DEFINIZIONE di NUOVI TIPI

Titolo nota 22/10/2015

ENUMERAZIONE

```
# type giorno = Lun | Mar | Mer | Gio | Ven | Sab | Dom ;;
  type giorno defined

# Gio ;;
- : giorno = Gio

# 3;;
- : int = 3
```

L'ordine di enumerazione induce <

```
# Gio < Sab ;;
  - : bool = true

# Sab < Gio;;
  - : bool = false
```

```
# let feriale x = x < Sab ;;
```

feriale : giorno ⟶ bool = <fun>

(giorno = x, bool = ris)

```
# feriale Lun ;;
  - : bool = true

# feriale Dom ;;
  - : bool = false
```

Titolo nota 22/10/2015

DEFINIZIONE mediante costruttori e costanti

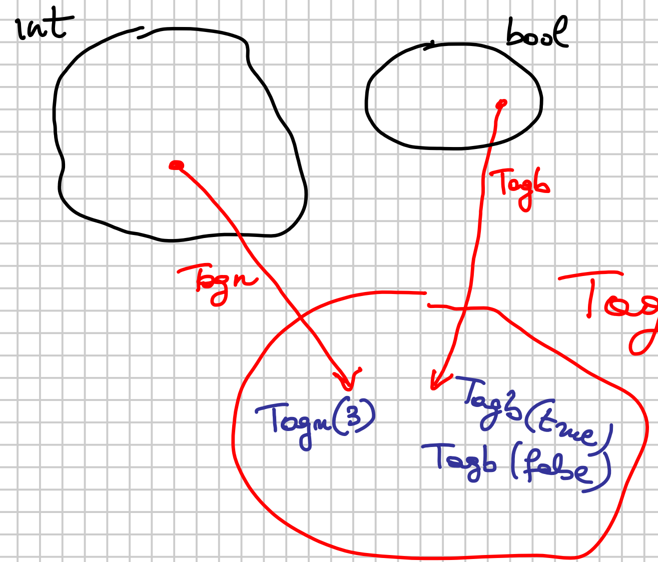
```
type tag = Tagn of int | Tagb of bool ;;
```

costruttori di valori

```
type tag defined

# 3 ;;
-: int = 3

# Tagn 3 ;;
-: tag = Tagn 3
```

valore di tipo tag perché costruito attraverso il costruttore Tagn

```
# Tagn;;
-: int -> tag = <fun>

# Tagb;;
-: bool -> tag = <fun>
```



TIPI POLIMORFI

Titolo nota 22/10/2015

```
type 'a foo = Foo of 'a ;;
```

'a foo → tipo polimorfo — 'a è una variabile di tipo

Foo → costruttore di valori

```
# Foo 3 ;;
- : int foo = Foo 3

# Foo true ;;
- : bool foo = Foo true

# Foo ;;
- : 'a -> 'a foo = <fun>
```

TIPI RICORSIVI

Titolo nota 08/10/2015

Definire il tipo dei naturali

```
type nat = Zero | Succ of nat ;;
type nat defined

# Zero ;;
- : nat = Zero

# Succ (Zero) ;;
- : nat = Succ Zero

# Succ (Succ (Succ Zero)) ;;
- : nat = Succ (Succ (Succ Zero))
```

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

let rec val n = match n with
    Zero -> 0
  | Succ m -> (val m) + 1 ;;
```

ESEMPIO : rappresentazione della "sintassi" del linguaggio imperativo

Titolo nota 08/10/2015

Num → Zero | Succ (Num)

Exp → Num | Exp Op Exp

Op → + | *

V = { Num, Exp, Op }

Λ = { Zero, Succ, +, * }

S = Exp

```
type Num = . . . .
type Exp = num of Num | exp of Exp * Op * Exp
```

(Exp * Op * Exp: prodotto cartesiano)

```
type Op = Sum | Mul ;;
```

3 + 4 * 2

```
# exp (num (Succ (Succ (Succ Zero))) ,
       Sum ,
       exp (Succ (Succ (Succ (Succ Zero)))) ,
            Mul ,
            Succ (Succ Zero)
           )
    )
```

"SEMANTICA" di EXP

Titolo nota 08/10/2015

```
let rec valnum n =
    match n with
  Zero -> 0
 Succ m -> (valnum m) + 1;;

valnum : Num -> int = <fun>

let valop op = match op with
    Sum -> (prefix +)
  | Mul -> (prefix * ;;)

valop : Op -> int -> int -> int = <fun>
```

Sem_{exp} n ρ μ = val n

```
let rec valexp e = match e with
   num x -> valnum x
 | exp (e1, op, e2) ->
     (valop op) (valexp e1) (valexp e2);;

valexp : Exp -> int = <fun>
```

ALBERO BINARIO

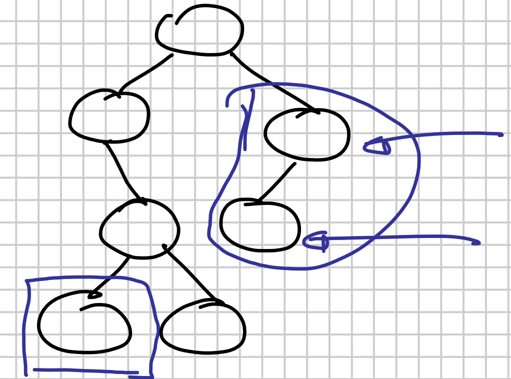
Titolo nota 08/10/2015

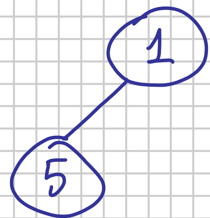
Sono alberi in cui ogni nodo (non foglia) ha al più 2 figli

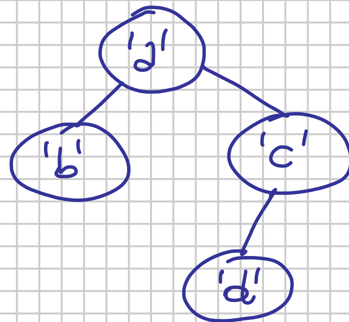
Definizione induttiva di albero binario

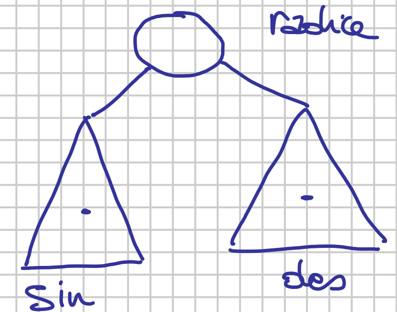
1. l'albero vuoto è un albero binario

2. Un albero binario non vuoto è costituito da un nodo radice e da due sottoalberi (destro e sinistro) che sono a loro volta alberi binari





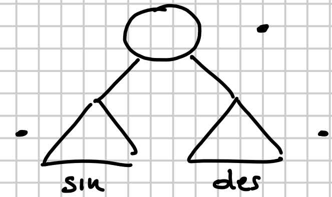




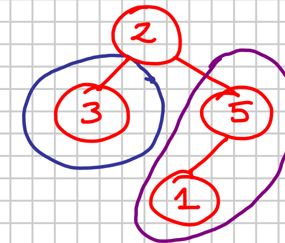
ALBERI BINARI POLIMORFI


```
type 'a btree = Void | Node of 'a * 'a btree * 'a btree ;;
```

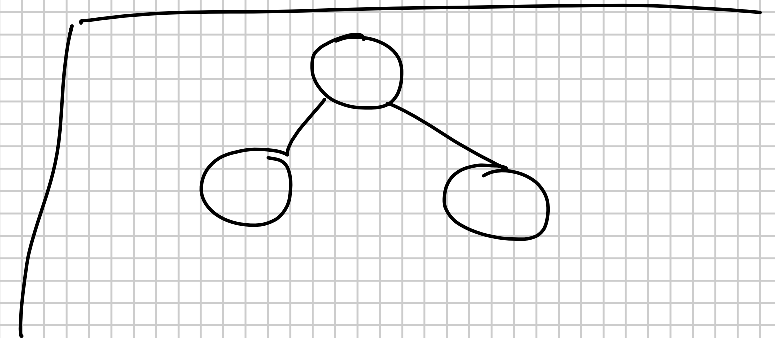
Void → costante; Node → costruttore di tipo




```
type 'a btree defined

# Node (2, Node (3, Void, Void),
         Node (5, Node (1, Void, Void), Void)) ;;

-: int btree = ..........
```

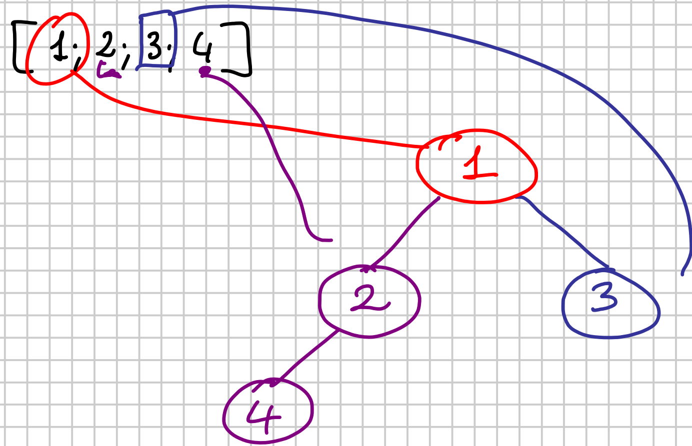


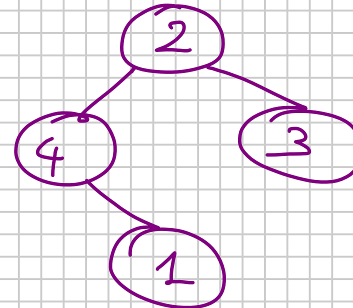

```
# Node ((2, 'a'), Void, Void) ;;

-: int * char btree = .....
```



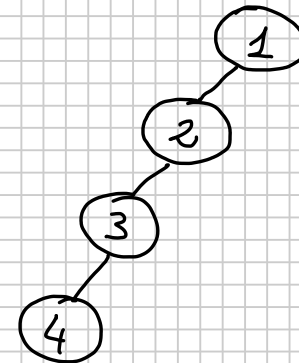
COSTRUZIONE di un albero da una LISTA di VALORI

Titolo nota 08/10/2015






```
let rec buildtree l = match l with
      []     ->    Void
    | x::xs  ->    Node (x , buildtree xs, Void);;

buildtree : 'a list ---> 'a btree = <fun>
```



Titolo nota 08/10/2015

① data una lista non vuota x::xs costruire un albero che ha come radice x e come sottoalberi destro e sinistro due alberi costruiti ricorsivamente a partire da due liste l1 e l2 tali che l1 e l2 sono uno "split" di xs

```
let rec buildtree l =
    let rec split l = match l with
        []  ->  [], []
      | [x] ->  [x], []
      | x::y::ys -> let split ys = l1, l2
                    in
                    x::l1 , y::l2
  in match l with
    [] -> Void   | x::xs -> let l1,l2 = split xs in Node (x, buildtree l1, buildtree l2);;
```

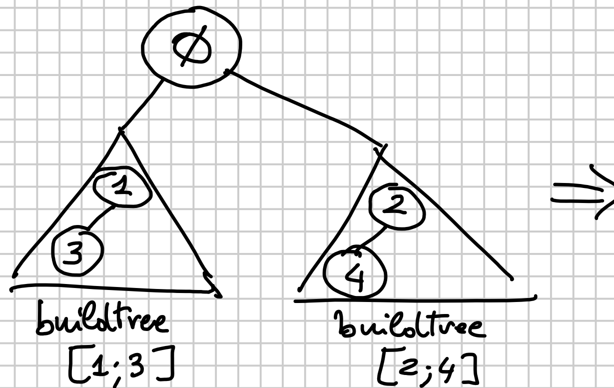
split [1;2;3;4]

= { 3° pattern, x = 1, y = 2, ys = [3;4] }

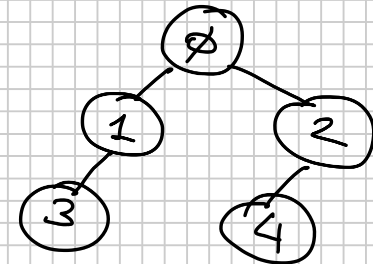
[1;3], [2;4]

buildtree [0;1;2;3;4]

```
let rec split l = match l with
    [] -> [], [] |
    [x] -> [x], [] |
  x::y::ys -> let l1, l2 = split ys
              in x::l1, y::l2
```

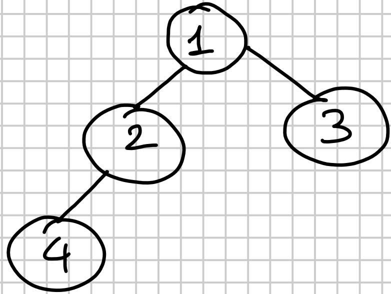


⇒



buildtree [1;2;3;4]

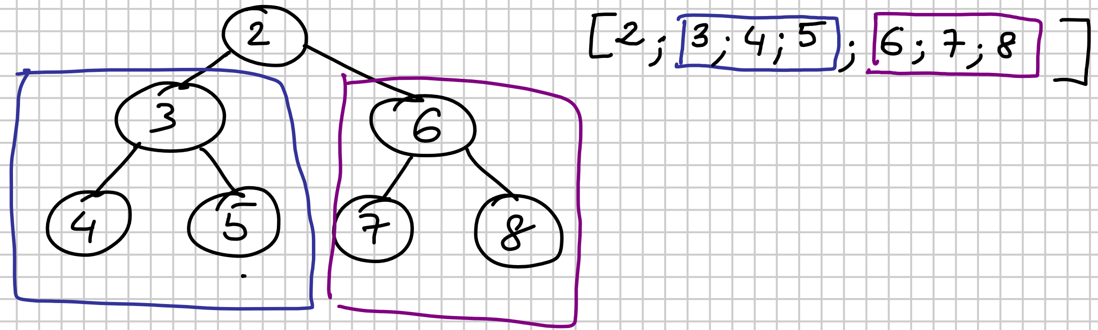
split [2;3;4] = [2;4], [3]



LINEARIZZAZIONE di UN ALBERO

Dato un albero binario, costruire una lista di valori con tutti i valori nei nodi dell'albero.

LINEARIZZAZIONE anticipata
- si prende la radice
- linearizzare il sottoalbero sin
- " " destro



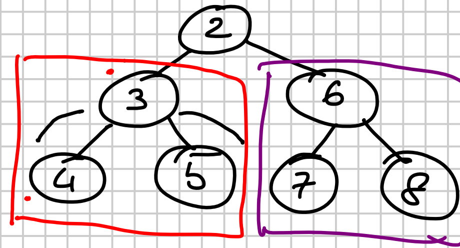

```
let rec alin bt = match bt with
    Void -> []
  | Node (x, lt, rt) -> x :: ((alin lt) @ (alin rt)) ;;
```

alin : 'a btree -> 'a list = <fun>

('a btree = bt, 'a list = ns)

LINEARIZZAZIONE DIFFERITA

- prima si linearizza il sottoalbero sin
- " " " destro
- si prende la radice



[4; 5; 3; 7; 8; 6; 2]

```
let rec dlin bt = match bt with
      Void -> []
    | Node (x, lt, rt) -> (dlin lt)@(dlin rt)@[x] ;;
```

```
let rec slin bt = match bt with
      Void -> []
    Node (x, lt, rt) -> (slin lt)@ (x :: (slin rt)) ;;
```

[4; 3; 5; 2; 7; 6; 8]