

Programmazione Funzionale

Esercizi Svolti di Programmazione Funzionale

Anno Accademico: 2024/25

Giacomo Sturm

*Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche
Università degli Studi “Roma Tre”*

Sorgente del file L^AT_EX disponibile al seguente link:

<https://github.com/00Darxk/Programmazione-Funzionale/>

Indice

1	Esercitazione 1	1
2	Esercitazione 2	2
3	Esercitazione 3	3
4	Esercitazione 4	5
5	Esercitazione 5	6
6	Esercitazione 6	7

1 Esercitazione 1

2 Esercitazione 2

Definire una funzione `ultime_cifre: -> int * int` che riporti il valore intero delle due ultime cifre di un `int`. Dato un numero, il suo modulo base 10 restituisce l'ultima cifra. Mentre per ottenere la penultima si effettua una divisione intera per 10 per eliminare l'ultima cifra e si riapplica il modulo base 10 per ottenere questa cifra:

```
let ultime_cifre x = (abs(x) mod 10, abs(x/10) mod 10);;
```

Una cifra è bella se è 0, 3 o 7; un numero è bello se la sua ultima cifra è bella e la penultima (se esiste) non lo è. Definire un predicato `bello: int -> bool`, che determini se un numero è bello, la funzione non deve mai sollevare eccezioni, ma riportare sempre un `bool`.

```
let bello x =  
  match abs(x mod 10) with  
  0 | 3 | 7 -> (match (abs(x mod 10))/10 with  
    0 | 3 | 7 -> x < 10  
    3 | 7 -> false  
    | 0 -> x < 100  
    | _ -> true)  
  | _ -> false;;
```

Scrivere una funzione `data: int * string -> bool`, che applicata ad una coppia `(d, m)` di un intero `d` e una stringa `m`, determini se la coppia rappresenta una data corretta, assumendo che l'anno non sia bisestile, si assuma che i mesi siano rappresentati da stringhe con caratteri minuscoli. La funzione non deve sollevare eccezioni, ma riportare sempre un `bool`. È utile scrivere una funzione `gdm` per determinare i giorni di un mese:

```
let gdm m =  
  match with  
  "gennaio" | "marzo" | "maggio" | "luglio" | "agosto" | "ottobre" | "dicembre" -> 31  
  | "aprile" | "giugno" | "settembre" | "novembre" -> 30  
  | "febbraio" -> 28  
  | _ -> 0
```

La funzione principale è quindi:

```
let data (d,m) = d <= gdm m && d > 0;;
```

3 Esercitazione 3

```
exception NoElement ;;
let maxlist = function
  [] -> raise NoElement
  | x::xs -> let rec aux m = function
    [] -> m
    | x::xs -> if x > m then aux x xs
                else aux m xs
  in aux x xs
```

Si ipotizza di non avere a disposizione l'operatore @, definire la concatenazione:

```
let rec append l1 l2 = match l1 with
  [] -> l2
  | x::xs -> x::(append xs l2)
```

In modo iterativo diventa:

```
let append l1 l2 =
  let rec aux a = function
    [] -> a
    | x::xs -> aux (x::a) xs
  in aux l2 List.rev(l1)

let rec nth n = function
  [] -> raise NoElement
  | x::xs -> if n < 0 then raise NoElement
             else if n = 0 then x
                   else nth (n-1) xs;;

let rec nondec = function
  [] | [a] -> true
  | x::y::xs -> if x > y then false
                else nondec (y::xs)

let min_dei_max ls =
  let rec listmax acc = function
    [] -> acc
    | x::xs -> listmax ((maxlist x)::acc) xs
  in let rec minlist = function
    [] -> raise NoElement
    | x::xs -> try let y = minlist(xs)
                  in min x y
                with _ -> x
  in minlist(listmax [] ls);;
```

Concatena l'accumulatore con i massimi di ciascuna lista del secondo argomento, l'argomento implicito è una lista di liste, essendo l'argomento della funzione esterna `ls`. Si può inferire il tipo dato che ad un elemento della lista `x` viene passato come argomento a `maxlist`, quindi deve essere una lista.

`split2` agisce in modo simile alla funzione `split` definita precedentemente, divide in due parti ...

4 Esercitazione 4

5 Esercitazione 5

6 Esercitazione 6

Si definisca la funzione `find`, definita nel modulo `List`, tale che `find p lst` restituisca il primo elemento della lista `lst` che soddisfa il predicato `p`, altrimenti solleva un'eccezione.

```
exception NotFound;;
let rec find p = function
| [] -> raise NotFound
| x::xs -> if (p x) then x
           else find p xs
```

Si definisce la funzione `takewhile p lst` che riporti la più lunga parte iniziale di `lst` costituita da tutti gli elementi che soddisfano il predicato `p`.

```
let rec takewhile p = function
| [] -> []
| x::xs -> if (p x) then x::(takewhile p xs)
           else []
```

Definire ora un predicato `p`, tale che la funzione `takewhile` restituisca la parte iniziale contenente solo numeri pari:

```
takewhile (function x -> x mod 2 = 0) 1
```

Si definisca la funzione `partition: ('a → bool) → 'a list → ('a list * 'a list)`, tale che applicata ad un predicato `p` ed una lista `lst` restituisca una coppia di liste, dove la prima contiene tutti gli elementi che soddisfano il predicato, mentre la seconda lista contiene tutti gli elementi che non lo soddisfano:

```
let partition p l =
  let rec aux (yes, no) = function
  | [] -> (yes, no)
  | x::xs -> if (p x) then aux ((yes@[x]), no) xs
              else aux (yes, (no@[x])) xs
  in aux ([], []) l
```

Definire la funzione `pairwith`, che dato un elemento ed una lista restituisce una lista di coppie formate dall'elemento passato e l'elemento corrispondente della lista passata, utilizzando `List.map`

```
let pairwith x l = List.map (function y -> (x, y) ) l
```

Definire la funzione `setdiff` per la differenza insiemistica, utilizzando `List.filter`:

```
let rec isin l x = match l with
| [] -> true
| y::ys -> if (x = y) then false
           else isin ys x;;
let setdiff l1 l2 = List.filter (isin l2) l1
```

In maniera più concisa, invece di utilizzare un'altra funzione `isin` si può utilizzare la funzione `exists` dal modulo `List`, che prende un predicato ad una lista e restituisce un booleano che rappresenta se esiste un elemento nella lista che verifica quel predicato.

```
let mem l x = List.exists ((=) x) l;;  
let setdiff l1 l2 = List.filter (not (mem l2)) l1
```

Un'altra soluzione possibile è la seguente:

```
let rec setdiff l1 = function  
| [] -> l1  
| x::xs -> setdiff (List.filter (function y -> y <> x) l1) xs
```