LISTE

OCaml offre vari <u>tipi di dato</u> (datatype)

- liste
- array
- record
- alberi

⋮

Le liste in OCaml sono scritte tra parentesi quadre

[ ] $\rightsquigarrow$ α list

[1] $\rightsquigarrow$ int list

[1;2] ; per separare gli elementi

[1;2;3]

[1.;2.,3.] $\rightsquigarrow$ float list

NB. Tutti gli elementi di una lista devono avere lo stesso tipo

T list = tipo delle liste con elementi di tipo T

Es. [[1]; [1; 2], [ ]] $\underbrace{\text{int list list}}_{\text{liste annidate}}$

Operatore concat

$$\underbrace{t}_{T} :: \underbrace{ts}_{T\ list} : T\ list$$

$$1 :: [1; 2] = [1; 1; 2]$$

NB. $[t_1; \dots; t_n] \approx t_1 :: (t_2 :: (\dots t_n :: [\,])\dots))$

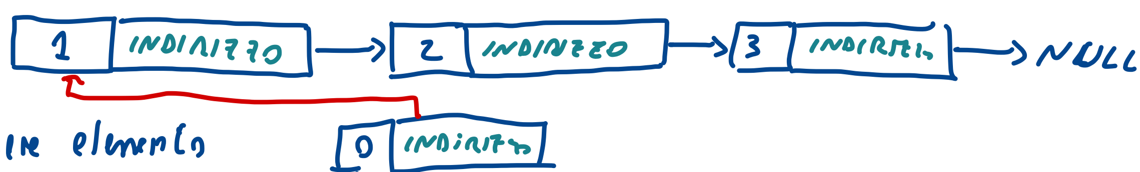
NB 2 Le liste sono **immutabili**

```
let xs = [1; 2; 3]
in 0 :: xs
```

↳ non ho aggiunto 0 ad xs, ma solamente creato una nuova lista

Vantaggi. Implementazione in memoria

$[1; 2; 3] \qquad (1 :: (2 :: (3 :: [\,])))$



Per aggiungere elemento

SINTASSI $\quad t, s ::= \ \dots \ \mid \ \underset{\text{nil}}{[\ ]} \ \mid \ t \underset{\text{cons}}{::} s$

zucchero sintattico $\quad [t_1; .. ; t_n] = t_1 :: (t_2 : \dots (t_n :: [\ ])..))$

DINAMICA $\quad v, w ::= [\ ] \mid v :: w$

$$\frac{}{[\ ] \Rightarrow [\ ]} \qquad \frac{t \Rightarrow v \qquad s \Rightarrow w}{t :: s \Rightarrow v :: w}$$

Esercizio. Inferire la dinamica di: $[t_1, \dots ; t_n]$

STATICA

$$\tau, \sigma ::= \underbrace{int \mid bool}_{\text{tipi costanti/base}} \mid \ldots \underbrace{\alpha \mid \beta}_{\text{variabili di tipo}} \mid \ldots \mid \underbrace{\tau \to \sigma \mid \tau\ list}_{\text{costruttori di tipo}}$$

$$\frac{}{[\,] : \alpha\ list} \qquad \frac{t : \tau \qquad s : \tau\ list}{t :: s : \tau\ list}$$

<u>Domanda</u> Nil e cons ci dicono come creare liste. Ma se abbiamo una lista, come possiamo usarla?

⇝ pattern matching

```
let sum (ns : int list) : int =
  match ns with
  | [] -> 0
  | x :: xs -> x + sum xs
```

→ ogni valore di tipo int list è

(i) [] (nil)

(ii) oppure della forma ::

testa coda

⟹ se dico cosa fare in questi due casi, ho detto cosa fare per ogni lista

Es.

$\text{sum } [1;2;3]$

$= \text{sum } 1 :: [2;3]$ } ha forma $x :: xs$, con $x = 1$, $xs = [2;3]$ }

$\rightarrow 1 + \text{sum } [2;3]$

$= 1 + \text{sum } 2 :: [3]$ } ha forma $x :: xs$, con $x = 2$, $xs = [3]$ }

$\rightarrow 1 + 2 + \text{sum } [3]$

$= 1 + 2 + \text{sum } (3 :: [\,])$ } ha forma $x :: xs$, con $x = 3$, $xs = [\,]$ }

$\rightarrow 1 + 2 + 3 + \text{sum } [\,]$ } ha forma $[\,]$ }

$\rightarrow 1 + 2 + 3 + 0$

$\rightarrow 6$

Il pattern matching fa un matching del valore con un pattern sintattico

```
let flip_two xs =
    match xs with
      [] -> []
      y1 :: y2 :: ys -> y2 :: y1 :: ys
```

Domande:

- Qual è il tipo di flip_two?
- Cosa succede se eseguo flip_two [2]?

  ⇝ pm non esaustivo

- È flip_two zucchero sintattico per PM della forma

```
[] -> ...
x::xs -> ...
```
?

```
let flip_two xs =
    match xs with
        [ ] -> [ ]
        y1 :: ys -> match ys with
                        [ ] -> caso mancante
                        y2 :: ys -> y2 :: y1 :: ys
```

Es.

```
let first_five (n : int) : bool =
    match n with
        | 0 -> true
        | 2 -> true
        | 2 -> true
        | 3 -> true
        | 4 -> true
        | _ -> false
```

wildcard : si usa per il match triviale con tutto il resto

Liste.

SINTASSI $t, s ::= \dots \mid [\,] \mid t :: s \mid \text{match } t \text{ with } \underbrace{p_1 \to t_1 \mid \dots \mid p_n \to t_n}_{\text{pattern}}$

$p, q ::= x \mid \_ \mid [\,] \mid p :: q$

(le variabili possono apparire una volta sola nel pattern: e.g. no $x :: x$)

STATICA $T, S ::= \dots \mid T \to S \mid T \text{ list}$

$$\frac{}{\Gamma \vdash [\,] : T \text{ list}} \qquad \frac{\Gamma \vdash t : T \qquad \Gamma \vdash s : T \text{ list}}{\Gamma \vdash t :: s : T \text{ list}}$$

$$\frac{\Gamma \vdash t : T \qquad \Gamma \vdash p_i : T \qquad \Gamma \vdash s_i : S}{\Gamma \vdash \text{match } t \text{ with } p_1 \to s_1 \mid \dots \mid p_n \to s_n : S}$$

DINAMICA

Abbiamo bisogno di: introdurre una nozione di match tra un pattern e un valore

Definiamo la relazione

$$\text{match}(\text{pattern}, \text{valore}, \text{binding})$$

Intuizione: $\text{match}(p, v, [x_1 \mapsto v_1, \ldots, x_n \mapsto v_n])$

"il valore v ha la struttura descritta dal pattern p, e $v = p[v_1/x_1 \ldots v_n/x_n]$"

$$\frac{}{Match\,(x,\ v,\ [x \mapsto v])}$$

$$\frac{}{Match\,(\_,\ v,\ \cdot)}$$

$$\frac{}{Match\,([\,],\ [\,],\ \cdot)}$$

$$\frac{Match\,(p_1,\ v_1,\ [x_1 \mapsto v_1, \dots, x_n \mapsto v_n]) \qquad Match\,(p_2, v_2,\ [y_1 \mapsto w_1 \dots y_m \mapsto w_m])}{Match\,(p_1 :: p_2,\ v_1 :: v_2,\ [\underbrace{x_1 \mapsto v_1 \dots x_n \mapsto v_n,\ y_1 \mapsto w_1 \dots y_m \mapsto w_m}_{\text{perché ok?}}])}$$

$$\overbrace{\boxed{Match(p_i, v, [x_1 \mapsto v_1, \dots x_n \mapsto v_n])}}^{\text{pattern matching rispetta l'ordine}}$$

$$\frac{t \Rightarrow v \qquad \forall j < i.\ \neg Match(p_j, v, \sigma) \qquad s[v_1/x_1, \dots, v_n/x_n] \Rightarrow w}{\mathtt{match}\ t\ \mathtt{with}\ p_1 \to s_1 \mid \dots \mid p_n \to s_n \Rightarrow w}$$

Se non si riesce a fare il match, viene sollevata una eccezione

Programmazione con Liste

Riscaldamento

```
let rec length (l : α list) : int =
    match l with
      [] -> 0
      x::xs -> 1 + length xs
```

```
let rec append l1 l2 =
    match l1 with
      [] -> l2
      x::xs -> x::(append xs l2)
```

@ : α list → α list → α list

MAP

```
let rec map f l =
    match l with
      [] -> []
      x::xs -> f x :: map f xs
```

$$\text{map} : \underbrace{(\alpha \to \beta) \to \alpha\ \text{list} \to \beta\ \text{list}}_{\text{higher-order}}$$

Ex. map (fun x => x+1) [1; 2; 3]

$\Rightarrow$ [2; 3; 4]

map length [[ ]; [1]; [1; 2]]

$\Rightarrow$ [0; 1; 2]

map è fondamentale per la parallelizzazione

. supporta molte ottimizzazioni

$$\underbrace{\text{map } g \ (\text{map } f \ xs)}_{\text{2 passate di } xs} = \overbrace{\text{map } (\underbrace{g \cdot f}_{\text{function composition}}) \ xs}^{\text{1 passata}}$$

Importante: f, g devono essere **pure** !

Filter

Un predicato è una funzione $p : T \rightarrow bool$

```
let rec filter p l =
    match l with
      [] -> []
      x::xs -> match p x with
                  false -> filter p xs
                  true -> x :: (filter p xs)
```

$filter : (\alpha \rightarrow bool) \rightarrow \alpha\ list \rightarrow \alpha\ list$

Ex.

```
let dispari n = n mod 2 <> 0
in
filter dispari [1; ...; 10]
```

Fold

Funzioni di ordine superiore per fare ricorsione su liste in modo efficiente

Idea

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

la chiamata ricorsiva non è un sum (...) ma una espressione più complessa che contiene sum(..) (recursive call)

⟹ Per calcolare x + sum xs, devo prima calcolare sum xs

⟹ computazione in sospeso

sum [1;2;3]

→ 1 + sum [2;3]

→ 1 + (2 + sum [3])

→ 1 + 2 + (3 + sum [])

→ 1 + (2 + 3)

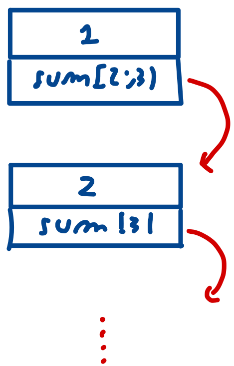
→ 1 + 5

→ 6

riflette quello che succede in memoria

Per calcolare sum [1;2;3] alloco spazio in memoria record di attivazione etc...

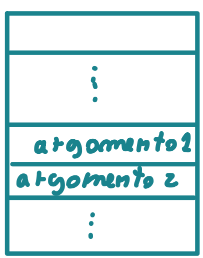
ciò che calcolo qui è + (1, sum [2;3]) e per fare ciò devo conoscere gli argomenti...



iterazione 1

iterazione 2

alloco spazio per ogni chiamata ricorsiva; arrivato al caso base propago all'indietro il risultato e termino le computazioni sospese



1

sum [2;3]

per eseguire + (1, sum [2;3]) devo calcolare prima sum [2;3]
$\Longrightarrow$ altro spazio in memoria, etc...

accumulatore

```
let rec sum_tail l acc =
    match l with
      [] -> acc
      x::xs -> sum_tail xs (x+acc)
```

$\underbrace{\texttt{sum\_tail xs (x+acc)}}_{\text{espressione in cui la chiamata ricorsiva è in testa}}$

```
sum_tail [1;2;3] 0  -> sum_tail [2;3] (1+0)
                    -> sum_tail [2;3] 1
                    -> sum_tail [3] (2+1)
                    -> sum_tail [3] 3
                    -> sum_tail [] (3+3)
                    -> sum_tail [] 6
                    -> 6
```