ANCORA RICORSIONE SU LISTE

martedì 28 novembre 2017 11:18

→ abbiamo visto il pattern matching

match e with
  p1 → e1
| p2 → e2
  ⋮ ⋮
| pN → eN

↖ il risultato della valutazione dell'espressione e viene confrontato con i pattern p1, …, pN uno dopo l'altro. Appena si incontra un pattern pI che "fa match" si da come risultato il valore calcolato valutando l'espressione eI corrispondente

In ogni espressione eI si possono usare le variabili del corrispondente pattern pI
I valori di tali variabili saranno quelli che hanno consentito al pattern pI di fare match con il valore dell'espressione e

ESEMPIO: CALCOLO DELLA LUNGHEZZA DI UNA LISTA

```
# let rec len l =
      match l with
        [] -> 0
      | x::xs -> 1 + (len xs);;
```

X VIENE ISTANZIATO CON IL VALORE IN TESTA ALLA LISTA l

XS VIENE ISTANZIATO CON LA LISTA OTTENUTA TOGLIENDO IL PRIMO ELEMENTO DA l

martedì 28 novembre 2017 20:31

NOTA Per verificare se una funzione ricorsiva termina per ogni input possiamo disegnare la sua RELAZIONE DI PRECEDENZA INDOTTA

VISITA QUALCHE LEZIONE FA

ESEMPIO

len

$[x_1; x_2; \ldots; x_n]$

|

$[x_2; \ldots; x_n]$

|

$[x_3; \ldots; x_n]$

|

|

$[x_n]$

|

$[\,]$ ← CASO BASE

NON CI SONO SEQUENZE DISCENDENTI INFINITE!!

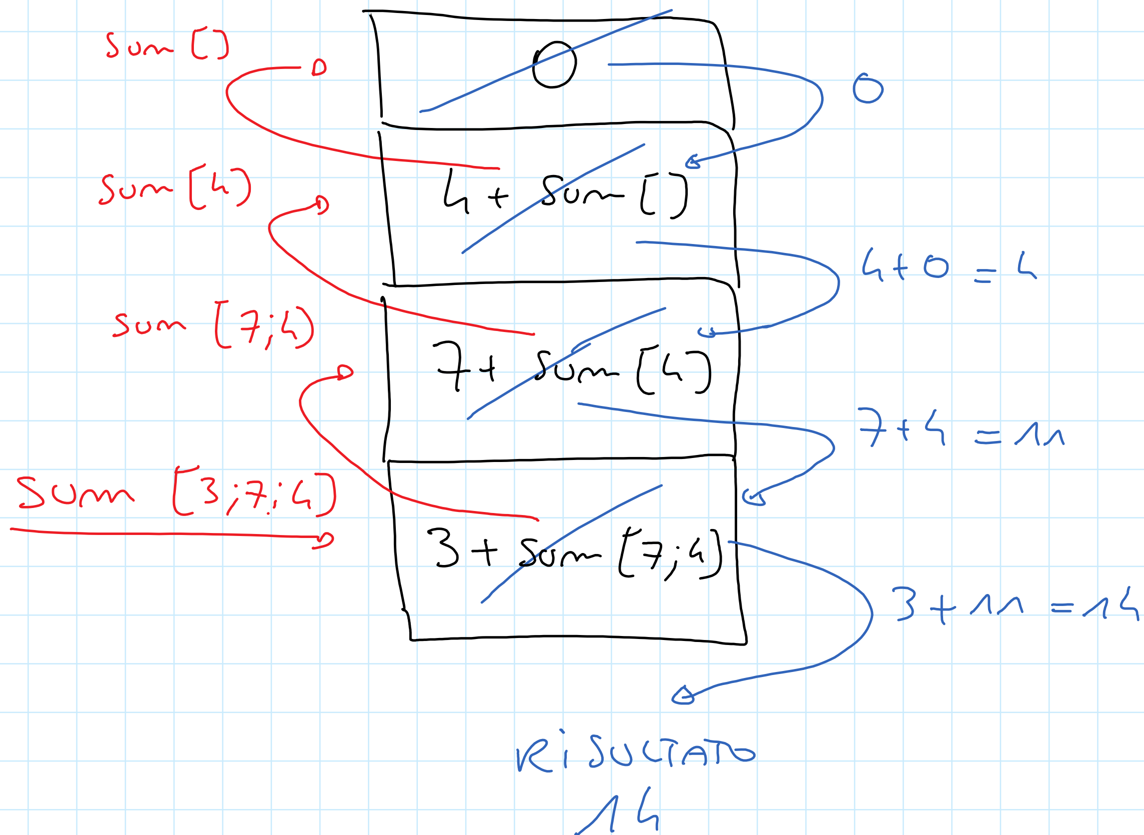
OK! TERMINO PER OGNI INPUT (RAGGIUNGE SEMPRE IL CASO BASE)

ESEMPIO: Calcolo della somma degli elementi di una lista

```
# let rec sum l =
    match l with
      [] -> 0
    | x::xs -> x + sum xs ;;
```

ESECUZIONE (PILA DI ATTIVAZIONE)

calcoliamo sum [3;7;4]



ESEMPIO: cancellare l'ultimo elemento una lista

NOTA: La lista passata non viene modificata. La funzione restituisce una nuova lista uguale a quella passata ma senza l'ultimo elemento!!

```
let rec canc l =
    match l with
      [] -> []
    | x::[] -> []
    | x::xs -> x::(canc xs);;
```

DUE CASI BASE !! (riferito a `[] -> []` e `x::[] -> []`)

x::[] fa match con liste che contengono ESATTAMENTE un valore

x::xs farebbe match con liste contenenti ALMENO due elementi, ma visto che le liste con un solo elemento sono catturate dal pattern precedente, questo caso si applica solo a liste con almeno 2 elementi

Il tipo di canc è 'a list -> 'a list
(si applica a liste di qualunque tipo)

NOTA: che cosa succede se scambio l'ordine del 2° e 3° caso nel match? (prima x::xs e poi x::[]) il caso x::[] non verrebbe mai raggiunto (l'interprete segnala questa situazione con un messaggio di WARNING) e la funzione restituirebbe l'intera lista

martedì 28 novembre 2017 20:52

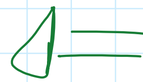
<u>ESEMPIO</u>: funzione che restituisce l'ultimo elemento di una lista

```
# let rec last l =
    match l with
      x::[] -> x
    | x::xs -> last xs;;
```

CASO BASE (x::[] -> x)

il caso della lista vuota [] non è contemplato

PATTERN MATCHING NON ESAUSTIVO (NON CONSIDERA TUTTI I CASI POSSIBILI



La funzione NON È DEFINITA per la lista vuota (come ho detto)

OK, in questo caso ha senso!!

L'interprete avvisa di questa situazione con un messaggio di WARNING.
La non esaustività potrebbe essere voluta (come in questo caso) ma potrebbe essere anche un errore di programmazione

ESEMPIO: Calcolo dell'elemento massimo

```
# let rec max l =
    match l with
      x::[] -> x
    | x::xs -> if (x > max xs)
                 then x else max xs;;
```

OK, MA CHIAMO 2 VOLTE RICORSIVAMENTE MAX SU XS (PERDO TEMPO)

-> MEGLIO COSÌ:

```
let rec max l =
    match l with
      x::[] -> x
    | x::xs -> let m = max xs
               in
               if (x>m) then x else m;;
```

DICHIARAZIONE LOCALE (-> let m)

UNA SOLA VOLTA (-> max xs)

martedì 28 novembre 2017 21:04

ESEMPIO: Funzione che restituisce una coppia costituita dal minimo e dal massimo nella lista

SOLUZIONE 1: min e max come funzioni ausiliarie (ricorsive)

QUESTA NON È RICORSIVA (NON RICHIAMA SE STESSA) → minmax

```
# let minmax l =
      let rec min l1 =
         match l1 with
           x::[] -> x
         | x::xs -> let m = min xs
                    in if (x<m) then x else m
      in
      let rec max l1 =
         match l1 with
           x::[] -> x
         | x::xs -> let m = max xs
                    in if (x>m) then x else m
      in
         (min l, max l);;
```

minmax restituisce la coppia formata dai risultati delle chiamate di min e di max

OK, MA LA LISTA VIENE SCANDITA TUTTA 2 VOLTE (UNA PER CALCOLARE min E UNA PER max)

SOLUZIONE 2 : calcoliamo min e max insieme, ricorsivamente

```
# let rec minmax l =
    match l with
      x::[] -> (x,x)
    | x::xs -> let p = minmax xs
               in
               if (x<fst p) then (x, snd p)
               else if (x> snd p)
                    then (fst p, x)
                    else p ;;
```

Coppia!! (p)

RISCRIVO IL SECONDO CASO IN MODO DIVERSO

DECOMPONGO SUBITO LA COPPIA NEI SUOI DUE ELEMENTI!!

```
| x::xs -> let (p1,p2) = minmax xs
           in
           if (x < p1) then (x, p2)
           else if (x> p2) then (p1,x)
                           else (p1,p2) ;;
```

NOTA IMPORTANTE

Il tipo di questa funzione è
'a list -> 'a * 'a

Questo tipo è stato inferito dai pattern e dalle espressioni ad essi associate.

REGOLA: In un pattern matching

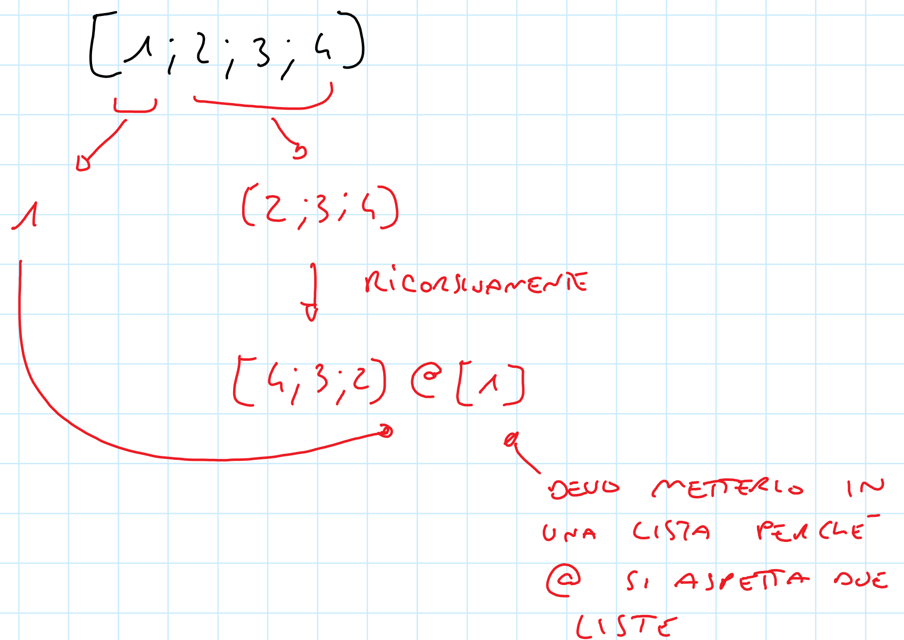
```
match e with
  p1 -> e1
| p2 -> e2
  ⋮      ⋮
| pN -> eN
```

1. TUTTI I PATTERN p1, ..., pN DEVONO AVERE LO STESSO TIPO, CHE DEVE ESSERE LO STESSO DI e
2. TUTTE LE ESPRESSIONI e1, ..., eN DEVONO AVERE LO STESSO TIPO

ESEMPIO : Rovesciare una lista

SOLUZIONE 1 : Semplice ma inefficiente

IDEA:
- Tolgo il primo elemento
- mi chiamo ricorsivamente sul resto della lista
- riaggiungo il primo elemento in fondo usando @ (append)



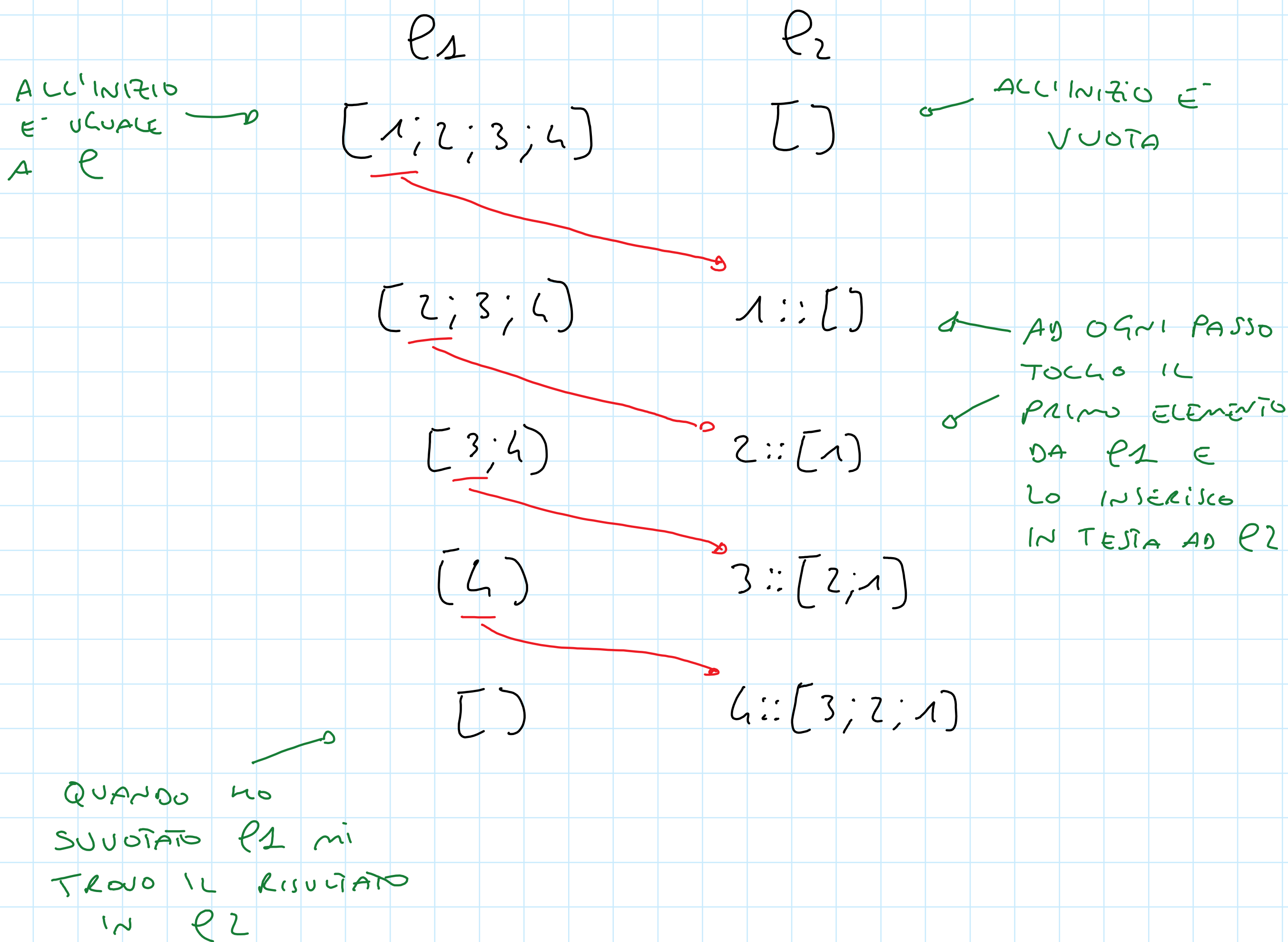

```
# let rec reverse l =
    match l with
      [] -> []
    | x::xs -> (reverse xs)@[x] ;;
```

NOTA SULL'EFFICIENZA:

@ PER AGGIUNGERE [x] IN CODA OGNI VOLTA DEVE SCORRERE TUTTA LA LISTA DA CAPO.....

SOLUZIONE 2 : uso una funzione ricorsiva CON ACCUMULATORE

Idea:
- uso due liste l1 ed l2
- l1 è l'input
- l2 è l'output, che costruisco un passo alla volta



devo definire una funzione ricorsiva con 2 parametri: l1 ed l2.
La definisco come funzione locale all'interno di reverse (che invece prevede il solo parametro l)

reverse non è ricorsiva!!

PARAMETRO DI ACCUMULAZIONE (DEL RISULTATO)

sposto x da l1 a l2

```
# let reverse l =
    let rec reverse_accum l1 l2 =
      match l1 with
        [] -> l2
      | x::xs -> reverse_accum xs x::l2
  in
    reverse_accum l [] ;;
```

inizialmente l'accumulatore è vuoto!!

La seconda versione di reverse è più efficiente perché ogni elemento spostato viene inserito IN TESTA (senza scorrere la lista.

Per provarlo:

```
# let rec crealista n =
      if n=0 then [0]
             else n :: (crealista (n-1));;
```

CREA UNA LISTA CON n ELEMENTI

```
# let listalunga = crealista 10000 ;;
```

PROVARE A PASSARE listalunga ALLE DUE VERSIONI DI reverse .... IMPIEGANO LO STESSO TEMPO ??