Funzioni su liste CAML

Titolo nota 24/09/2015

take n l restituisce la lista dei primi n elementi di l (se ci sono)

take 2 [3;4;-1;7] = [3;4]

take 4 [3;4] = [3;4]

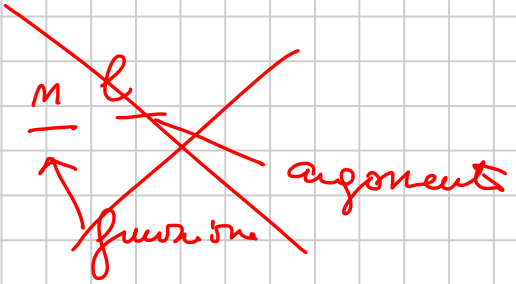
take ∅ [3;4] = []

take 3 [] = []

Titolo nota 24/09/2015

```
let rec take n l = match (n,l) with
(0,l) when l <> [] -> 0
```

l è una variabile locale al pattern che viene uguagliata al valore del parametro l



Clausola "when": il risultato viene dato da questo pattern se il pattern uguaglia i valori dell'espressione che segue "match" e la condizione dopo "when" è VERA!!

Titolo nota 24/09/2015

```
let rec take n l = match (n, l) with
    | (0, l) -> []
    | (n, []) -> []
```

I pattern non sono "mutuamente esclusivi"
(se si può uguagliare un valore a un pattern NON si può uguagliarlo all'altro)

(0, [])

Titolo nota 24/09/2015

```
# let rec take n l = match (n, l) with
    (0, l) -> []
  | (n, []) when n > 0 -> []
  | (n, x::xs) when n > 0 -> x :: take (n-1) xs ;;
```

pattern mutuamente esclusivi

take : int -> 'a list -> 'a list = <fun>

tipo di n — tipo di l — tipo del risultato

(n-1, xs) ⊏ (n, x::xs)

drop n l. Cancella dalla lista l i primi n elementi (se ci sono)

Titolo nota 24/09/2015

drop 3 [3;4;5;6] = [6] - drop 0 [3;4] = [3;4] - drop 4 [] = []

drop 3 [2;3] = []

```
# let rec drop n l = match (n, l) with
      (0, l) -> l
    | (n, []) when n>0 -> []
    | (n, x::xs) when n>0 -> drop (n-1) xs ;;
```

(n-1, xs) è (n, x::xs) ricorsivamente

drop : int -> 'a list -> 'a list = <fun>

int: tipo di n; 'a list: tipo di l; 'a list: tipo risultato

$(\forall l \in 'a\ list, n \in \mathbb{N}.\ (take\ n\ l) @ (drop\ n\ l) = l)$

Titolo nota 24/09/2015

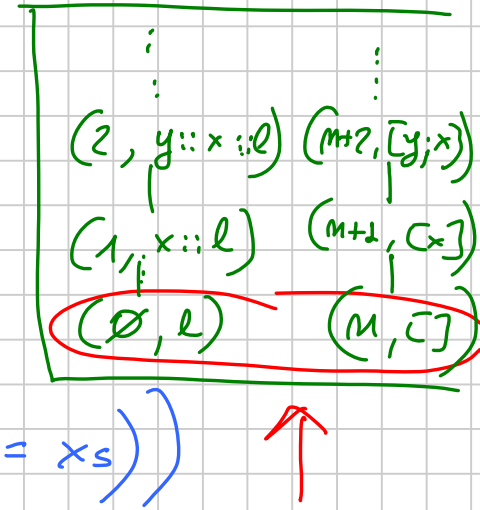
Per induzione ben fondata (precedente indotta) de quale definizione? de quella di take o quella di drop

Fortunatamente la precedenza indotta delle due definizioni è LA STESSA!

$(\forall n, n' \in \mathbb{N},\ l, l' \in 'a\ list.\ (n, l) \sqsubset (n', l') \equiv n = n' - 1 \wedge l = TL\ l')$

$\equiv$

$(\exists x \in 'a,\ xs \in 'a\ list.\ n = n' - 1 \wedge l' = x :: xs \wedge l = xs))$



$$\left(\forall n \in \mathbb{N},\ l \in \text{'a list} .\ (\text{take}\ n\ l) @ (\text{drop}\ n\ l) = l\right)$$

Titolo nota 24/09/2015

Caso base

$(\text{take}\ 0\ l) @ (\text{drop}\ 0\ l) = l$

$(\text{take}\ 0\ l) @ (\text{drop}\ 0\ l)$

$=$ { def. take, 1° p, def. drop 1° p }

$[] @ l$

$=$ { proprietà @ }

l

Caso base

$(\text{take}\ n\ []) @ (\text{drop}\ n\ []) = []$

$n \geq 0$

$(\text{take}\ n\ []) @ (\text{drop}\ n\ [])$

$=$ { def take 2° p, def drop 2° p }

$[] @ []$

$=$ { dproprietà di @ } $[]$

Caso induttivo ip. induttiva

$(\text{take}\ n\ xs) @ (\text{drop}\ n\ xs) = xs$

$\Rightarrow (\text{take}\ (n+1)\ (x :: xs)) @ (\text{drop}\ (n+1)\ (x :: xs)) = x :: xs$

$(\text{take}\ (n+1)\ (x :: xs)) @ (\text{drop}\ (n+1)\ (x :: xs))$

$=$ { def. take 3° p, def drop 3° p }

$(x :: \text{take}\ n\ xs) @ (\text{drop}\ n\ xs)$

$=$ { proprietà @ : $(x :: xs) @ ys = x :: (xs @ ys)$ }

$x :: ((\text{take}\ n\ xs) @ (\text{drop}\ n\ xs))$

$=$ { ip. induttiva

$x :: xs$

Titolo nota 24/09/2015

nth n l calcola l'n-simo elemento della lista l

è indefinita se l'elemento non esiste

nth 3 [3;4;5;6] = 5 — nth 3 [3;4] = indefinita

nth ∅ [3;4] = indefinita

≠ let rec nth n l = match (n, l) with

~~(∅, l)~~ → non va meno (quando n = ∅ la funzione non è definita)

→ ~~un boccolo~~

~~(n, [])~~ when n > 0

Titolo nota 24/09/2015

```
# let rec nth n l = match (n, l) with
    (1, x::xs) -> x
  | (n, x::xs) when n>1 -> nth (n-1) xs;;
nth : int -> 'a list -> 'a = <fun>
```

tipo n — tipo l — tipo ris.

Calcola la somma di tutti gli elementi di una lista di interi

Titolo nota 24/09/2015

sum [3; 4; -1; 7] = 13

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

sum : int list -> int = <fun>
```

int list: tipo di l — int: Tipo risultato

```
# let rec sum l =
    if l=[] then 0
    else hd l + sum(tl l);;

sum : int list -> int = <fun>
```

Ricerca del valore massimo in una lista non vuota

Titolo nota 24/09/2015

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

```
max : 'a list -> 'a = <fun>
```

[x] ≡ x::[] si uguaglia solamente a liste lunghe 1

x::y::ys si uguaglia solamente a liste con almeno due elementi

Titolo nota 24/09/2015

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

dichiarazione locale che ci permette di fare una sola chiamata ricorsiva

```
max : 'a list -> 'a = <fun>
```

```
# let (n, m) = (3,4) ;;
n, m : int * int = 3,4
```

```
# n ;;
- : int = 3
# m ;;
- : int = 4
```

```
let (n,m) = (3,4)
in n + m ;;
- : int = 7
```

Titolo nota 24/09/2015

Funzione che calcola la coppia (elemento minimo, elemento massimo) di una lista non vuota.

minmax [3] = (3,3) — minmax [3; -1; 7; 2] = (-1, 7)

```
# let rec minmax l = match l with
    [x] -> (x,x)
  | x::y::ys ->
      if x > secondo (minmax (y::ys))
        then (primo (minmax (y::ys)),
              x)
      else if x < primo (minmax (y::ys))
        then (x, secondo (minmax (y::ys)))
      else minmax (y::ys) ;;
```

```
#let primo (m,n) = m ;;
primo : 'a * 'b -> 'a = <fun>
#let secondo (m,n) = n ;;
secondo : 'a * 'b -> 'b = <fun>
```

Titolo nota 24/09/2015

Funzione che calcola la coppia (elemento minimo, elemento massimo) di una lista non vuota.

minmax [3] = (3,3) — minmax [3; -1; 7; 2] = (-1, 7)

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

minmax : 'a list -> 'a * 'a = <fun>
```

Invertire l'ordine degli elementi di una lista (reverse)

Titolo nota 24/09/2015

reverse [3;4;5] = [5;4;3] reverse [] = [] reverse [3] = [3]

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

molto inefficiente

reverse : 'a list → 'a list = <fun>

'a :: 'a list

~~'a list :: 'a~~

r [3;4;5]
= {def r, 2° p}
r [4;5] @ [3]
= { " }
(r [5] @ [4]) @ [3]
= { " }

((([] @ [5]) @ [4]) @ [3])
((([] @ [5]) @ [4]) @ [3])

concatenazione (ricorda append) scorre tutti gli elementi della lista primo argomento

Reverse con accumulatore

Titolo nota 24/09/2015

```
let reverse l =
    let rec rev l a = match l with
        [] -> a
      | x::xs -> rev xs (x::a)
    in rev l [];;

reverse : 'a list -> 'a list = <fun>
```

[7;8]

```
rev [3;4;5] []
= {def rev, 2° p}
rev [4;5] [3]
= {def rev, 2° p}
rev [5] [4;3]
= {def rev, 2° p}
rev [] [5;4;3]
= {def rev, 1° p}
[5;4;3]
```

[5;4;3;7;8]