Semantica formale del linguaggio imperativo

int * x;

$\nu : Loc \to Val_\perp$

$Val = \mathbb{N}$

$\nu : Loc \to (Val \cup Loc)_\perp$

Valori, ma anche indirizzi

nuovo tipo

$Exp \to \ldots \mid *Ide \mid \&Ide$

$Dec \to \ldots \mid Type * Ide; \mid$ non serve

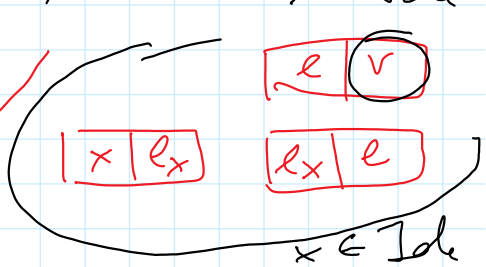
$Com \to \ldots \mid *Ide = Exp;$

$Type \to int \mid float \mid int* \mid float*$

$$Sem_e : Exp \to P \to M \to (Val \cup Loc)_\perp$$

$$Sem_e\ (*x)\ \rho\ \mu = \mu\ (\mu\ (\rho\ x))$$

ℓ = $\mu(\rho\ x)$, ℓ_x = $\mu(\rho\ x)$, v = $\mu\ (\mu\ (\rho\ x))$



$$Sem_e\ (\&x)\ \rho\ \mu = \rho(x)$$

$$Sem_c : Com \to P \to M \to M$$

$$Sem_c\ (*x = e;)\ \rho\ \mu = \mu\left[\ v \Big/ \mu(\rho\ x)\ \right]^{mod}$$

$x \in Ide$, $e \in Exp$

$\mu(\rho\ x)$ = ℓ_x

dove

$$v = Sem_e\ e\ \rho\ \mu$$

giovedì 7 dicembre 2017 16:26

Cancellare gli elementi ripetuti più di una volta in una lista lasciandone una sola occorrenza. (canc)

canc [2; 3; 5; 2; 7; 3; 2] = [2; 3; 5; 7]

Ci serve la funzione

member : 'a → 'a list → bool

pensiamola in modo ricorsivo

```
let rec member a l = match l with
    [] -> false
  | x::xs -> if a=x then true
             else member a xs;;
```

Ha senso definirla con un accumulatore?

No! perché non appena troviamo l'elemento o arriviamo alla lista vuota sappiamo dare il risultato senza fare ulteriori operazioni.

giovedì 7 dicembre 2017 16:34

Con exists

```
let rec exists p l = match l with
    [] -> false
  | x::xs -> if p x then true
             else exists p xs ;;
```

```
let member a l =
    let p x = x = a
    in exists p l ;;
```

def. — uguaglianza

```
let member a l =
    let f x y = if x = a then true
                else y
    in fold_left f false l ;;
```

giovedì 7 dicembre 2017 16:43

Cancellare le occorrenze multiple degli elementi di una lista

Ricorsione pura

```
let rec canc l = match l with
    [] -> []
  | x::xs -> if member x xs
             then canc xs
             else x::canc xs;;

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

non è "tail recursive"

Con accumulatore

giovedì 7 dicembre 2017 16:48

```
let conc l =
    let rec conca l a = match l with
        [] -> a
      | x::xs -> if member x xs
                 then conca xs a
                 else conca xs (x::a)
    in conca l [];;
```

Conc utilizzando folder

giovedì 7 dicembre 2017 16:52

```
let conc l =
    let f x y = if member x y
                then y
                else x::y
    in folder f [] l ;;
```

giovedì 7 dicembre 2017 16:55

Data una lista restituire la coppia

(valore massimo, numero di occorrenze di questo valore)

max [2; 7; 3; 7; -2; -2; 7] = (7, 3)

(7, 3): 7 = massimo, 3 = n. di occorrenze

```
let rec max l = match l with
    [x] -> (x, 1)
  | x::y::ys -> let (m,n) = max (y::ys)
                in if x=m then (m, n+1)
                   else if x>m then (x, 1)
                   else (m, n);;
```

non è tail ricorsiva

max : 'a list -> 'a * int = <fun>

giovedì 7 dicembre 2017 17:03

Con accumulatore

```
let max l =
    let maxa l (m,n) = match l with
        [] -> (m,n)
      | x::xs -> if x > m
                 then maxa xs (x,1)
                 else if x=m
                      then maxa xs (m,n+1)
                      else maxa xs (m,n)
    in maxa (tl l) (hd l,1);;
```

```
  max [2;-1;2]
= {def max}
  maxa [-1;2] (2,1)
= {def maxa}
```

a

2° p

maxa [2] (2,1)

= {def maxa, 2° p}

maxa [] (2,2)

= {def maxa, 1° p}

(2,2) ←

---

max [2];;

= {def max}

maxa [] (2,1)

= {def. maxa, 1° p}

(2,1)

---

max [3;7]

= {def. max}

maxa [7] (3,1)

= {def

giovedì 7 dicembre 2017 17:19

```
let max l =
  let maxa l (m,n) = match l with
      [] -> (m,n)
    | x::xs -> if x>m then maxa xs (x,1)
               else if x=m
                    then maxa xs (m,n+1)
                    else maxa xs (m,n)
  in maxa (tl l) (hd l, 1) ;;
```

≡

```
match l with
  x::xs -> maxa xs (x,1)
```

```
let max l =
    let f x y =
        let (m,n) = y in ... ---
```

ok

in

```
let max l =
    let f x (m,n) =
        if x > m then (x,1)
        else if x = m then (m,n+1)
        else (m,n)
in foldr f (hd l,1) (tl l)
```

≡

```
match l with
x::xs -> foldr f (x,1) xs
```

x::xs ⟶ foldr f (x, 1) xs