F#

Malchiodi Riccardo

Esercizio 1

mulALL : int list -> int

`mulAll ls` restituisce il prodotto di tutti i numeri interi contenuti nella lista `ls`.

Esempio:

```
assert  mulAll (2 :: 4 :: 3 :: []) = 24)
```

Soluzione:

```
let rec mullALL ls =
   match ls with
   | hd :: tl -> hd * mullALL tl   // hd = head, tl = tail
   | [] -> 1
```

Siccome dobbiamo moltiplicare tutti gli elementi della lista ci conviene farlo in modo ricorsivo, per farlo dobbiamo scrivere ***let rec (nome funzione)(nome lista)***
L'istruzione ***match (nome lista) with*** significa "esamina la lista ls e cerca di capire a quale pattern corrisponde tra quelli elencati".

Pattern matching

Pattern matching and function definitions

- pattern matching is useful for defining functions by cases
- example

```
let rec addAll ls = (* adds all elements of ls *)
  match ls with
  | hd::tl -> hd + addAll tl (* inductive step, non-empty list *)
  | [] -> 0                    (* base case, empty list *)

assert (addAll (1::2::3::[]) = 6)
```

Screen delle slide

Tornando nel nostro esercizio :

| [ ] -> 1 : indica il caso base, nel nostro caso è quando la lista è vuota, restituisce 1

| hd :: tl -> hd * mullALL tl: Questo pattern corrisponde a una **lista non vuota**, rappresentata come **hd :: tl.**
hd rappresenta la testa della lista mentre **tl** la coda (il resto della lista).
Infine moltiplichiamo **hd** per il risultato della chiamata ricorsiva **mulAll tl**, che calcola il prodotto degli elementi rimanenti.

Esercizio 2

isIn : 'a -> 'a list -> bool

`isIN el ls` restituisce `true` se e solo se `el` è un elemento della lista `ls`.

Esempio:

```
assert isIn 3(2 :: 4 :: 3 :: [])
assert not (isIn 5(2 :: 4 :: 3 :: []))
```

Soluzione:

```
let isIn el ls =
    let rec isInAux ls =
        match ls with
        | hd :: tl -> el = hd || isInAux tl
        | [] -> false
    isInAux ls

assert isIn 3 (2 :: 4 :: 3 :: [])
assert not (isIn 5 (2 :: 4 :: 3 :: []))
```

Dividiamo per step :

1) Definiamo la funzione esterna(principale), nel nostro caso è la funzione isIn.
   accetta 2 argomenti: el, ls.

2) Definire funzione ausiliare (ricorsiva)
   let rec isInaux ls =

   lista è il parametro della funzione aux, ed è una lista (inizialmente corrisponderà a ls, cioè alla lista passata a isIn).

3) ***match lis with***
   Stessa speigazione di sopra, è semplicemente un pattern matching per analizzare la lista ***lista***

4) Caso base

   Il nostro **caso base** ora corrisponde a ***| [] -> false*** , questa istruzione va a indicare che che se la lista è vuota allora deve ritornare false (non abbiamo trovato el)

5) Caso ricorsivo

   il nostro caso ricorsivo è ***| hd :: lt -> el = hd || isInaux lt***

- Se lista ha almeno un elemento, viene suddivisa in:
  - hd, il primo elemento della lista.
  - lt, la "coda" della lista, ovvero il resto degli elementi.
- La struttura hd :: lt significa "lista con testa hd e coda lt".

Condizione e Ricorsione

- **Condizione**: ***el = hd || isInaux lt***
  - ***el = hd***: Verifica se **hd** (il primo elemento) è uguale a **el.**
    - Se **hd = el** è true, allora ***el = hd || isInaux tl*** sarà true (grazie all'operatore | |), quindi aux restituirà true.
    - Se **hd = el** è false, ***isInaux tl*** viene chiamata per verificare se el è presente in **tl.**
  - **Ricorsione**: ***isInaux*** richiama la funzione aux con tl (la coda della lista) come nuovo argomento, continuando così la ricerca su ogni elemento della lista rimanente.

6) Chiamata Iniziale a aux con ls

   Infine, chiamiamo **aux** passando **ls** come argomento iniziale. Questo avvia la ricorsione e inizia a scorrere la **lista ls** per verificare se contiene l'elemento **el.**

In alternativa il codice poteva essere abbreviato in questo modo:

```
let isIn el ls =
        let rec aux = function
          | hd :: lt -> (hd = el) || aux lt
          | [] -> false
        aux ls
```

Example of abbreviated definition

```
let rec addAll =
    function
    | hd::tl -> hd + addEven tl
    | [] -> 0
```

is an abbreviation of

```
let rec addAll l =
  match l with
   | hd::tl -> hd + addAll tl
   | [] -> 0
```

Screen delle slide

Esercizio 3

insert : 'a -> 'a list -> 'a list

`insert el ls` restituisce la lista ottenuta aggiungendo `el` in fondo alla lista `ls` se `el` non appartiene già a `ls`; restituisce `ls` altrimenti.

Esempio:

```
assert (insert 0(2 :: 4 :: 3 :: []) = 2 :: 4 :: 3 :: 0 :: [])
assert (insert 3(2 :: 4 :: 3 :: []) = 2 :: 4 :: 3 :: [])
```

Soluzione:

```
let rec insert el ls =
   match ls with
   | head :: tail ->
      if head = el then
         ls
      else
         head :: insert el tail
   | [] -> [el]
```

1) **Controllo della Lista Vuota**:

- Se la lista è vuota ([ ]), restituiamo una nuova lista contenente solo l'elemento **el.**

2) **Controllo degli Elementi**:

- Con ***head :: tail***, esaminiamo il primo elemento della lista (head) e il resto della lista (tail).
- Se head è uguale a el, non facciamo nulla e restituiamo la lista originale (ls).

- Se head è diverso da el, utilizziamo ***head :: insert el tail:***
    - Qui insert el tail chiama ricorsivamente la funzione con il resto della lista.
    - Quando insert trova il posto giusto (ossia, quando tail diventa vuoto o quando trova el), costruisce la nuova lista con head seguito dagli elementi già elaborati.

NOTA BENE

L'operatore **::** concatena un elemento all'inizio di una lista, creando una nuova lista. Se vogliamo costruire una lista finale che contiene tutti gli elementi originali più el (se non è già presente), ogni volta che chiamiamo insert, aggiungiamo il head alla nuova lista fino a che non troviamo il punto giusto per inserire el.

Esercizio 4

insert2 : 'a -> 'a list -> 'a list

come l'esercizio precedente, ma provare a usare una funzione ricorsiva ausiliaria per evitare di passare a ogni chiamata ricorsiva l'elemento da inserire.

Soluzione:

```
let rec insert2 el ls =
    let rec aux ls =
        match ls with
        | head :: tail ->
            if head = el then ls  else head :: insert el tail
        | [] -> [el]
    aux ls

assert (insert 0 (2 :: 4 :: 3 :: []) = 2 :: 4 :: 3 :: 0 :: [])
assert (insert 3 (2 :: 4 :: 3 :: []) = 2 :: 4 :: 3 :: [])
```

Spiegazione simile **all'esercizio 3** ma con aggiunta della funzione ausiliare spiegata **nell'esercizio 2**.

Esercizio 5

odd : 'a list -> 'a list

Esempio:

```
assert (odd (7 :: 3 :: 4 :: 1 :: 2 :: 5 :: []) = 3 :: 1 :: 5 :: [])
```

Soluzione:

```
    let rec odd ls =
```

```
    match ls with
      | _ :: el2 :: tl -> el2 :: odd tl
      | _ :: [] -> []          // caso solo un elemento
      | [] -> []               // lista vuota

assert (odd (7 :: 3 :: 4 :: 1 :: 2 :: 5 :: []) = [3; 1; 5])
```

Come sempre dichiariamo una funzione ricorsiva rec e una lista ls.
La riga ***| _ :: el2 :: tl -> el2 :: odd tl*** è costituita da:

- Elemento _ che corrisponderebbe agli elementi in posizione pari, nel nostro caso non ci servono e vengono indicati con _ le variabili "inutili"
- Il secondo elemento (el2) rappresenta gli elementi in posizione dispari.
- **tl** rappresenta il resto della lista su cui si richiama **odd** in modo ricorsivo.
- ***el2 :: odd tl*** rappresenta la concatenazione della lista e andiamo ad inserire in coda tramite tl

Mentre ***| _ :: [ ] -> [ ]*** gestisce la situazione in cui rimane solo un elemento in posizione pari (primo elemento della lista, che ignoriamo).

Infine ***| [ ] -> [ ]*** Se la lista è vuota, restituiamo una lista vuota.

Esercizio 6

ordInsert : 'a -> 'a list -> 'a list

Esempio:

```
assert (ordInsert 0(1 :: 2 :: 4 :: 5 :: []) = 0 :: 1 :: 2 :: 4 :: 5[])
assert (ordInsert 3(1 :: 2 :: 4 :: 5 :: []) = 1 :: 2 :: 3 :: 4 :: 5[])
assert (ordInsert 7(1 :: 2 :: 4 :: 5 :: []) = 1 :: 2 :: 4 :: 5 :: 7[])
assert (ordInsert 2(1 :: 2 :: 4 :: 5 :: []) = 1 :: 2 :: 4 :: 5 :: [])
```

Soluzione:

```
let rec ordInsert el ls=
   match ls with
   | hd :: tl ->
      if el < hd then el :: ls
      elif el = hd then ls
      else hd :: ordInsert el tl
   | [] -> [el]
```

- **if el < hd**: Se l'elemento el è minore di hd, significa che el è più piccolo di tutti gli elementi di ls. Quindi, el viene inserito in testa e la lista risultante sarà el :: ls.

- **elif el = hd:** Se el è uguale a hd, significa che l'elemento el è già presente nella lista (poiché ls è senza duplicati). In questo caso, la funzione restituisce semplicemente la lista ls originale, senza alcuna modifica.
- **else hd :: ordInsert el tl**: Se el è maggiore di hd, significa che el dovrebbe apparire dopo hd. In questo caso, hdviene mantenuto in testa e la funzione continua ricorsivamente su tl, cercando la posizione corretta per el.

Esercizio 7

ordInsert2 : 'a -> 'a list -> 'a list

come l'esercizio precedente, ma provare a usare una funzione ricorsiva ausiliaria per evitare di passare a ogni chiamata ricorsiva l'elemento da inserire.

Soluzione:

```
let ordInsert2 el ls =
    let rec auxOrd ls =
        match ls with
        | hd :: tl ->
            if el < hd then el :: ls
            elif el = hd then ls
            else hd :: auxOrd tl
        | [] -> [el]
    auxOrd ls
```