2021-09-08

Sia find : ’a -> ’a list -> int la funzione tale che find e l restituisce il numero di volte con cui si ripete l’elemento e nella lista l.

Esempi:

find "a" ["a";"c";"c"] = 1
find "c" ["a";"c";"c"] = 2
find "c" ["a";"b";"c"] = 1
find "d" ["a";"b";"c"] = 0

(a) Implementare find senza uso di parametri di accumulazione.
(b) Implementare find usando un param. di acc. affinchè la ricorsione sia di coda.

2021-07-12

Sia swap : (’a ∗ ’b) list -> (’b ∗ ’a) list la funzione cos`ı specificata:

swap $[(x_1,y_1); ... ;(x_k,y_k)] = [(y_1,x_1); ... ;(y_k,x_k)]$, con $k \geq 0$.

Esempi:

swap [(1,"one");(2,"two");(3,"three")] = [("one", 1); ("two", 2); ("three", 3)]
swap []=[]

(a) Definire swap senza uso di parametri di accumulazione.
(b) Definire swap usando List.map:(’a -> ’b) -> ’a list -> ’b list.

2018-02-12

Sia insert_after : (’a -> bool) -> ’a -> ’a list -> ’a list la funzione cos`ı specificata: insert_after p e laggiungel’elementoenellalistalimmediatamentedopoognisuoelementochesoddisfa il predicato p. Esempio:

insert_after (**fun** x->x>3) 0 [] - : int list = []

insert_after (**fun** x->x>3) 0 [1;4;2;5] - : int list = [1;4;0;2;5;0]

- (a) Definire la funzione insert_after senza uso di parametri di accumulazione.
- (b) Definire la funzione insert_after usando un parametro di accumulazione affinchè la ricorsione sia di coda.
- (c) Definire la funzione insert_after come specializzazione della funzione it_list così definita:

let rec it_list f a = **function** x::l -> it_list f (f a x) l | _ -> a;; val it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = **<fun>**

2018-06-04

Sia first : ('a ∗ 'b) list -> 'a list la funzione cos`ı specificata: first $[(a_1, b_1); \ldots; (a_n, b_n)]$ restituisce la lista $[a_1; \ldots; a_n]$.

Esempio: # first [(1,"one");(2,"two");(3,"three")];;

- : int list = [1; 2; 3]

(a) Definire first senza uso di parametri di accumulazione.
(b) Definire first usando un parametro di accumulazione affinche´ la ricorsione sia di coda.
(c) Definire first come specializzazione della funzione map:('a -> 'b) -> 'a list -> 'b list.

2022-01-20

Sia sum_wise : int list -> int list -> int list la funzione tale che

sum_wise $[a_0;...;a_n]$ $[b_0;...;b_n]$ = $[a_0+b_0;\ldots;a_n+b_n]$ e sum_wise l_1 l_2 solleva un'eccezione (usare raise (Invalid_argument "sum_wise")) se l_1 e l_2 non hanno la stessa lunghezza.

Esempi:

sum_wise [0;2;4] [1;3;5]=[1;5;9]
sum_wise [] []=[]
sum_wise [0] [1;3;5] *solleva un'eccezione* sum_wise [0;2;4] [1] *solleva un'eccezione*

(a) Implementare sum_wise senza uso di parametri di accumulazione.
(b) Implementare sum_wise usando un parametro di accumulazione affinche´ la ricorsione sia di coda.

2021-06-03

Sia fuse : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list la funzione cos`ı specifica- ta:
fuse f $[x_1; \ldots; x_k]$ $[y_1; \ldots; y_n]$ = $[f\ x_1\ y_1; \ldots; f\ x_m\ y_m]$,conk,n ≥ 0em = min(n,k).
Esempi:
fuse max [1;2;3] [3;1;4] = [3;2;4]

fuse max [1;2;3] [3] = [3]
fuse max [4] [3;5;7] = [4]
fuse (+) [1;2;3] [3;1;4] = [4;3;7]
fuse (+) [1;2;3] [3] = [4]

fuse (+) [4] [3;5;7] = [7]
(a) Definire fuse senza uso di parametri di accumulazione.
(b) Definire fuse usando un parametro di acc. affinche´ la ricorsione sia di coda.

2021-06-21

Sia cat : (string ∗ string) list -> string list la funzione cos`ı specificata:

cat $[(x_1,y_1); ... ;(x_k,y_k)] = [x_1 \hat{} y_1; ... ;x_k \hat{} y_k]$, dove $k \geq 0$ e ^ rappresenta l'operatore di concatenazione tra stringhe in OCaml.

Esempi:

cat [("hello"," world");("ciao ","mondo")]=["hello world"; "ciao mondo"]
cat []=[]
(a) Definire cat senza uso di parametri di accumulazione.
(b) Definire cat usando List.map:('a -> 'b) -> 'a list -> 'b list.

2020-01-23

Sia zip : 'a list -> 'b list -> ('a ∗ 'b) list la funzione cos`ı specificata (con $n, k \geq 0$): • zip $[x_1;...;x_{n+k}]$ $[y_1;...;y_n]$ restituisce la lista di coppie $[(x_1,y_1);...;(x_n,y_n)]$;

zip $[x_1;...;x_n]$ $[y_1;...;y_{n+k}]$ restituisce la lista di coppie $[(x_1,y_1);...;(x_n,y_n)]$.

Esempi:

zip [1;2;3] ["one";"two";"three"]=[(1, "one"); (2, "two"); (3, "three")];;
zip [1;2] ["one";"two";"three"]=[(1, "one"); (2, "two")];;
zip [1;2;3] ["one";"two"]=[(1, "one"); (2, "two")];;
(a) Completare la seguente definizione di zip senza uso di parametri di accumulazione.
let rec zip l1 l2 = **match** l1,l2 **with** ***(∗ completare ∗)***

(b) Definire zip usando un parametro di accumulazione affinche´ la ricorsione sia di coda.

2019-09-09

Sia count_zeros : ('a -> int) -> 'a list -> int la funzione cos`ı specificata: count_zeros f l restituisce il numero di elementi e della lista l per i quali vale l'uguaglianza f (e) = 0.
Esempi:

count_zeros (**fun** x->(x-1)∗(x-2)∗(x+3)) [-3;1;2;0;4] - : int = 3
count_zeros (**fun** x->(x-1)∗(x-2)∗(x+3)) [-1;0;4]
- : int = 0

(a) Definire count_zeros senza uso di parametri di accumulazione.

(b) Definire count_zeros usando un parametro di accumulazione affinche´ la ricorsione sia di coda.

(c) Definire count_zeros come specializzazione della funzione List.fold_left : (’a -> ’b -> ’a) -> ’a -> ’b list -> ’a.

2019-07-10

Sia gen_cat : (’a -> string) -> ’a list -> string la funzione cos`ı specificata, dove ^ e` l’operatore di concatenazione di stringhe:

gen_cat f [s_1; s_2; . . . ; s_n] = f(s_1)^f(s_2)^. . .^f(s_n), con n ≥ 0. Esempio:

gen_cat (**fun** s -> s^"__") ["one";"two";"three"];; - : string = "one__two__three__"

(a) Definire gen_cat senza uso di parametri di accumulazione.

(b) Definire gen_cat usando un parametro di accumulazione affinche´ la ricorsione sia di coda.

(c) Definire gen_cat come specializzazione della funzione
List.fold_left : (’a -> ’b -> ’a) -> ’a -> ’b list -> ’a.

2019-06-19

Sia gen_prod : (’a -> int) -> ’a list -> int la funzione cos`ı specificata: gen_prod f [x_1; x_2; ... ; x_n] = f(x_1)·f(x_2)·...·f(x_n),conn≥0. Esempi:

gen_prod (**fun** x->x+2) [] - : int = 1
gen_prod (**fun** x->x+2) [1] - : int = 3

gen_prod (**fun** x->x+2) [1;2]
- : int = 12
gen_prod (**fun** x->x+2) [1;2;3] - : int = 60

(a) Definire gen_prod senza uso di parametri di accumulazione.

(b) Definire gen_prod usando un parametro di accumulazione affinche´ la ricorsione sia di coda.

(c) Definire gen_prod come specializzazione della funzione
List.fold_left : (’a -> ’b -> ’a) -> ’a -> ’b list -> ’a.

2019-06-05

Sia gen_sum : (’a -> int) -> ’a list -> int la funzione cos`ı specificata: gen_sum f [x_1; x_2; ... ; x_n] = f(x_1)+f(x_2)+...+f(x_n),conn≥0. Esempi:

gen_sum (**fun** x->x∗x) [] - : int = 0
gen_sum (**fun** x->x∗x) [1] - : int = 1

gen_sum (**fun** x->x∗x) [1;2]
- : int = 5
gen_sum (**fun** x->x∗x) [1;2;3] - : int = 14

(a) Definire gen_sum senza uso di parametri di accumulazione.

(b) Definire gen_sum usando un parametro di accumulazione affinche´ la ricorsione sia di coda.

(c) Definire gen_sum come specializzazione della funzione
List.fold_left : (’a -> ’b -> ’a) -> ’a -> ’b list -> ’a.

2019-02-11

Siacond_map : (’a -> ’b) -> (’a -> ’b) -> (’a -> bool) -> ’a list -> ’b listla funzione cos`ı specificata:

cond_map f g p l restituisce la lista ottenuta da l applicando, nell’ordine, la funzione f agli elementi di l che soddisfano il predicato p e la funzione g a quelli che non lo soddisfano.

Esempio:

cond_map sqrt (**fun** x -> 0.) (**fun** x -> x>=0.0) [-1.0;9.0;-4.0;4.0] - : float list = [0.; 3.; 0.; 2.]

(a) Definire cond_map senza uso di parametri di accumulazione.

(b) Definire cond_map usando un parametro di accumulazione affinche´ la ricorsione sia di coda.

(c) Definire cond_map come specializzazione della funzione List.map : (’a -> ’b) -> ’a list -> ’b list .

2019-01-24

Sia cond_map : (’a -> ’a) -> (’a -> bool) -> ’a list -> ’a list la funzione cos`ı spe- cificata:

cond_map f p l restituisce la lista ottenuta da l applicando, nell’ordine, la funzione f agli elementi di l che soddisfano il predicato p e lasciando invariati i restanti.

Esempio:

cond_map sqrt (**fun** x->x>=0.0) [-1.0;9.0;-4.0;4.0] - : float list = [-1.0; 3.0; -4.0; 2.0]

(a) Definire cond_map senza uso di parametri di accumulazione.

(b) Definire cond_map usando un parametro di accumulazione affinche´ la ricorsione sia di coda.

(c) Definire cond_map come specializzazione della funzione it_list o List.fold_left: it_list: (’a -> ’b -> ’a) -> ’a -> ’b list -> ’a

2018-09-10

Sia filter_map : (’a -> bool) -> (’a -> ’b) -> ’a list -> ’b listlafunzionecos`ıspe- cificata:

filter_map p f l restituisce la lista ottenuta da l eliminando gli elementi che non soddisfano il predicato p e applicando, nell’ordine, la funzione f ai restanti.

Esempio:

filter_map (**fun** x->x>=0.0) sqrt [-1.0;0.0;-4.0;4.0];; - : float list = [0.0; 2.0]

(a) Definire filter_map senza uso di parametri di accumulazione.

(b) Definire filter_map usando un parametro di accumulazione affinche´ la ricorsione sia di coda.

(c) Definire filter_map come specializzazione della funzione it_list o List.fold_left: it_list: (’a -> ’b -> ’a) -> ’a -> ’b list -> ’a

2018-07-11

Sia merge : (’a ∗ ’b -> ’c) -> (’a ∗ ’b) list -> ’c list la funzione cos`ı specificata: merge f $[(a_1, b_1); \ldots; (a_n, b_n)]$ restituisce la lista $[f(a_1, b_1); \ldots; f(a_n, b_n)]$.
Esempio:

merge (**fun** (x,y) -> x+y) [(1,2);(3,4);(5,6)];;
- : int list = [3; 7; 11]

merge (**fun** (x,y) -> x+String.length y) [(1,"one");(2,"two");(3,"three")];; - : int list = [4; 5; 8]

(a) Definire merge senza uso di parametri di accumulazione.
(b) Definire merge usando un parametro di accumulazione affinche´ la ricorsione sia di coda.
(c) Definire merge come specializzazione della funzione map:(’a -> ’b) -> ’a list -> ’b list.

2018-06-20

Sia swap : (’a ∗ ’b) list -> (’b ∗ ’a) list la funzione cos`ı specificata: swap $[(a_1,b_1);...;(a_n,b_n)]$restituiscelalista$[(b_1,a_1);...;(b_n,a_n)]$.
Esempio:
swap [(1,"one");(2,"two");(3,"three")];;

- : (string ∗ int) list = [("one", 1); ("two", 2); ("three", 3)]

(a) Definire swap senza uso di parametri di accumulazione.
(b) Definire swap usando un parametro di accumulazione affinche´ la ricorsione sia di coda.
(c) Definire swap come specializzazione della funzione map:(’a -> ’b) -> ’a list -> ’b list.