Programmazione Funzionale

Correzione Esonero 1 – Giovedi 30 novembre 2023

NB. Queste sono le possibili soluzioni agli esercizi dell'esonero 1. Oviamente le soluzioni non sono uniche e per una domanda esistono tanti modi diversi di risolverla, pero potete trovare qua dei modi di affrontare le domande.

Esercizio 1 (Liste Palindrome). I-1. Vogliamo definire una funzione `invert : a' list -> a' list` che inverte una lista. Ci sono diverse possibilitá.

Una versione tail recursive:

```
let invert l =
    let rec aux = function
        ([] , acc) -> acc |
        (x::l , acc) -> aux (l,x::acc)
    in aux (l , []);;
```

In questo modo la funzione ausiliaria aux e definita localmente all'interno della definizione di `invert`. Volendo si puo definire aux globalmente e poi definire `invert`.

```
let rec aux = function
    ([] , acc) -> acc |
    (x::l , acc) -> aux (l,x::acc);;
let invert l = aux (l,[]);;
```

Un altra possibilitá era di usare l'operazione di concatenazione `@` e una versione non tail recursive;

```
let rec invert = function
    [] -> [] | x::l -> invert(l) @ [x] ;;
```

I-2. Vogliamo definire `pali` che verifica se una lista e un palindrome. Semplicemente, questo significa che una lista e uguale al suo inverso, allora possiamo semplicemente scrivere;

```
let pali l = l = invert(l);;
```

Se non siamo sicuri che l'uguaglianza sulle liste e definita la possiamo definire:

```
let rec isequal = function
    ([],[]) -> true |
    ([],x) -> false |
    (x,[]) -> false |
    (x1::l1,x2::l2) -> if (x1=x2) then (isequal(l1,l2)) else (false);;
let pali l = isequal(l,invert(l));;
```

Volendo possiamo usare when durante il pattern matching per un lettura piu facile;

```
let rec isequal = function
    ([],[]) -> true |
    ([],x) -> false |
    (x,[]) -> false |
    (x1::l1,x2::l2) when x1=x2 -> isequal(l1,l2) ;;
    (x1::l1,x2::l2) -> false ;;
```

II-1. La funzione length viene definita semplicemente come seguente (volendo lo possiamo fare in modo tail recursive):

```
let rec length = function
    [] -> 0 | (*deve essere 0 e non un eccezione !*)
    x::l -> 1+ length(l) ;;
```

II-2. Non andiamo a complicare il problema... cerchiamo semplicemente di adattare la funzione ausiliaria di `invert` con un parametro (un intero) in piu che conta al contrario;

```
let rec invertWithControl = function
 (l1,l2,0) -> (l1,l2) | (*n il nostro contattore e arrivato a 0 e quindi abbiamo finito*)
    ([],l2,n) -> ([],l2) | (*n e piu grande che la lunghezza di l1*)
    (x1::l1,l2,n) -> invertWithControl(l1,x1::l2,n-1);;
```

II-3 Usiamo `invertWithControl` per creare `divide`;

```
exception Dispari;;
let divide_aux l = if (length(l) mod 2 = 0)
    then (invertWithControl(l,[] , length(l)/2))
    else (raise Dispari);;
```

E praticamente la funzione che vogliamo pero una delle liste e invertita, definiamo `divide` fissando quel problema:

```
let divide l =
    let (l1,l2) = divide_aux(l)
    in (l1,invert(l2));;
```

II-4 Quest'ultima domanda e piu difficile, possiamo ottenere una soluzione parziale usando `divide_aux` (é non `divide` perche vogliamo testare l'uguaglianza sulla la lista invertita dei n primi di l1, cioè `[1;2;2;1]` e un palindrome ma non `[1;2;1;2]`);

```
let palieasy l =
    let (l1,l2) = divide_aux l
    in l1=l2;;
```

La soluzione e solo parziale perche `palieasy` sollevera l'eccezione `Dispari` quando la lista in entrata ha una lunghezza dispari. Dobbiamo quindi definire una funzione che toglie l'elemento in mezzo di una lista.

```
exception OutofBound;;
let rec remove = function
    (x::l,0) -> l |
    ([],n) -> raise OutofBound |
    (x::l , n) -> x:: remove(l,n-1);;
```

Ora, dato una lista possiamo rimuovere il suo elemento in mezzo;

```
let removeMid l = remove (l,length(l)/2);;
```

Quando la lista ha una lunghezza dispari possiamo togliere il suo elemento in mezzo e chiamare palieasy:

```
let pali1 = function
    l when length(l) mod 2 = 0 -> palieasy(l) |
    l -> palieasy(removeMid l);;
```

Esercizio 2 (Clausole). 1. Ci sono diversi modi di fare. (E inutile usare troppi `if-then-else` o dichiarazioni del stile `b=true`).

```
let rec andListHighCost = function
    [] -> true | b::l -> b && andListHighCost(l);;

let rec andList = function
    [] -> true |
    b::l when b = false -> false |
    b::l -> andList(l) ;;
```

2. Simile al punto precedente.

```
let rec orListHighCost = function
    [] -> true | b::l -> b || orListHighCost(l);;

let rec orList = function
    [] -> false |
    b::l when b = true -> true |
    b::l -> orList(l) ;;
```

1. Si poteva usare l'equivalenza $A \Rightarrow B \equiv \neg A \vee B$ cioè;

```
let clausola l1 l2 = not( andList(l1) ) || orList(l2);;
```

Esercizio 3 (Massimo Comune Divisore). 1. Per la divisione e comodo usare il modulo, qua non sono necessari `if-then-else`.

```
let divide n m = (m mod n = 0);;
```

2. Per ottenere i divisori di un intero n possiamo percorere i interi piu piccoli di n e salvare (in un accumulatore) quelli che dividono n

```
let rec divisors n =
    let rec aux acc d n = match d with
        x when (n=x) -> n::acc | (*Fermiamo la ricerca dei divisori a n*)
        x when (n mod x = 0) -> aux (x::acc) (x+1) n |
        x -> aux acc (x+1) n
    in aux [] 1 n;;
```

3. Ci sono diversi modi di risolvere il problema, possiamo usare `divisors` sugli interi n e m per poi fare l'intersezione delle due liste e trovare l'elemento piu grande della lista-intersezione. La difficolta di quel metodo e di definire l'intersezione per le liste ma e fattibile.

   Un altro metodo piu efficace in tempo e di usare `divisors2` che dato due interi trova la lista degli loro divisori comuni, adattendo `divisors` possiamo trovare una tale funzione;

```
let rec divisors2 n m =
    let rec aux acc d n m = match d with
     x when (n=x && (n mod x = 0) && (m mod x = 0)) -> x::acc | (*Fermiamo la ricerca dei divisori a n*)
       x when (n=x) -> acc | (*Fermiamo la ricerca dei divisori a n*)
       x when ((n mod x = 0) && (m mod x = 0)) -> aux (x::acc) (x+1) n m |
       x -> aux acc (x+1) n m
    in aux [] 1 n m;;
```

Per determinare il mcd ci serve una funzione che dato una lista di interi ne trova l'elemento massimale;

```
let maxList l =
    let aux l current = match l with
        [] -> current |
        x::l when (x< current || x=current)-> aux l current|
        x::l -> aux l x
    in aux l 0;;
```

Attenzione nel caso in cui la lista e vuota ritorna 0, volendo possiamo gestire quel eccezione. Notate che nel caso in cui cerchiamo il mcd, dato che 1 divide qualsiasi intero le liste di divisori contengono almeno sempre 1 e quindi questo 'errore' non ha consequenze in questo caso.

```
exception Empty;;
let maxListSafe = function
    [] -> raise Empty | l -> maxList (l);;
```

Ora possiamo concludere componendo le funzioni:

```
let mcd n m = maxList(divisors2(n,m));;
```

Esercizio 4 (Iterazione e Interi di Church)**.**

1.
```
let cur f a b = f(a,b);;
```

La funzione iter puo essere definito in diversi modi equivalenti, qua abbiamo una versione che prende in entrata una tripla e una sua forma 'currificata' se vogliamo.

```
let rec iter = function
    (f,0,k) -> k |
    (f,n,k) -> f(iter(f,n-1,k));;

let rec iter f n k = match n with
    0 -> k | n -> f(iter f (n-1) k);;
```

2. La funzione `church` deve prendere un intero e restituisce una funzione che prende f e k per restituire `iter(f,n,k)`. Piu precisamente, e una funzione in n che restituisce una funzione che prende k e restituisce una funzione che prende f per finalmente dare `iter(f,n,k)`.

```
let church n =
function k -> function f -> iter(f,n,k);;
```

3. Dato un intero n la sua rappresentazione e `function x -> function f ->` $\mathtt{f}^{n}$(x) cioè `function x -> function f -> iter(f,n,k)`.

   Se x e di tipo α dato che f puo essere applicata a x, la funzione f e di tipo $\alpha \to \beta$ in piu f puo essere applicata a $f(x)$ quindi per forza i tipi α e β sono i stessi.

   Di consequenza la nostra funzione e di tipo $\alpha \to (\alpha \to \alpha) \to \gamma$ dove γ e il tipo di `iter(f,n,k)` ma `iter` restituisce dei elementi della forma $f(y)$ e quindi questi elementi hanno il tipo α.

   Quindi la rappresentazione di un intero n ha il tipo $\alpha \to (\alpha \to \alpha) \to \alpha$.

4. `church` prende un intero n e gli restituisce la sua rappresentazione di tipo $\alpha \to (\alpha \to \alpha) \to \alpha$. di fatto la funzione `church` e di tipo `int` $\to (\alpha \to (\alpha \to \alpha) \to \alpha)$.

Esercizio 5 (Espressione ricorsive e stringhe)**.** Consideriamo di aver dichiarato il tipo ricorsivo seguente:

```
type espr = Name of string | Space of espr | Concat of espr*espr
```

1. La sostituzione e un processo ricorsivo sul espressione. Se il construttore e un construttore di base (in questo caso `Name`) viene confrontato la stringa, nei altri casi la funzione passa attraverso il construttore (per `Space` e `Concat`).

```
let rec substitution = function
    (Name s1 , s2 , e) when (s1 = s2) -> e |
    (Name s1 , s2 , e)  -> Name s1 |
    (Space(e1),s,e) -> Space (substitution(e1,s,e)) |
    (Concat(e1,e2),s,e) -> Concat (substitution(e1,s,e),substitution(e2,s,e)) ;;
```

2. La funzione `eval: espr -> string` a un comportamento simile ma restituisce una stringa, ad ogni construttore corrisponde un unico comportamento.

```
let rec eval = function
    Name s -> s |
    Space s  -> eval (s) ^ " "|
    Concat(e1,e2)  -> eval(e1) ^ eval(e2);;
```

Esercizio 6 (Ordinamento di liste).

1. La funzione transition aveva un comportamento esplicitamente definito;

```
let transition = function
    ([],l2) -> l2 |
    (x1::l1,[]) -> [x1] |
    (x1::l1,x2::l2) when (x2<x1) -> x1::x2::l2 |
    (x1::l1,x2::l2) -> x2::l2;;
```

2. Usando la funzione transition possiamo definire `suborder` (non e l'unico modo):

```
let suborder l =
    let rec aux = function
        ([],l2) -> l2 |
        (x1::l1,l2) -> aux(l1 , transition(x1::l1,l2)) |
    in invert(aux (l,[]));;
```

   Se non facciamo l'inversione della lista otteniamo i elementi di l ordinati nel modo decrescente e invertito rispetto alla letture in l.

3. Diverse soluzione esistono. Qua uno puo pensare ad ottenere `suborder(l)` e poi costruire la lista di `l` togliendo tutti i elementi ti `l` che occorono in `suborder(l)`. L'implementazione di questo metodo e possibile ma forse non la cosa piu semplice. Una cosa piu grave e che questo metodo non funziona quando `l` contiene delle ripetizione cioe magari l'intero 2 occore due volte in `l`. Un contro esempio puo essere la lista `[2;3;5;2;7]`, applicata a questa lista `suborder` restituira `[2;3;5;7]` e la differenza della due liste e allora `[]`.

   Una soluzione e di adattare la funzione `suborder` precedente, basta aggiungere un altra lista nel accumulatore a chi vengono aggiunti i elementi che `suborder` eliminava, poi la funzione restituira una coppia di liste alla fine de l'esecuzione.

```
let orderSplit1 l =
    let rec aux = function
        ([],l2,l3) -> (l2,l3) |
        (x1::l1,[],l3) -> aux(l1,[x1],l3)  |
        (x1::l1,x2::l2,l3) when x1>=x2 -> aux(l1,x1::x2::l2,l3)  |
        (x1::l1,x2::l2,l3) -> aux(l1,x2::l2,x1::l3)
    in aux (l,[],[]);;
```

   E quasi la soluzione in realta le liste ottenute sono invertite quindi possiamo concludere con:

```
let orderSplit l =
    let (la,lb) = orderSplit1(l)
    in (invert(la),invert(lb));;
```

4. Per aggiungere un elemento al posto giusto basta percorere la lista e aggiungere l'elemento al momento giusto. In questo caso, n viene aggiunto prima di un elemento $x \leq n$.

```
let rec orderAdd l n = match l with
    [] -> [n] |
    x::l when x<n -> x:: orderAdd l n |
    x::l -> n::x::l ;;
```

5. Dato una lista ordinata possiamo usare `orderAdd` per aggiungersci degli elementi senza rompere la proprieta di ordinamento della lista, cioè se `l` e ordinata allora per qualsiasi intero `n` la lista `orderAdd l n` rimane ordinata.

   Usando `orderSplit` otteniamo una lista ordinata `l1` a partire di `l` in piu abbiamo la lista `l2` degli elementi non ordinati di `l` basta allora aggiungere a `l1` i elementi di `l2` usando `orderAdd`.

```
let rec orderSum l1 l2 = match l2 with
    [] -> l1 | x2::l2 -> orderSum (orderAdd l1 x2) l2;;

let order l = let (la,lb)=orderSplit l in orderSum la lb;;
```

`orderSplit` restituisce una coppia di liste mentre `orderSum` prende due liste una dopo l'altra (essentialmente corrisponde a una curryficazione) quindi usiamo una dichiarazione locale per dare l'output di `orderSplit` a `orderSum`.