Utility Programmazione Funzionale

by Daniele "Palla" Palladino

Alberi

```
(*Tr: 'a * 'a tree * 'a tree -> 'a tree *)
type 'a tree = Empty
                | Tr of 'a * 'a tree * 'a tree

Alcune funzioni Base
(*is empty : 'a tree -> bool*)
let is_empty = function
    Empty -> true
    | _ -> false;;

(*root : 'a tree -> 'a*)
exception EmptyTree;;
let root = function
    Empty -> raise EmptyTree
    | Tr(x,_,_) -> x;;

(*left e right : 'a tree -> 'a tree*)
let left = function
     Empty -> raise EmptyTree
    | Tr(_,t,_) -> t;;

let right = function
     Empty -> raise EmptyTree
    | Tr(_,_,t) -> t;;

(*leaf : 'a -> 'a tree*)
let leaf x = Tr(x,Empty,Empty);;

(*is leaf : 'a tree -> bool*)
let is_leaf = function
     Tr(_,Empty,Empty) -> true
    | _ -> false;;

(*size : 'a tree -> int *)
let rec size = function
     Empty -> 0
     | Tr(_,t1,t2) -> 1 + size t1 + size t2;;

(*height : 'a tree -> int*)
let rec height = function
    Empty -> 0
    | Tr(_,t1,t2) -> 1 + max (height t1) (height t2);;

(*balanced : 'a tree -> bool*)
let rec balanced = function
    Empty -> true
    | Tr(_,t1,t2) -> balanced t1 && balanced t2 && abs(height t1 - height t2) <= 1;;

Visite
(*'a tree -> 'a list*)
Visita in preordine
let rec preorder = function
    Empty -> []
    | Tr(x,t1,t2) -> x::(preorder t1 @ preorder t2)
Visita simmetrica
let rec inorder = function
```

```
    Empty -> []
    | Tr(x,t1,t2) -> (inorder t1) @ (x::(inorder t2))
Visita in postordine
let rec postorder = function
    Empty -> []
    | Tr(x,t1,t2) -> (postorder t1) @ ((postorder t2) @ [x])

Ricerca di un ramo (se esite) con backtracking
exception NotFound;;
(* path_to : 'a -> 'a tree -> 'a list *)
let rec path_to x = function
    Empty -> raise NotFound
    | Tr(y,Empty,Empty) -> if y=x then [x]
                           else raise NotFound
| Tr(y,t1,t2) -> y::(try path_to x t1
                     with NotFound -> path_to x t2)
```

Grafi

```
type 'a graph = Gr of ('a * 'a) list

funzioni base
successori
(* succ : 'a graph -> 'a -> 'a list *)
let succ (Gr arcs) node =
    let rec aux = function
        [] -> []
        | (x,y)::rest -> if x = node then y::aux rest
                         else aux rest
in aux arcs;;

Successori di un grafo non orientato
(* succ : 'a graph -> 'a -> 'a list *)
let rec succ_in_nog (Gr arcs) node =
    let rec aux = function
        [] -> []
        | (x,y)::rest -> if x = node then y::aux rest
                         else if y = node then x::aux rest
                              else aux rest
in aux arcs;;

Visita in profondità
(* depth_first_collect : 'a graph -> 'a -> 'a list *)
let depth_first_collect graph start =
    let rec search visited = function
        [] -> visited
        | n::rest -> if List.mem n visited then search visited rest
                     else search (n::visited) ((succ graph n) @ rest)
                     (* i nuovi nodi sono inseriti in testa *)
in search [] [start];;

Visita in ampiezza
(* breadth_first_collect : 'a graph -> 'a -> 'a list *)
let breadth_first_collect graph start =
    let rec search visited = function
        [] -> visited
        | n::rest -> if List.mem n visited then search visited rest
                     else search (n::visited) (rest @ (succ graph n))
```

```
                    (* i nuovi nodi sono inseriti in coda *)
in search [] [start];;

ricerca di un path
(* search_path : 'a graph -> ('a -> bool) -> 'a -> 'a list *)
let search_path graph p start =
   let rec from_node visited a =
      if List.mem a visited then raise NotFound
      else if p a then [a]
      (* il cammino e' trovato *)
            else a::from_list (a::visited) (succ graph a)
   and from_list visited = function
      [] -> raise NotFound
      | a::rest -> try from_node visited a
                   with NotFound -> from_list visited rest
in from_node [] start;;

Ricerca di un cammino mediante backtracking
(* search_path : 'a graph -> ('a -> bool) -> 'a -> 'a list *)
let search_path (Graph succ) p start =
   (* from_node cerca un cammino a partire dal nodo a *)
   let rec from_node visited a =
      if List.mem a visited then raise NotFound
      else if p a then [a] (* il cammino e' trovato *)
            else (* cerca un cammino da uno dei successori di a *)
                a::from_list (a::visited)(succ a)
                (* from_list cerca un cammino che parta da uno dei nodi
                   della lista suo argomento *)
   and from_list visited = function
      [] -> raise NotFound
      | a::rest -> try from_node visited a
                   with NotFound -> from_list visited rest
in from_node [] start;;

#use "List.ml";;
      Objective Caml version 3.10.0
#
val length_aux : int -> 'a list -> int = <fun>
val length : 'a list -> int = <fun>
val hd : 'a list -> 'a = <fun>
val tl : 'a list -> 'a list = <fun>
val nth : 'a list -> int -> 'a = <fun>
val append : 'a list -> 'a list -> 'a list = <fun>
val rev_append : 'a list -> 'a list -> 'a list = <fun>
val rev : 'a list -> 'a list = <fun>
val flatten : 'a list list -> 'a list = <fun>
val concat : 'a list list -> 'a list = <fun>
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
val rev_map : ('a -> 'b) -> 'a list -> 'b list = <fun>
val iter : ('a -> 'b) -> 'a list -> unit = <fun>
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>
val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list = <fun>
val iter2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> unit = <fun>
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a =
  <fun>
val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c =
```

```
  <fun>
val for_all : ('a -> bool) -> 'a list -> bool = <fun>
val exists : ('a -> bool) -> 'a list -> bool = <fun>
val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool = <fun>
val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool = <fun>
val mem : 'a -> 'a list -> bool = <fun>
val memq : 'a -> 'a list -> bool = <fun>
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
val assq : 'a -> ('a * 'b) list -> 'b = <fun>
val mem_assoc : 'a -> ('a * 'b) list -> bool = <fun>
val mem_assq : 'a -> ('a * 'b) list -> bool = <fun>
val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list = <fun>
val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list = <fun>
val find : ('a -> bool) -> 'a list -> 'a = <fun>
val find_all : ('a -> bool) -> 'a list -> 'a list = <fun>
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list = <fun>
val split : ('a * 'b) list -> 'a list * 'b list = <fun>
val combine : 'a list -> 'b list -> ('a * 'b) list = <fun>
val merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list = <fun>
val chop : int -> 'a list -> 'a list = <fun>
val stable_sort : ('a -> 'a -> int) -> 'a list -> 'a list = <fun>
val sort : ('a -> 'a -> int) -> 'a list -> 'a list = <fun>
val fast_sort : ('a -> 'a -> int) -> 'a list -> 'a list = <fun>
#
```