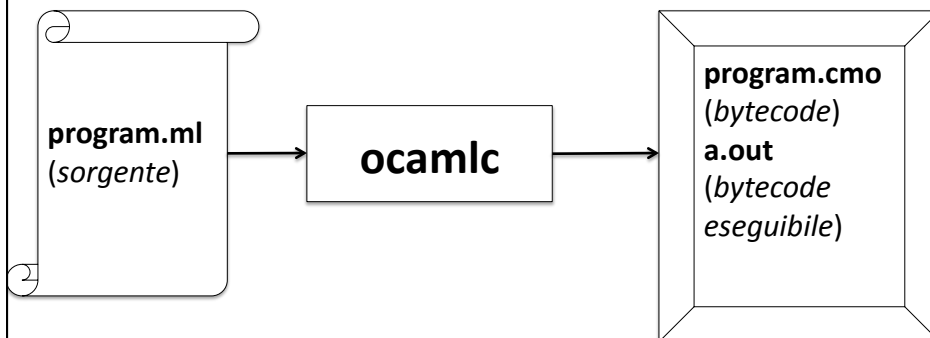


```
Type tree =
  | Empty
  | Node of tree * int * tree
```

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

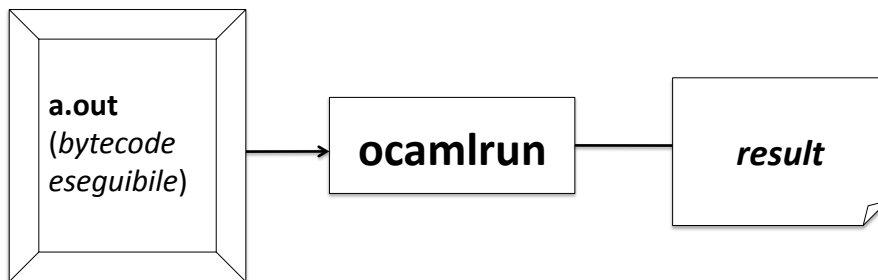


Compilare programmi OCaml



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

Eeguire bytecode OCaml



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

Esempi



trees.ml
treeExamples.ml

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 su cui viene fatta la ricerca

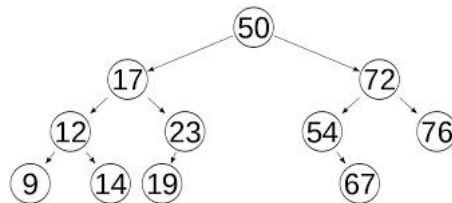
Un albero binario di ricerca (BST) e' un albero binario che deve soddisfare

Alcune proprietà invarianti addizionali

INVARIANTE DI RAPPRESENTAZIONE

- Node(lt, x, rt) e' 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) e' un BST

Esempio



L'invariante di rappresentazione dei BST e' soddisfatto.
 Come di dimostra?
 Ricordate le tecniche che avete imparato a LPP!!!

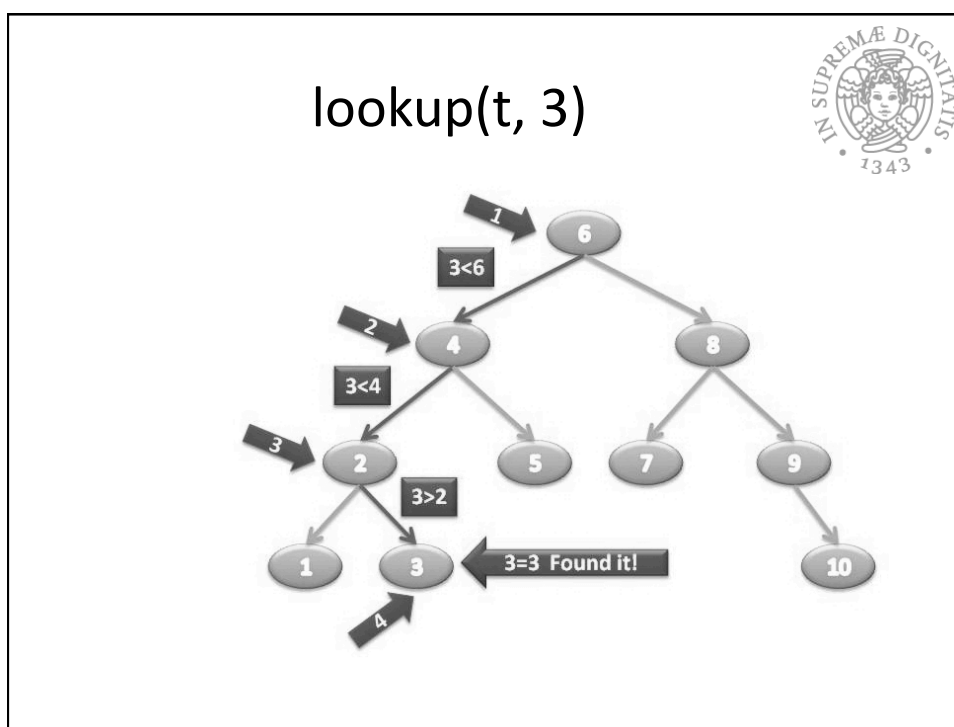
Ricerca su un BST



```

(* Ipotesi t e' 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
Osservazione 2: La ricerca potrebbe restituire valori non corretti se applicata a un albero che non soddisfa l'invariante di rappresentazione



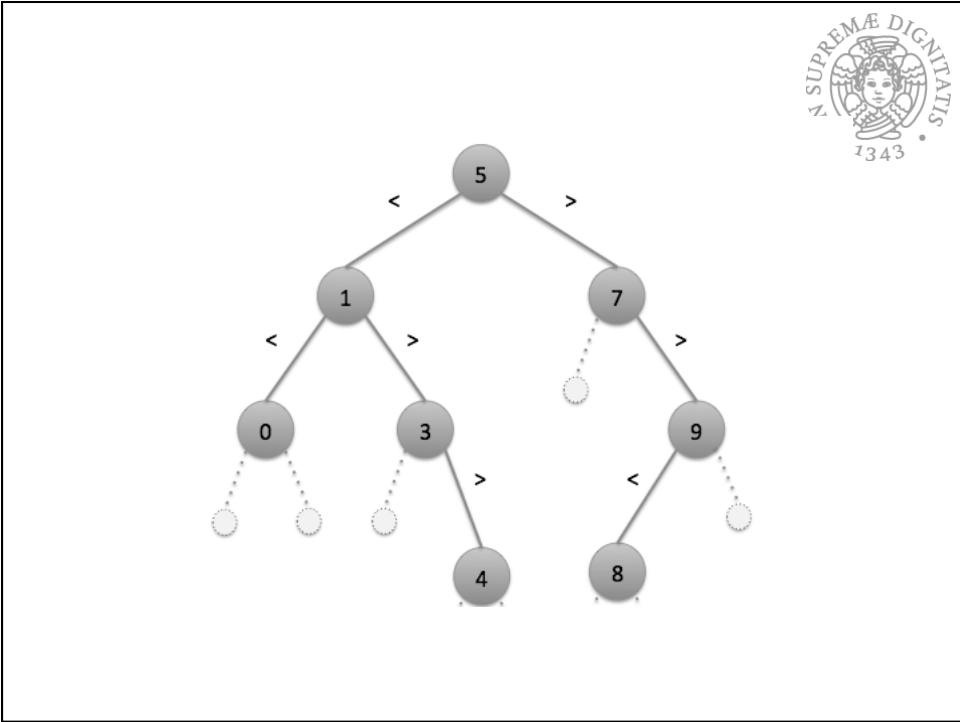
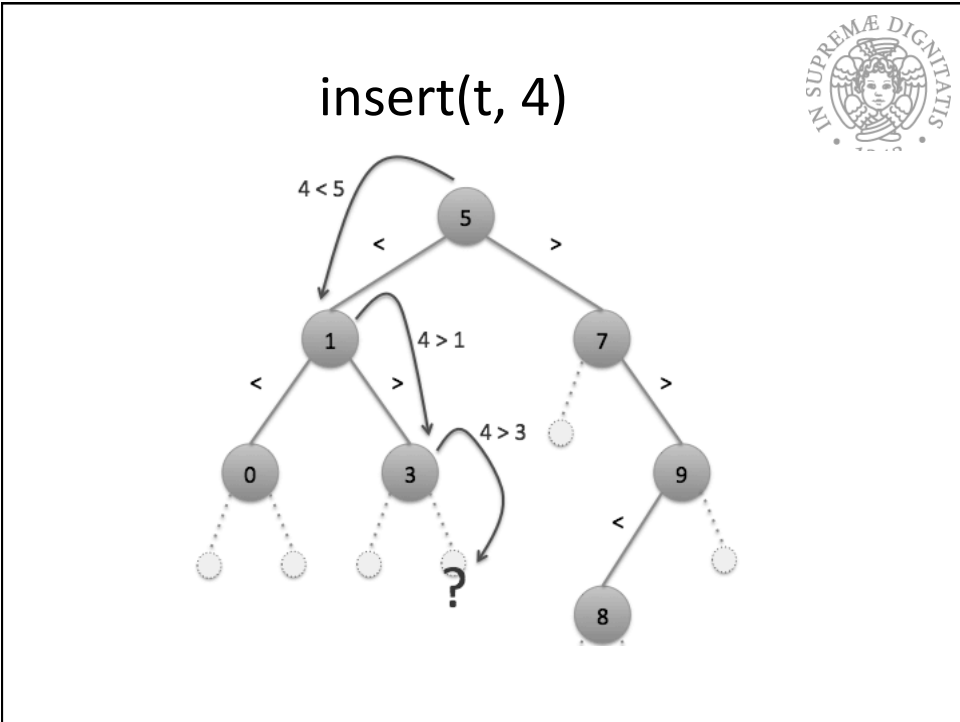
Come costruiamo un BST

Opzione 1:

- Costruiamo un albero e poi controlliamo (check) che vale l'invariante di rappresentazione

Opzione 2:

- Definire le funzioni che costruiscono BST a partire da BST (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.



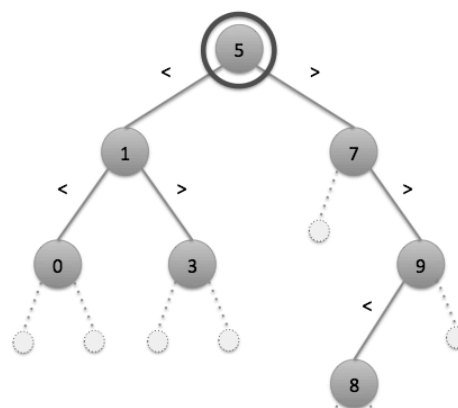
insert



```
(* 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 e' un BST?

Delete(t,5)



Operazione di rimozione e' piu' 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 piu' 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
```



GENERICI



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
```

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

Le funzioni sono identiche
eccettuato per l'annotazione
di tipo

Generici in OCaml



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

La notazione `'a list` viene usata per indicare 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
(`'a list` e `'b list`, (`'a * 'b`) list)

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

Generic trees



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

Notare l'uso 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



ABSTRACT COLLECTIONS

Insiemi (Sets)



- ✎ Un insieme (set) e' una collezione di dati omogenei con operazioni di unione, intersezione, etc, etc.
- ✎ Un Set e' sostanzialmente un lista in cui
 - La struttura d'ordine non e' importante
 - Duplicati non sono presenti
 - Ma non e' 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, l'insieme dei dati di un esperimento al CERN, ...

Sets



- ✎ 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)*

Abstract Sets



- ✎ Un tipo di dato astratto e' definito dalla sua interfaccia di uso (le operazioni) e le proprieta' di queste operazioni
- ✎ Abstract Set: interface
 - Esiste l'insieme vuoto
 - Esiste la possibilita' di aggiungere un elemento a un insieme ottenendo un insieme piu' grande,
 - Esiste la possibilita' di rimuovere un elemento da un insieme,
 - Possiamo testare l'appartenenza di un elemento all'insieme.
- ✎ Abstract Set: proprieta'
 - Gli elementi inseriti in uninsieme appartengono all'insieme (e quindi possono essere trovati nell'insieme)
 - Aggiungere una seconda volta un elemento non modifica la struttura dell'insieme.
 -

Progettare e programmare



- ✎ Il problema: In qualita' di docente del corso di laurea Andrea Corradini ha il compito di leggere il syllabus degli insegnamenti e di determinare quali tra le parole del syllabus di un insegnamento appartengono al vocabolario delle parole chiave del dizionario ICT del 2014.
- ✎ Una specie di tag-cloud
- ✎ Possiamo aiutarlo a farlo in modo automatico?

Astrazioni e programmazione



- ✎ Quali sono i concetti maggiormente significativi, **le astrazioni**, per definire una soluzione di questo problema?
- ✎ L'insieme delle parole che appaiono in un syllabus
- ✎ L'insieme delle parole del vocabolario ICT 2014
- ✎ L'insieme delle parole del syllabus che compaiono nel vocabolario ICT

Procediamo alla soluzione



- ✎ Supponiamo di avere il tipo generico degli insiemi: 'a set
 - Dettagli nel seguito
- ✎ Definiamo l'interfaccia del nostro problema:
- ✎ `let countWord (textSyllabus : string list)
 (vocabICT : string set) : int =
 failwith "Parliamone!!"`

Definire i casi di test



```
let vocabICT : string set = list_to_set ["Java"; "Cloud"; "Social-Networks";
    "BigData"; "Web-Services"; "e-commerce"; "Macchine-Virtuali"]
```

let PR2Syllabus= [Il corso di programmazione si propone di illustrare le caratteristiche principali dei linguaggi di programmazione e le loro strutture di implementazione in termine di macchine-virtuali. il corso si propone presentare e discutere le tecniche per la programmazione a oggetti esemplificate e sperimentate utilizzando il linguaggio Java] (*visto come lista di stringhe*)

```
let test () : bool =
    countWord PR2Syllabus vocabICT = 2 ;;
    run_test "countWord" test!
```

Implementazione



- ✦ L'implementazione di countWord e' banale (lasciata come esercizio)
- ✦ Facile vedere che richiede una funzionalita' ulteriore: il test di appartenenza
- ✦

```
let member (x:'a) (s:'a set) : bool =
    failwith "unimplemented"
```



COMPONIAMO LE DIVERSE ASTRAZIONI CHE ABBIAMO ESAMINATO



OCaml: Set Interface

```
module type Set = sig
  type 'a set
  val empty : 'a set
  val add : 'a->'aset->'aset
  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
```

Val: nome dei valori che devono essere definiti
e dei loro tipi

Moduli in OCaml



Nome del Modulo

Signatura 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
```

Implementare astrazioni



- ✎ Abbiamo molte possibilità per scegliere la struttura di implementazione degli insiemi
 - list, tree, array, etc.
- ✎ Come scegliamo la struttura di implementazione?
- ✎ Ricordiamo che un "insieme e' una struttura soggetta a alcuni invarianti di rappresentazione."
- ✎ Esempi:
 - Un insieme e' implementato con una *lista senza elementi ripetuti*
 - Un insieme e' implementato con un *albero senza elementi ripetuti*
 - Un insieme e' implementato con un *array di bit 0 = non appartiene, 1 = appartiene*
- ✎ **Il punto importante e' rispettare l'invariante di rappresentazione?**

Tipi di dato astratti



- ✎ Un tipo di dato astratto (ADT Abstract Data Type) è un tipo di dato le cui istanze possono essere manipolate con modalità che dipendono esclusivamente dalla semantica del dato e non dalla sua implementazione.
- ✎ Idea: Hide the *concrete representation* of a type behind an *abstract interface* to preserve invariants (Barbara Liskov)
- ✎ Lettura utile: http://en.wikipedia.org/wiki/Abstract_data_type

ADT



- ✎ La nozione di interfaccia (interface) restringe le modalità di interazione del tipo di dato con le altre parti che compongono il programma.
- ✎ Vantaggi:
 - **Safety**: gli utenti del tipo di dato non possono rompere l'invariante di rappresentazione
 - **Modularità**: E' possibile modificare l'implementazione del tipo di dato senza dover modificare l'implementazione dei programmi che lo utilizzano.

ADT: Implementazione



```

module MySet : Set =
struct
  type 'a tree =
  | Empty

  | Node of 'a tree * 'a * 'a tree
  type 'a set = 'a tree
  let empty : 'a set = Empty
  ...
end

```

1. L'implementazione del ADT Set deve includere l'implementazione dell'operazioni descritte nell'interfaccia. Può contenere anche altre operazioni (ausiliarie) che non possono essere usate da fuori il modulo.
2. I tipi dell'implementazione devono essere consistenti con quelli definiti nell'interfaccia.

Implementazione basata su list



```

module MySet2 : Set =
struct
  type 'a set = 'a list
  let empty : 'a set = []
  ...
end

```

Una definizione
Differente
per il tipo set

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?

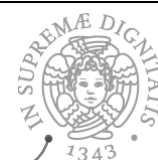
No: il costruttore Empty non è visibile
Esternamente al modulo!!!

Sperimentazione



- ✦ Il file Set.ml contiene la definizione del ADT Set e la sua implementazione come lista e come albero.
- ✦ Definire una implementazione alternativa basata su array.

Correttezza del software



- ✦ Cosa vuole dire il software e' corretto?
- ✦ Una componente software (programma, oggetto ...) è corretto se "fornisce il comportamento per cui è stato pensato"
 - è necessario descrivere la "specifica" della componente software
 - Come si verifica la correttezza di una componente software
- ✦ Diverse strategie per verificare la correttezza (effettuare il test) di una componente software rispetto alla sua specifica
 - approccio formale: la correttezza del programma viene dimostrata
 - approccio pragmatico: la correttezza del programma viene verificata sperimentalmente