

Esercizio A1

Specificare per la funzione o l'espressione il tipo più generale, oppure il valore, oppure "NT" per "non è tipizzabile".

```
print_int ;;
List.map print_string ;;
List.iter print_int ;;
(fun x -> x) 12 ;;
(fun x y -> x y) ;;
let f1 x = [x ; x] ;;
let f2 x y = (x @ (y x)) ;;
let f3 x = List.map x ;;
Array.create 4 3;;
if false then true else false ;;
let f4 x = Array.create (List.length x) ;;
Hashtbl.create 12 ;;
[| 'a' ; 'b' ; 'c' |].(2) ;;
int_of_char ;;
List.map ;;
List.map int_of_char ;;
List.filter char_of_int ;;
let eq (x, y) = x = y ;;
let neq1 (x, y) = not (eq (x, y)) ;;
let neq2 x y = not (eq (x y)) ;;
List.map eq ;;
List.filter eq ;;
Array.create 3 'a' ;;
Array.init 5
  (fun x -> char_of_int (x + (int_of_char 'a')))) ;;
Array.map (fun x -> (x > 5)) [| 1; 4; 9; 16; 25 |] ;;
float_of_int ;;
List.iter ;;
Array.map ;;
Array.map float_of_int ;;
let ap1 x y z = (x (y z)) ;;
```

```

let ap2 x y z = ((x y) z) ;;

let f1 x y = if x = y then ((y, x), x)
              else ((x, y), x) ;;

let f2 x y = (y, x, x) ;;

let f3 x y = [[x; x]; y] ;;

open out ;;

List.map open in ;;

List.map print_string ;;

List.iter open out ;;

(1, '2', 3) :: (1, '3', 2) :: [] ;;

List.map string_of_int ;;

let f x = Some(x) = None ;;

let f x y z = (x = (y = z)) ;;

(true < false) < false ;;

((1, 2), 3) = (3, 2, 1) ;;

[| 'a' ; 'b' ; '3' |] ;;

Array.iter print_string ;;

List.iter (fun x -> (print_int x; print_int x)) ;;

let f x y = ((x y), (y, x)) ;;

List.map (fun x -> (x, x)) [ '1' ; '2' ] ;;

Hashtbl.add ;;

(fun x -> (fun y -> (fun z -> (x (x z))))) ;;

(fun (x, y) -> (y x)) ;;

((fun x y -> (x, x, y)) 1) 'a' ;;

List.map print_string ;;

List.map (fun x -> [(x,x)]) [1;2] ;;

[| "a" ; "A" ; "a" |];;

Hashtbl.find ;;

[| None; Some(2) |] ;;

let f x y = Hashtbl.add x x y ;;

let x = "aa" in (print_string x; print_string x) ;;

let g x = if x > 0 then 1 else 1 + g x ;;

```

```
(fun x -> (fun y -> (fun x -> (x, y, x)))) ;;
(fun x -> ([1, 2] ; (x 'a')))) ;;
(1 ; 2 ; 3 ; 'a') ;;
None ;;
Some('a') ;;
List.iter ;;
(fun x -> (x :: [x])) ;;
let f1 x = (x :: x) ;;
let f2 x = (x @ x) ;;
let f3 x = List.map (Array.create x) ;;
let f4 x = (List.map Array.create) x ;;
let f5 x y z = [x; (y, y)] @ z ;;
"abcdef".[3];;
let f6 x y = x.[y] ;;
Array.init 5 (fun x -> ("abcdef".[x])) ;;
List.iter print char ;;
print int ;;
List.map print string ;;
List.iter print int ;;
(fun x -> x) 12 ;;
(fun x y -> x y) ;;
let f1 x = [x ; x] ;;
let f2 x y = (x @ (y x)) ;;
let f3 x = List.map x ;;
Array.create 4 3;;
if false then true else false ;;
let f4 x = Array.create (List.length x) ;;
Hashtbl.create 12 ;;
[| 'a' ; 'b' ; 'c' |].(2) ;;
List.sort;;
let rec fact n = if n<=1 then 1 else n*(fact (n-1));;
Array.map fact [|0;2;3;4|];;
```

```

let greater (a,b) = a > b;;

List.filter greater
  (List.map (fun (a,b) -> (b,a)) [(1,2);(3,3);(9,4);(4,0)]);;

let coupler x y = (x, y);;

let f (x,y) = x=y ;;

let square x = x * x;;

let h f1 f2 x = (f1 x) > (f2 x);;

List.map (h square (fun x -> 10*x)) [2;4;-4;4;32;6;-4];;

let f1 a b = a b;;

let f2 a = a+3;;

f1 f2 1;;

```

Esercizio A2

Scrivere delle funzioni con i seguenti tipi:

```

int * 'a -> ('a * 'a) list
int -> 'a -> (int * bool * 'a) list
('a -> 'a) -> 'a list -> 'a list
('a, 'b) Hashtbl.t -> 'a -> 'b
bool -> 'a -> 'a -> bool
'a -> 'a -> int -> bool
'a list -> 'a -> int -> bool
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
'a -> ('a -> 'a) -> bool
'a list -> 'a array -> bool
'a list -> 'a -> int -> bool
('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
('a * 'b) list -> ('b * 'c) list -> ('a * 'b * 'c) list
'a -> int -> (int * (bool * 'a)) list
('a -> 'a) -> 'a list -> ('b -> 'b)
int -> char -> (int list -> char list)
'a list -> 'b list -> ('a * 'b) list
(int * bool * 'a) -> ('a -> int -> bool)

```

```

('a -> 'a list) -> 'a list -> 'a list list
'a -> int -> 'b list
'a * 'a -> 'a -> ('a * 'a) list -> ('a * 'a) list
'a array -> 'a list -> (bool * 'a) list
bool list -> 'a list -> ('a * 'a) list
('a -> 'b) list -> ('a -> 'c) list -> ('b * 'c) list
int * 'a -> ('a * 'a) list
int -> 'a -> (int * bool * 'a) list
('a -> 'a) -> 'a list -> 'a list
('a, 'b) Hashtbl.t -> 'a -> 'b
'a -> ('a -> 'b) -> 'b -> 'a * 'b
'a list -> ('a -> 'b) -> ('b -> 'c) -> 'c list
'a -> 'b -> ('a list -> 'c list) -> ('a * 'b) array
'a * 'a -> float -> bool

```

Esercizio B1

Scrivere una funzione tail-recursive `occurstwice : 'a -> 'a list -> bool` che ritorna `true` se e solo se il primo argomento occorre due volte nel secondo.

Scrivere una funzione tail-recursive `occursntimes : 'a -> 'a list -> int -> bool` che ritorna `true` se e solo se il primo argomento occorre nel secondo un numero di volte esattamente uguale al terzo argomento.

Scrivere la funzione `occursntimes : 'a -> 'a list -> int -> bool` come descritta sopra in modo piu' compatto possibile, usando le funzioni del package `List`.

Scrivere una funzione `interleave : 'a list -> 'a list -> 'a list`, definita come segue. Siano $[P_1; \dots; P_m]$ e $[Q_1; \dots; Q_n]$ gli argomenti. Se $m = n$, il risultato deve essere la lista $[P_1; Q_1; \dots; P_m; Q_m]$; se $n = m + k$, il risultato deve essere la lista $[P_1; Q_1; \dots; P_m; Q_m; Q_{m+1}; \dots; Q_n]$ (e dualmente se $m = n + k$).

Scrivere una funzione `minmax : 'a list -> ('a * 'a)` che ritorni una coppia il cui primo elemento e' il minimo elemento della lista (secondo l'operatore `<`), e il cui secondo elemento e' il massimo elemento della lista (secondo l'operatore `>`). La funzione deve essere tail-recursive, attraversare la lista una volta sola, e sollevare un'eccezione nel caso in cui la lista sia vuota.

Scrivere una funzione `applfuns` che, data come primo argomento una lista di n funzioni $[f_1; \dots; f_n]$ e come secondo argomento una lista di n elementi $[e_1; \dots; e_n]$, restituisce la lista dei risultati ottenuti dall'applicazione della i -esima funzione f_i all' i -esimo elemento e_i , e che genera un'eccezione quando le liste hanno lunghezza diversa. Definire l'eccezione, definire la funzione, e scrivere il suo tipo piu' generale.

Data una lista di lunghezza n , $[e_1; \dots; e_n]$, definire una funzione che costruisce la lista delle coppie (e_i, e_j) tale che $1 \leq i, j \leq n$, dove $j = i + 2$.

Scrivere una funzione `mkmatrix : int -> int -> (int * int) list list`. Dato `m` come primo argomento e `n` come secondo argomento, la funzione deve restituire `[[(1,1); (1,2); ...; (1,m)]; ...; [(n,1); (n,2); ...; (n,m)]]`.

Scrivere una funzione `transpose : 'a list list -> 'a list list` che, data in input `[[e11; e21; ...; em1]; ...; [e1n; e2n; ...; emn]]`, restituisce `[[e11; e12; ...; e1n]; ...; [em1; em2; ...; emn]]`.

Scrivere una funzione `condapply` a tre argomenti che, dati in input `p : ('a -> bool)`, `f e g` di tipo `'a -> 'b`, e `l : 'a list` della forma `[e1; e2; ...; en]`, restituisce `[r1; r2; ...; rn]`, dove `ri` è il risultato dell'applicazione di `f` a `ei` se `(p ei)` è `true`, e il risultato dell'applicazione di `g` a `ei` se `(p ei)` è `false`.

Scrivere una funzione `hashinverse : ('a -> 'b) -> 'a list -> ('b, 'a list) Hashtbl.t` che restituisce una hash table `tbl` che rappresenta l'inversa della funzione `f` passata come primo argomento, limitatamente alla lista `l` passata come secondo argomento. Formalmente, per ogni `a ∈ l`, `tbl` associa a `(f a)` la lista di tutti gli `ai ∈ l` tali che `(f ai) = (f a)`.

Scrivere una funzione tail-recursive `countoccurrences : 'a -> 'a list list -> int` che conta il numero complessivo di volte in cui il primo argomento occorre nelle liste del secondo argomento.

Scrivere una funzione ad un argomento `countallocalcurrences : 'a list list -> ('a -> int)`. La funzione in output, se applicata ad un elemento, restituisce il numero di volte in cui esso occorre nell'argomento.

Scrivere una funzione tail-recursive `occurstwice : 'a -> 'a list -> bool` che ritorna `true` se e solo se il primo argomento occorre due volte nel secondo.

Scrivere una funzione tail-recursive `occursntimes : 'a -> 'a list -> int -> bool` che ritorna `true` se e solo se il primo argomento occorre nel secondo un numero di volte esattamente uguale al terzo argomento.

Scrivere la funzione `occursntimes : 'a -> 'a list -> int -> bool` come descritta sopra in modo piu' compatto possibile, usando le funzioni del package `List`.

Date due liste ordinate, restituire la lista che contiene solo gli elementi comuni ad entrambe.

Definire una funzione tail-recursive che, date due liste ordinate di tipo `'a`, restituisce una tripla di liste di elementi di tipo `'a`. Il primo elemento e' la lista degli elementi che occorrono solo nella prima lista; il secondo elemento della tripla e' la lista degli elementi che occorrono in entrambe le liste; il terzo elemento della tripla e' la lista degli elementi che occorrono solo nella seconda lista. Le liste in ingresso devono essere attraversate una volta sola.

Definire una funzione tail-recursive che, date due liste non necessariamente ordinate, restituisce una tripla degli elementi tali che occorrono solo nella prima, in entrambe, o solo nella seconda. Non ordinare le liste.

Definire una funzione `countoccs : 'a list -> ('a, int) Hashtbl.t` che, data una lista, costruisce una hash table che associa ad ogni elemento il numero di volte che l'elemento compare nella lista. Per esempio, data `['a'; 'b'; 'c'; 'a'; 'a'; 'b'; 'd'; 'd'; 'd'; 'c']` in input, la hash table deve contenere le coppie `{ ('a', 3); ('b', 2); ('c', 2); ('d', 3) }`.

Definire una funzione `sameoccs : ('a * int) list -> 'a list`, che data una lista di coppie (elemento, frequenza) costruisce una lista di liste, in cui ogni lista contiene gli elementi che ricorrono con la stessa frequenza. Per esempio, dato in input `[('a', 4); ('b', 2); ('c', 3); ('d', 2)]`, l'output deve essere `[['a']; ['b'; 'd']; ['c']]`.

Esercizio B2

Scrivere una funzione tail recursive `findoccs : 'a -> 'a list -> (int * int list)` che restituisce la coppia il cui primo elemento è il numero di occorrenze del primo argomento nel secondo argomento, ed il cui secondo elemento è la lista degli indici delle occorrenze.

Esempio: se `x` è la lista `('a'; 'b'; 'c'; 'a')`, `findoccs 'a' x` restituisce `(2, [1; 4])`, mentre `findoccs 'c' x` restituisce `(1, [3])`.

Scrivere una funzione tail-recursive `crunchlist : 'a list -> ('a, int * int list) Hashtbl.t` che costruisce una hash table in cui, ad ogni elemento che occorre nella lista, è associata una coppia il cui primo elemento è il numero di occorrenze nella lista, ed il secondo elemento è la lista degli indici delle occorrenze.

Scrivere una funzione `member : 'a -> 'a list -> bool` che ritorna `true` se e solo se il primo argomento occorre nel secondo argomento.

Scrivere una funzione tail-recursive `occurs : 'a -> 'a list -> int` che ritorna il numero di occorrenze del primo argomento nel secondo argomento.

Scrivere una funzione `remove : 'a -> 'a list -> 'a list` che ritorna la lista ottenuta rimuovendo tutte le occorrenze del primo argomento dal secondo argomento.

Scrivere una funzione tail-recursive `equalocc : 'a list -> 'a list -> bool` che ritorna `true` se e solo se le liste contengono gli stessi elementi, ed ognuno di essi occorre con lo stesso numero di occorrenze.

Scrivere una funzione `reverse : 'a list -> 'a list` che ritorna la lista roversciata.

Scrivere una funzione tail-recursive `member : 'a -> 'a list -> bool` che ritorna `true` se e solo se il primo argomento occorre nel secondo argomento.

Scrivere una funzione tail-recursive `nomultiples : 'a list -> 'a list` che attraversa una sola volta la lista, e per ogni elemento contenuto nella lista, elimina le occorrenze successive alla prima, preservando l'ordine.

Dato il tipo `type 'a tree = Empty | Tr of 'a * 'a tree * 'a tree` scrivere una funzione `addheight : 'a tree -> ('a * int) tree` che restituisce un albero avente stessa struttura dell'albero in input, e tale che ogni nodo ha al posto della label la coppia (label, altezza del sottoalbero). (Si definisce ricorda altezza di un albero la lunghezza massima tra i path dalla root ad una foglia.)

Scrivere una versione ottimizzata della stessa funzione che implementi memoizing per evitare di ripetere la computazione su alberi uguali; per il memoizing utilizzare una hash table.

Dato il tipo `type 'a ott tree = Empty | Th1 of 'a * 'a ott tree | Th2 of 'a * 'a ott tree * 'a ott tree | Th3 of 'a * 'a ott tree * 'a ott tree * 'a ott tree` scrivere una funzione `count : 'a ott tree -> (int * int * int * int)` che restituisce una quadrupla di interi, i cui elementi sono rispettivamente il numero di occorrenze di nodi di tipo `Empty`, `Th1`, `Th2`, and `Th3`.

Scrivere una funzione di tipo `'a ott tree -> bool`, che restituisce `true` se e solo se ognuno dei nodi dell'albero in input ha un numero di figli inferiore o uguale al proprio padre, altrimenti `false`. Usare il meccanismo delle eccezioni per implementare early termination.

Scrivere una funzione `filter : ('a -> 'b) -> 'a ott tree -> 'b ott tree` che

restituisce un albero con la stessa struttura (ovvero, uguale a meno delle labels) dell'albero in input. Ogni nodo dell'albero risultante ha come etichetta il risultato dell'applicazione della funzione passata in input come primo argomento al corrispondente nodo dell'albero in input.

Scrivere una funzione `filter : 'a ott tree -> 'b ott tree -> bool` che restituisce `true` se e solo se i due alberi hanno la stessa struttura (ovvero, sono uguali a meno delle labels).

Scrivere una funzione tail-recursive `swapfronteven : ('a * 'a) list -> ('a * 'a) list` che restituisce la lista in cui la coppia in posizione pari (secondo, quarto, etc) ha gli elementi scambiati.

Scrivere una funzione `swaptailodd : ('a * 'a) list -> ('a * 'a) list` che restituisce la lista in cui la coppia in posizione dispari contando dal fondo (ultimo, terz'ultimo, etc) ha gli elementi scambiati. Non utilizzare `reverse`.

Scrivere una versione tail-recursive di `position : 'a -> 'a list -> int option` che identifica la prima occorrenza del primo argomento nel secondo argomento. La prima posizione corrisponde a 0. Usare il tipo `option` nel modo ovvio.

Scrivere una funzione tail-recursive `applyfollowing : 'a list -> ('a -> 'a -> 'b) -> 'b list` che restituisce la lista dei valori ottenuti applicando il secondo argomento ad ogni elemento della lista (eccettuato l'ultimo) ed a quello che lo segue.

Scrivere una funzione tail recursive `findoccs : 'a -> 'a list -> (int * int list)` che restituisce la coppia il cui primo elemento e' il numero di occorrenze del primo argomento nel secondo argomento, ed il cui secondo elemento e' la lista degli indici delle occorrenze. Esempio: se `x` e' la lista `('a'; 'b'; 'c'; 'a')`, `findoccs 'a' x` restituisce `(2, [1; 4])`, mentre `findoccs 'c' x` restituisce `(1, [3])`.

Scrivere una funzione tail-recursive `crunchlist : 'a list -> ('a, int * int list)` `Hashtbl.t` che costruisce una hash table in cui, ad ogni elemento che occorre nella lista, e' associata una coppia il cui primo elemento e' il numero di occorrenze nella lista, ed il secondo elemento e' la lista degli indici delle occorrenze.

Data la definizione:

```
type operator = Sum | Mul | Pow
type expression_tree = Value of int | Tr of operator * expression_tree *
expression_tree;;
```

che rappresenta espressioni aritmetiche su numeri interi,

Definire la funzione `string_tree : expression_tree -> string` che, dato in input l'albero di un'espressione, restituisce una stringa contenente una stampa dell'espressione.

Ad esempio se l'albero definito come:

```
Tr(Mul, Tr(Sum, Value 1, Value 3), Tr(Pow, Value 2, Value 2)),
```

L'espressione che deve essere stampata : `"((1 + 3) * (2 ^ 2))"`.

Definire la funzione `expand_power : expression_tree -> expression_tree` che, dato in input un albero espressione, restituisce un albero espressione in cui ogni nodo `Pow` (che rappresenta l'elevamento a potenza) viene sostituito dal corrispondente albero che lo rappresenta tramite moltiplicazioni.

Ad esempio: `2^5` deve venire convertito in `2*2*2*2*2`.

Dato in input l'albero `Tr(Mul, Tr(Sum, Value 3, Value 5), Tr(Pow, Value 2, Value 5))`, che rappresenta l'espressione `(3+5)*2^5`, la funzione deve restituire l'albero corrispondente all'espressione `(3+5)*(2*2*2*2*2)`, ovvero

```
Tr(Mul,
  Tr (Sum, Value 3, Value 5),
  Tr (Mul, Value 2,
    Tr (Mul, Value 2,
      Tr (Mul, Value 2,
        Tr (Mul, Value 2, Value 2))))))
```

Il figlio destro di un nodo Pow deve essere di tipo Value e non un'espressione; generare un'eccezione nel caso in cui questa condizione non sia soddisfatta.

Definire la funzione `eval_tree : expression_tree -> int` che, dato in input l'albero di un'espressione, restituisce l'intero corrispondente alla valutazione dell'espressione. Ad esempio se l'albero definito come: `Tr (Mul, Tr (Sum, Value 1, Value 3), Tr (Pow, Value 2, Value 2))`, il risultato deve essere 16.

Esercizio B3

Data la dichiarazione `type 'a tree = Empty | Tr of 'a * 'a tree * 'a tree ;;` definire la funzione `previsit : 'a tree -> 'a list`, che visita l'albero binario in preordine.

Definire la funzione `notsoclose : int tree -> int -> bool` che, dato un albero `t` con etichette intere come primo argomento ed un intero positivo `lim` come secondo argomento, ritorna `true` sse per ogni nodo `N` di `t` vale che le distanze tra l'etichetta di `N` e l'etichetta dei nodi destro e sinistro di `N` sono entrambi maggiore di `lim`. (Per distanza si intende il valore assoluto della differenza.)

Data la dichiarazione `type 'a tree = Empty | Tr of 'a * 'a tree * 'a tree ;;` definire la funzione `postvisit : 'a tree -> 'a list`, che visita l'albero binario in postordine.

Definire la funzione `rightnotfar : int tree -> int -> bool` che, dato un albero `t` con etichette intere come primo argomento ed un intero positivo `lim` come secondo argomento, ritorna `true` sse per ogni nodo `N` di `t` vale che la distanza (ovvero il valore assoluto della differenza) tra l'etichetta di `N` e l'etichetta del nodo destro di `N` e' minore di `lim`.

Data la dichiarazione `type 'a tree = Empty | Tr of 'a * 'a tree * 'a tree ;;` definire una funzione `sort_branches : 'a tree -> 'a tree`, che ritorna un albero con gli stessi path dell'albero argomento, e tale che per ogni nodo l'etichetta del ramo destro sia minore o uguale (secondo l'operatore `<=`) dell'etichetta del ramo sinistro; per convenzione, i nodi `Empty` devono essere trattati come aventi etichetta di peso maggiore di qualsiasi nodo `Tr`.

Definire la funzione `deepestpath : 'a tree -> 'a list` che, dato un albero `t` come argomento, ritorna il path (ovvero la lista delle etichette dei nodi) dalla radice alla foglia di profondità massima.

Data la dichiarazione `type 'a ttree = Empty | TTr of 'a * 'a ttree * 'a ttree * 'a ttree` scrivere una funzione `tail recursive` che conta i nodi di un albero ternario visitando i nodi una sola volta.

Scrivere una funzione `tail recursive` che ritorna `true` se e solo se ogni nodo ha label uguale al massimo tra le labels dei figli. Ignorare nel test eventuali figli `Empty`.

Scrivere una versione della stessa funzione che usa eccezioni per implementare `early termination`.

Scrivere una funzione `tfilter : 'a ttree -> ('a -> bool) -> 'a list` che dato un albero ternario come primo argomento e una funzione di filtro come secondo argomento e restituisce la lista dei sottoalberi la cui label soddisfa il filtro.

Data la dichiarazione `type 'a ntree = Empty | Tr of 'a * 'a ntree list`, scrivere una funzione di tipo `'a ntree -> 'a list` che lista le etichette dei nodi dell'albero.

Scrivere una funzione di tipo `(a -> bool) -> 'a ntree -> int` che conta quanti nodi

dell'albero (passato come secondo argomento) soddisfano il predicato (passato come primo argomento).

Data la dichiarazione `type 'a tree = Empty | Tr of 'a * 'a tree * 'a tree ;;` definire una funzione `minmax : 'a tree -> ('a * 'a) tree`, che restituisce un albero con la stessa struttura dell'albero in input. Ogni nodo dell'albero risultato e' etichettato dalla coppia costituita dalle etichette minima e massima (secondo l'operatore `<=`) del sottoalbero corrispondente nell'albero in input.

Definire la funzione `storepath : 'a tree -> ('a * 'a list) tree` che, dato un albero `t` come argomento, restituisce un albero con la stessa struttura. Ogni nodo dell'albero risultato ha come etichetta la coppia costituita dall'etichetta del nodo originario corrispondente, e dal path dalla radice al nodo.

Data la dichiarazione `type 'a tree = Empty | Tr of 'a * 'a tree * 'a tree ;;` definire la funzione `previsit : 'a tree -> 'a list`, che visita l'albero binario in preordine.

Definire la funzione `notsoclose : int tree -> int -> bool` che, dato un albero `t` con etichette intere come primo argomento ed un intero positivo `lim` come secondo argomento, ritorna `true` sse per ogni nodo `N` di `t` vale che le distanze tra l'etichetta di `N` e l'etichetta dei nodi destro e sinistro di `N` sono entrambi maggiore di `lim`. (Per distanza si intende il valore assoluto della differenza.)