Programmazione Funzionale

http://cialdea.dia.uniroma3.it/teaching/pf/

Linguaggi funzionali

- di alto livello: un programma è una funzione.
- di tipo dichiarativo: il programmatore specifica **che cosa** calcola la funzione, piuttosto che **come**.

Objective Caml

- Linguaggio della famiglia ML sviluppato e distribuito dall'INRIA (Francia) dal 1984
- Supporta diversi stili di programmazione: funzionale, imperativo, a oggetti.

Paradigmi di programmazione

Diverse tipologie di linguaggi di programmazione.
Sotto ogni linguaggio c'è un **modello di calcolo**, che determina

- le operazioni eseguibili
- una classe di linguaggi
- uno stile di programmazione ("paradigma" di programmazione)

Modelli astratti e modelli rappresentati dall'hardware

Linguaggi dichiarativi
Un programma è più vicino alla descrizione di

- **che cosa** si deve calcolare,
- piuttosto che a **come** calcolare
  (linguaggi **imperativi**: assegna il valore ... alla variabile x, poi entra in un ciclo fino a che ...., fai questo e quello, ... alla fine riporta il valore della variabile ...).

Esempio: calcolo del fattoriale

Cos'è il fattoriale di un numero naturale?

$$\begin{aligned} n! &= 1 \times 2 \times \ldots \times n-1 \times n \\ &= (n-1)! \times n \end{aligned}$$

Caso particolare:

$$0! = 1$$

```
(* fact: int -> int *)
let rec fact n =
  if n=0
  then 1
  else fact(n-1) * n
```

~~COMANDI~~, ~~ASSEGNAZIONE~~, ~~CICLI~~

I costrutti di controllo fondamentali sono

- applicazione di funzioni : `f(x)` o semplicemente `f x`
- composizione di funzioni : `f (g x)`
- ricorsione

Modalità interattiva: ciclo READ - EVAL - PRINT

```
# pred 6;;
- : int = 5
# abs 7;;
- : int = 7
# abs (-7);;
- : int = 7
# 6 * 7;;
- : int = 42
```

```
# let numero = 6*7;;
val numero : int = 42
# succ (pred (pred numero));;
- : int = 41
# let rec fact n =
  if n=0 then 1
  else n * fact(n-1);;
    val fact : int -> int = <fun>
# fact 4;;
- : int = 24
```

- READ: il compilatore legge un' espressione o una dichiarazione
- EVAL: calcola il valore dell'espressione e ne **deduce** il tipo (o "memorizza" la dichiarazione)
- PRINT: stampa il tipo e il valore dell'espressione (o della variabile "dichiarata")

Inferenza di tipi

OCaml può dedurre (inferire) qual è il tipo di un'espressione senza bisogno di dichiarazioni esplicite

```
# if numero > 0 then "pippo" else "pluto";;
- : string = "pippo"
# let double n = 2 * n;;
val double : int -> int = <fun>
```

OCaml è un linguaggio a tipizzazione statica : ogni espressione ha un tipo che può essere **determinato a tempo di compilazione**.

Tipi semplici: bool, int, float, string, char, unit, exn
Prodotto cartesiano (tipo delle coppie): $T_1 \times T_2$
Tipo delle funzioni: *Dominio* $\rightarrow$ *Codominio*

Espressioni

- I costrutti di base sono espressioni (non comandi)
- Le espressioni hanno sempre un **valore** e un **tipo**
- Il calcolo procede **valutando espressioni**, non ci sono effetti collaterali
  Valutare un'espressione = semplificarla fino ad ottenere un'espressione non più semplificabile, cioè un valore.

```
# double((3+7) * (32 mod 5));;
- : int = 40
```

```
 double((3+7) * (32 mod 5)))
       ⇒ double (10 * (32 mod 5))
       ⇒ double (10 * 2)
       ⇒ double 20
       ⇒ 40
```

Dichiarazioni: sintassi

```
let numero = 6*7
```

Forma generale di una dichiarazione

let <identificatore> = <espressione>

Dichiarazione di funzione:

```
let double = function n -> 2 * n
```

O anche, con una sintassi più familiare:

```
let double n = 2 * n
```

let <identificatore> <parametri> = <espressione>

Per dichiarare funzioni ricorsive:

let rec <identificatore> <parametri> = <espressione>

Ambiente di valutazione

Un ambiente è una collezione di **legami** variabile-valore:

var_n	val_n
var_{n-1}	val_{n-1}
...	...
var_1	val_1

Ambiente iniziale:

...	...
not	*function x* $\rightarrow$ ...
...	...
mod	*function x* $\rightarrow$ *function y* $\rightarrow$ ...
...	...

Ambiente del **modulo *Stdlib***

Estensione dell'ambiente mediante dichiarazioni

La valutazione di una dichiarazione aggiunge un nuovo legame all'ambiente:

```
# let two = 2;;
val two : int = 2
```

two	2
ambiente del modulo Stdlib	

```
# let three = two + 1;;
val three : int = 3
```

- Viene valutata l'espressione `two + 1` nell'ambiente esistente, dove il valore di `two` è 2;
- Viene creato un nuovo legame, di `three` con il valore di `two + 1`

three	3
two	2
ambiente del modulo Stdlib	

Valore di una variabile in un ambiente

L'ambiente viene gestito come una **pila**: i nuovi legami vengono aggiunti in alto e si cerca il valore delle variabili a partire dall'alto.

```
# let two = "due";;
val two : string = "due"
```

two	`"due"`
three	3
two	2
ambiente del modulo Stdlib	

In questo ambiente, il valore di `two` è `"due"`: il nuovo legame nasconde il vecchio.

Scopo statico

Il valore delle variabili (globali) in una dichiarazione viene determinato a tempo di compilazione


```
# let six=6;;
```

Scopo statico

Il valore delle variabili (globali) in una dichiarazione viene determinato a tempo di compilazione

sixtimes	function n -> six * n
six	6
ambiente del modulo Stdlib	

```
# let six=6;;
# let sixtimes n=
  six*n;;
```

Scopo statico

Il valore delle variabili (globali) in una dichiarazione viene determinato a tempo di compilazione

six	50
sixtimes	function n -> six * n
six	6
ambiente del modulo Stdlib	

```
# let six=6;;
# let sixtimes n=
  six*n;;
# let six=50;;
```

Scopo statico

Il valore delle variabili (globali) in una dichiarazione viene determinato a tempo di compilazione

six	50
sixtimes	function n -> six * n
six	6
ambiente del modulo Stdlib	

```
# let six=6;;
# let sixtimes n=
  six*n;;
# let six=50;;
# six;;
- : int = 50
# sixtimes 3;;
- : int = 18
```

Il valore di `six` nel corpo di `sixtimes` viene cercato **nell'ambiente di dichiarazione** di `sixtimes`, cioè nell'ambiente in cui `sixtimes` è stata definita.

Chiamata di funzioni

sixtimes	function n -> six * n
six	6
...	...

```
# let six=6;;
# let sixtimes n = six * n;;
# sixtimes (2+1);;
```

Quando la funzione `sixtimes` viene applicata a un argomento:

- il suo argomento viene valutato nell'ambiente

Chiamata di funzioni

n	3
sixtimes	function n -> six * n
six	6
...	...

```
# let six=6;;
# let sixtimes n = six * n;;
# sixtimes (2+1);;
```

Quando la funzione `sixtimes` viene applicata a un argomento:

- il suo argomento viene valutato nell'ambiente
- viene creato un nuovo legame **provvisorio** del parametro formale `n` con il valore dell'argomento
- in questo nuovo ambiente viene valutato il corpo della funzione, `six * n`

Chiamata di funzioni

`sixtimes`	`function n -> six * n`
`six`	`6`
`...`	`...`

```
# let six=6;;
# let sixtimes n = six * n;;
# sixtimes (2+1);;
- : int = 18
```

Quando la funzione `sixtimes` viene applicata a un argomento:

- il suo argomento viene valutato nell'ambiente
- viene creato un nuovo legame **provvisorio** del parametro formale `n` con il valore dell'argomento
- in questo nuovo ambiente viene valutato il corpo della funzione, `six * n`
- il legame provvisorio viene cancellato.

Polimorfismo

```
# let first (x,y) = x;;
val first : 'a * 'b -> 'a = <fun>
# first (true,0);;
- : bool = true
# first(0,"pippo");;
- : int = 0
```

- `'a` e `'b` sono **variabili di tipo**
  (a mano usiamo le lettere greche α, β, ...)
- `first` si può applicare a qualsiasi coppia, ha un numero infinito di tipi
- Tipo di "qualsiasi coppia": $\alpha \times \beta$ (`'a * 'b`)

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# id 6.3;;
- : float = 6.3
# id "pluto";;
- : string = "pluto"
```

Le funzioni sono oggetti di prima classe

```
# double;;
- : int -> int = <fun>
```

- `double` è una funzione

Le funzioni sono oggetti di prima classe

```
# double;;
- : int -> int = <fun>
# (double,"pippo");;
- : (int -> int) * string = (<fun>, "pippo")
```

- `double` è una funzione
- Le funzioni possono essere **componenti di una struttura dati**

Le funzioni sono oggetti di prima classe

```
# double;;
- : int -> int = <fun>
# (double,"pippo");;
- : (int -> int) * string = (<fun>, "pippo")
# first(double,"pippo");;
- : int -> int = <fun>
```

- `double` è una funzione
- Le funzioni possono essere **componenti di una struttura dati**
- Le funzioni possono essere **argomenti** di altre funzioni

Le funzioni sono oggetti di prima classe

```
# double;;
- : int -> int = <fun>
# (double,"pippo");;
- : (int -> int) * string = (<fun>, "pippo")
# first(double,"pippo");;
- : int -> int = <fun>
# let times n = function m -> n * m;;
val times : int -> int -> int = <fun>
```

- `double` è una funzione
- Le funzioni possono essere **componenti di una struttura dati**
- Le funzioni possono essere **argomenti** di altre funzioni
- Le funzioni possono essere **valori** riportati da altre funzioni

Le funzioni sono oggetti di prima classe

```
# double;;
- : int -> int = <fun>
# (double,"pippo");;
- : (int -> int) * string = (<fun>, "pippo")
# first(double,"pippo");;
- : int -> int = <fun>
# let times n = function m -> n * m;;
val times : int -> int -> int = <fun>
# let double = times 2;;
val double : int -> int = <fun>
# (times 3) 5;;
- : int = 15
```

- `double` è una funzione
- Le funzioni possono essere **componenti di una struttura dati**
- Le funzioni possono essere **argomenti** di altre funzioni
- Le funzioni possono essere **valori** riportati da altre funzioni

Uso delle parentesi

- Nei tipi (**espressioni di tipo**)

```
int -> int -> int    =    int -> (int -> int)
```

  si associa a destra

- Nelle **espressioni**

```
(times 3) 5    =    times 3 5
```

  si associa a sinistra

```
# let square n = n* n;;
val square : int -> int = <fun>
# double square 3;;
Characters 0-6:
  double square 3;;
  ^^^^^^
Error: This function is applied to too many arguments;
maybe you forgot a `;'
# double (square 3);;
- : int = 18
```

Funzioni di ordine superiore

Prendono come argomento o riportano come valore una funzione

```
(* sum : (int -> int) -> (int * int) -> int *)
(* sum f (lower,upper) = (f lower) + (f (lower+1)) +
                                  ... + (f upper) *)
let rec sum f = function
    (lower,upper) -> if lower > upper then 0
                     else f lower + sum f (lower +1,upper)
```

O anche

```
let rec sum f (lower,upper) =
  if lower > upper then 0
  else f lower + sum f (lower +1,upper)
```

Tipo di sum

```
sum:  (int -> int) -> ((int * int) -> int)
sum double:  (int * int) -> int
sum double (3,5):  int
```

sommatoria dei doppi

6 + 8 + 10 = 24

Sommatoria in “forma currificata”

La funzione `sum` può essere applicata anche soltanto al suo primo argomento:

```
sum double: (int * int) -> int
```

Vogliamo “risparmiare” ancora qualche parentesi e fare in modo che `sum` si possa applicare alla funzione e all’estremo inferiore soltanto?

```
(* sum : (int -> int) -> int -> int -> int *)
let rec sum f lower upper =
  if lower > upper then 0
  else f lower + sum f (lower +1) upper

# sum double;;
- : int -> int -> int = <fun>
# sum double 3;;
- : int -> int = <fun>
```

Funzioni in forma “currificata”

```
(* mult:
     int * int -> int *)
let mult (m,n) = m * n
```

```
(* times:
     int -> int -> int *)
let times m n = m * n
```

times è la **forma currificata** di **mult**: calcola gli stessi valori, ma “consuma un argomento alla volta”

```
let times n = function m -> n*m
let times = function n -> function m -> n*m
```

Funzioni in forma “currificata”

```
(* mult:
     int * int -> int *)
let mult (m,n) = m * n
```

```
(* times:
     int -> int -> int *)
let times m n = m * n
```

times è la **forma currificata** di **mult**: calcola gli stessi valori, ma “consuma un argomento alla volta”

```
let times n = function m -> n*m
let times = function n -> function m -> n*m
```

Può anche essere applicata solo parzialmente:
times 5: int -> int è un’**applicazione parziale** di **times**.

sum (times 5) 1 10 $= \sum_{k=1}^{10}(5 \times k)$

Funzioni in forma “currificata”

```
(* mult:
     int * int -> int *)
let mult (m,n) = m * n
```

```
(* times:
     int -> int -> int *)
let times m n = m * n
```

times è la **forma currificata** di **mult**: calcola gli stessi valori, ma “consuma un argomento alla volta”

```
let times n = function m -> n*m
let times = function n -> function m -> n*m
```

Può anche essere applicata solo parzialmente:
times 5: int -> int è un’**applicazione parziale** di **times**.

sum (times 5) 1 10 $= \sum_{k=1}^{10}(5 \times k)$

```
# List.map (times 2) [1; 2; 3; 4];;
- : int list = [2; 4; 6; 8]
```

Forma currificata di una funzione su tuple

In generale, f_c è la forma currificata di f se

$$\begin{aligned} f: &\quad t_1 \times ... \times t_n \to t \\ f_c: &\quad t_1 \to (t_2 \to ... \to (t_n \to t)...) \end{aligned}$$

e per ogni $a_1, ..., a_n$: $f(a_1, ..., a_n) = (((f_c\, a_1)\, a_2) \ldots a_n)$

Le parentesi possono essere omesse

- sia nel tipo di f_c (si associa a destra),
- sia nell'applicazione di f_c (si associa a sinistra).

Forma currificata di una funzione su tuple

In generale, f_c è la forma currificata di f se

$$\begin{aligned} f: &\quad t_1 \times ... \times t_n \to t \\ f_c: &\quad t_1 \to (t_2 \to ... \to (t_n \to t)...) \end{aligned}$$

e per ogni $a_1, ..., a_n$: $f(a_1, ..., a_n) = (((f_c\, a_1)\, a_2) \ldots a_n)$

Le parentesi possono essere omesse

- sia nel tipo di f_c (si associa a destra),
- sia nell'applicazione di f_c (si associa a sinistra).

Espressioni per denotare funzioni

1. variabili: **times**
2. astrazioni funzionali: **function x -> function y -> x*y**, o anche **fun x y -> x*y**
3. espressioni funzionali: **sum (times 5)**, **sum (times 5) 0**

Funzioni in forma “currificata” (II)

```
# let pair x y = (x,y);;
val pair : 'a -> 'b -> 'a * 'b = <fun>
```

pair 3: `'a -> int * 'a`

è la funzione che “accoppia 3” al suo argomento

```
# let greaterthan x y = y > x;;
val greaterthan : 'a -> 'a -> bool = <fun>
```

greaterthan 0: `int -> int`

è un predicato: essere maggiore di 0

```
# List.filter (greaterthan 0) [2; 0; -1; 4; -8; -10; 5];;
- : int list = [2; 4; 5]
```

Funzioni in forma “currificata” (III)

```
(* sumbetween : int * int -> int *)
(* sumbetween (n,m) = n + (n+1) + ... + m *)
```

Funzioni in forma “currificata” (III)

```
(* sumbetween : int * int -> int *)
(* sumbetween (n,m) = n + (n+1) + ... + m *)
let rec sumbetween (n,m) =
  if n>m then 0 else n + sumbetween (n+1,m)
```

Funzioni in forma “currificata” (III)

```
(* sumbetween : int * int -> int *)
(* sumbetween (n,m) = n + (n+1) + ... + m *)
let rec sumbetween (n,m) =
  if n>m then 0 else n + sumbetween (n+1,m)

(* sbt : int -> int -> int *)
(* sbt n m =  sumbetween (n,m) *)
let rec sbt n m =
  if n>m then 0 else n + sbt (n+1) m
```

sbt “calcola” gli stessi valori di sumbetween, ma “consumando” gli argomenti uno alla volta

```
sumbetween:  int * int -> int
        sbt:  int -> int -> int
```

sbt è la **forma currificata** di sumbetween,
si può applicare parzialmente .
Ad esempio, `sbt 0: int -> int`, applicata a n, riporta la somma dei primi n numeri naturali.

Operazioni predefinite in Ocaml

Molte operazioni predefinite in Ocaml sono in forma currificata

```
# max;;
- : 'a -> 'a -> 'a = <fun>
```

Le operazioni infisse predefinite sono in forma currificata:

```
# (+);;
- : int -> int -> int = <fun>
# (+) 3 5;;
- : int = 8
```

(+) è l'operatore somma usato in forma infissa

```
# ( * );;
- : int -> int -> int = <fun>
# (mod);;
- : int -> int -> int = <fun>
# (=);;
- : 'a -> 'a -> bool = <fun>
```

Definizione di operatori infissi

```
# let (++) x y = 2 * (x+y);;
val ( ++ ) : int -> int -> int = <fun>

# (++) 3 5;;
- : int = 16

# 3 ++ 5;;
- : int = 16

# ++ ;;
Characters 0-2:
  ++ ;;
  ^^
Error: Syntax error

# (++);;
- : int -> int -> int = <fun>
```

Composizione di funzioni

```
(* comp : ('a -> 'b) -> ('c -> 'a) -> ('c -> 'b) *)
(* comp f g = composizione di f con g *)
let comp f g = function x -> f (g x)
```

O anche

```
let comp f g x = f (g x)
```

Che è come dire

```
comp = function f -> function g -> function x -> f (g x)
```

Esempio: composizione di `double` con la funzione predefinita `succ: int -> int` (succ n = n+1) – e viceversa

```
# comp double succ;;
- : int -> int = <fun>
# comp double succ 4;;
```

Composizione di funzioni

```
(* comp : ('a -> 'b) -> ('c -> 'a) -> ('c -> 'b) *)
(* comp f g = composizione di f con g *)
let comp f g = function x -> f (g x)
```

O anche

```
let comp f g x = f (g x)
```

Che è come dire

```
comp = function f -> function g -> function x -> f (g x)
```

Esempio: composizione di `double` con la funzione predefinita `succ: int -> int` (succ n = n+1) – e viceversa

```
# comp double succ;;
- : int -> int = <fun>
# comp double succ 4;;
- : int = 10
# comp succ double 4;;
```

Composizione di funzioni

```
(* comp : ('a -> 'b) -> ('c -> 'a) -> ('c -> 'b) *)
(* comp f g = composizione di f con g *)
let comp f g = function x -> f (g x)
```

O anche

```
let comp f g x = f (g x)
```

Che è come dire

```
comp = function f -> function g -> function x -> f (g x)
```

Esempio: composizione di `double` con la funzione predefinita `succ: int -> int` (succ n = n+1) – e viceversa

```
# comp double succ;;
- : int -> int = <fun>
# comp double succ 4;;
- : int = 10
# comp succ double 4;;
- : int = 9
```

La composizione definita come operatore infisso

```
# let (@@) f g x = f(g x);;
val @@ : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let treble n = 3 * n;;
val treble : int -> int = <fun>

# let sixtimes = double @@ treble;;
val sixtimes : int -> int = <fun>

# let f = (times 2) @@ ((+) 100);;
val f : int -> int = <fun>

# f 3;;
```

La composizione definita come operatore infisso

```
# let (@@) f g x = f(g x);;
val @@ : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>

# let treble n = 3 * n;;
val treble : int -> int = <fun>

# let sixtimes = double @@ treble;;
val sixtimes : int -> int = <fun>

# let f = (times 2) @@ ((+) 100);;
val f : int -> int = <fun>

# f 3;;
- : int = 206
```

Tipi predefiniti: bool

Un tipo è un **insieme di valori**
Tipi semplici:

http://caml.inria.fr/pub/docs/manual-ocaml/core.html#sec473
http://caml.inria.fr/pub/docs/manual-ocaml/libref/Stdlib.html

- bool: {*true*, *false*}

  Operazioni booleane: `not, && , ||`

```
# (true && not false) || false;;
- : bool = true
```

	`E'` è valutato solo se
`E && E'`	`E` ha valore true
`E or E'`	`E` ha valore false

Tipi predefiniti: tipi numerici

- int: $\{0, 1, -1, 2, -2, \ldots, max\_int, min\_int\}$

  Operazioni: `+ - * / mod succ pred`
- float: floating-point numbers (`0.01, 3.0, -4.0, 7E-5,...`)

  Operazioni: `+. -. *. /. ** sqrt sin ...`

  Attenzione: in OCaml non c'è conversione automatica dei tipi numerici

```
(* average : float -> float -> float *)
(* average x y = media aritmetica di x e y *)
let average x y = (x *. y) /. 2.0

# average 2 3;;
```

Tipi predefiniti: tipi numerici

- int: $\{0, 1, -1, 2, -2, \ldots, max\_int, min\_int\}$

  Operazioni: `+ - * / mod succ pred`
- float: floating-point numbers (`0.01, 3.0, -4.0, 7E-5,...`)

  Operazioni: `+. -. *. /. ** sqrt sin ...`

  Attenzione: in OCaml non c'è conversione automatica dei tipi numerici

```
(* average : float -> float -> float *)
(* average x y = media aritmetica di x e y *)
let average x y = (x *. y) /. 2.0

# average 2 3;;
Characters 8-9:
  average 2 3;;
          ^
Error: This expression has type int but an expression
       was expected of type float
```

Tipi predefiniti: char e string

- char: `'a', '9', ' ', ...`

  Operazioni: `int_of_char: char -> int,`
  `char_of_int: int -> char`

```
# int_of_char 'A';;
- : int = 65
# char_of_int 65;;
- : char = 'A'
```

- string: `"pippo", "pluto", "12Ev",...`

  Operazioni: ^ (concatenazione)

```
# "programmazione " ^ "funzionale";;
- : string = "programmazione funzionale"
# "ABCDEFG".[2];;
- : char = 'C'
# string_of_int 45;;
- : string = "45"
```

Operatori di confronto

Uguaglianza, disuguaglianza e operatori di confronto sono **definiti su qualsiasi tipo**, **eccetto le funzioni** e i tipi contenenti funzioni.

```
# 3*8 = 24;;
- : bool = true
# "pippo" = "pi" ^ "ppo";;
- : bool = true
# true = not true;;
- : bool = false
# true <> false;;
- : bool = true
```

```
# 3*8 <= 30;;
- : bool = true
# 6.0 < 5.9;;
- : bool = false
# 'A' >= 'B';;
- : bool = false
# "abc" > "ABC";;
- : bool = true
# false < true;;
- : bool = true
```

```
# (4,true) <= (10,false);;
- : bool = true

# double = times 2;;
Exception: Invalid_argument "equal: functional value"
```

Espressioni condizionali

if E then F else G

è un'espressione condizionale se:

- E è di tipo bool
- F e G hanno **uno** stesso tipo (almeno un sottotipo in comune)

  ML è un linguaggio fortemente tipato: il tipo di "if E then F else G" deve essere determinabile a tempo di compilazione

Le espressioni **hanno sempre un tipo e un valore**:

- Il tipo di "if E then F else G" è il tipo **più generale** che F e G hanno in comune
- Il suo valore è:
  - il valore di F se E ha valore true
  - il valore di G se E ha valore false

È un'espressione, non un costrutto di controllo
La parte else non può mancare

Nel valutare un'espressione "if E then F else G":

- se E è true, G non viene valutata
- se E è false, F non viene valutata

Operatori booleani e espressioni condizionali

Un'espressione della forma	equivale a
E && F	if E then F else false
E or F	if E then true else F

Espressioni condizionali che hanno true/false in uno dei due rami (then/else), si possono riscrivere usando gli operatori booleani.

- se uno dei due rami ha true: qual e' l'operatore logico al quale basta valutare un argomento per riportare true?
- se uno dei due rami ha false: qual e' l'operatore a cui basta valutare un argomento per riportare false?

if E then false else F $\Rightarrow$

Operatori booleani e espressioni condizionali

Un'espressione della forma	equivale a
E && F	if E then F else false
E or F	if E then true else F

Espressioni condizionali che hanno true/false in uno dei due rami (then/else), si possono riscrivere usando gli operatori booleani.

- se uno dei due rami ha true: qual e' l'operatore logico al quale basta valutare un argomento per riportare true?
- se uno dei due rami ha false: qual e' l'operatore a cui basta valutare un argomento per riportare false?

$$\text{if E then false else F} \Rightarrow \text{not E \&\& F}$$

Come nella valutazione delle espressioni booleane, la valutazione delle espressioni condizionali è **"pigra"**:

quando si valuta if E then F else G:

- viene valutata E;
- se il valore di E è true, allora viene valutata F – e G non viene valutata;
- se il valore di E è false, allora viene valutata G – e F non viene valutata.

Esempi

```
# 4 + (if 1 < 0 then 3 * 8 else 5 / 2);;
- : int = 6

4 + (if 1 < 0 then 3 * 8 else 5 / 2)
   ===> 4 + (if false then 3 * 8 else 5 / 2)
   ===> 4 + (5 / 2)
   ===> 4 + 2
   ===> 6

(* sign : int -> int *)
(* sign n = 0, 1 o -1 a seconda del "segno" di n *)
let sign n =
  if n > 0 then 1
  else if n = 0 then 0
       else -1
```

Indentazione: sempre gli else sotto i rispettivi if.

La parte else non manca mai: non ci sono problemi di ambiguità

Regole di calcolo per la valutazione di espressioni

MODELLO DI CALCOLO:
CALCOLARE = RIDURRE

```
# let square x = x * x;;
val square : int -> int = <fun>
     square (3*2) ==> square 6 ==> 6 * 6 ==> 36
     square (3*2) ==> (3*2)*(3*2) ==> 6*(3*2)
                  ==> 6*6 ==> 36
```

Regole di calcolo:

- **CALL BY VALUE**: calcolare il valore dell'argomento prima di applicare una funzione
- **CALL BY NAME**: applicare la funzione prima di aver calcolato il valore dell'argomento

Regola di calcolo di ML: call by value
Eccezioni: espressioni condizionali, operatori booleani

Necessità di espressioni “lazy”

```
let rec fact n = if n=0 then 1
                 else n * fact(n-1)
```

Call by value:

```
fact 1 ==> if 1 <= 0 then 1 else 1 * fact(1-1)
       ==> if false then 1 else 1 * fact 0
       ==> if false then 1
           else 1 * (if 0<=0 then 1 else 0 * fact(0-1))
       ==> if false then 1
           else 1 * (if true then 1 else 0 * fact(-1))
       ==> if false then 1
           else 1 * (if true then 1
                     else 0 * (if -1<=0 then 1
                               else fact(-1-1)))
       ==> if false then 1
           else 1 * (if true then 1
                     else 0 * (if true then 1
                               else fact(-2)))
       ..................
```

Necessità di espressioni “lazy” (II)

Se la valutazione di espressioni condizionali non fosse “pigra”:

```
(* cond : bool * 'a * 'a -> 'a *)
let cond(c,e1,e2) =
  if c then e1 else e2
```

L’applicazione di funzioni definite da programma è regolata dalla “call by value”

```
let rec fact n =
  cond(n=0, 1, n*fact(n-1))

# fact 1;;
Stack overflow during evaluation (looping recursion?).
```

Coppie

(E,F)

```
# let a = (3>4, "pippo");;
val a : bool * string = (false, "pippo")
# let b = (a,5.1);;
val b : (bool * string) * float = ((false, "pippo"), 5.1)
```

$t_1 \times t_2$ è il tipo delle coppie ordinate il cui primo elemento è di tipo t_1 ed il secondo di tipo t_2 (il prodotto cartesiano di t_1 e t_2).

Attenzione: $\times$ è un **costruttore di tipo** (un'operazione su tipi)

```
let c = (double,6);;
```

Coppie

(E,F)

```
# let a = (3>4, "pippo");;
val a : bool * string = (false, "pippo")
# let b = (a,5.1);;
val b : (bool * string) * float = ((false, "pippo"), 5.1)
```

$t_1 \times t_2$ è il tipo delle coppie ordinate il cui primo elemento è di tipo t_1 ed il secondo di tipo t_2 (il prodotto cartesiano di t_1 e t_2).

Attenzione: $\times$ è un **costruttore di tipo** (un'operazione su tipi)

```
let c = (double,6);;
val c : (int -> int) * int = (<fun>, 6)
```

Selettori del tipo coppia: **fst**: $\alpha \times \beta \to \alpha$
snd: $\alpha \times \beta \to \beta$

```
# snd (fst b);;
```

Coppie

(E,F)

```
# let a = (3>4, "pippo");;
val a : bool * string = (false, "pippo")
# let b = (a,5.1);;
val b : (bool * string) * float = ((false, "pippo"), 5.1)
```

$t_1 \times t_2$ è il tipo delle coppie ordinate il cui primo elemento è di tipo t_1 ed il secondo di tipo t_2 (il prodotto cartesiano di t_1 e t_2).

Attenzione: $\times$ è un **costruttore di tipo** (un'operazione su tipi)

```
let c = (double,6);;
val c : (int -> int) * int = (<fun>, 6)
```

Selettori del tipo coppia: **fst**: $\alpha \times \beta \to \alpha$
snd: $\alpha \times \beta \to \beta$

```
# snd (fst b);;
- : string = "pippo"
```

Triple, quadruple ... tuple

```
# (true,5*4,"venti");;
- : bool * int * string = (true, 20, "venti")
# (3<5,10.3,'K',int_of_string "50");;
- : bool * float * char * int = (true, 10.3, 'K', 50)
# (true,("pippo",98),4.0);;
- : bool * (string * int) * float = (true,("pippo",98),4.)
# (3<4,("pippo",90+8)) = (true, "pippo", 98);;
```

Triple, quadruple ... tuple

```
# (true,5*4,"venti");;
- : bool * int * string = (true, 20, "venti")
# (3<5,10.3,'K',int_of_string "50");;
- : bool * float * char * int = (true, 10.3, 'K', 50)
# (true,("pippo",98),4.0);;
- : bool * (string * int) * float = (true,("pippo",98),4.)
# (3<4,("pippo",90+8)) = (true, "pippo", 98);;
Characters 23-42:
    (3<4,("pippo",90+8)) = (true, "pippo", 98);;
                           ^^^^^^^^^^^^^^^^^^^
Error: This expression has type bool * string * int
       but an expression was expected of type
       bool * (string * int)
```

Il prodotto cartesiano non è associativo: bool * (string * int) $\neq$ bool * string * int

Costruttori e Selettori di un tipo di dati

Ogni tipo di dati è caratterizzato da un insieme di

- un insieme di **costruttori** (costanti + operazioni che “costruiscono” valori di quel tipo)
- un insieme di **selettori** (operazioni che “selezionano” componenti da un valore del tipo)

I tipi semplici (int, float, bool, string, char, ...) non hanno selettori ma solo costruttori:

i costruttori di un tipo di dati semplici sono tutti i valori del tipo

Costruttori e Selettori del tipo coppia

- **costruttore**: **(,)** (insieme di parentesi e virgola).
  Applicato a un'espressione di tipo α e una di tipo β, ne costruisce una di tipo $\alpha \times \beta$.
- **selettori**: **fst, snd**.
  Applicate a un'espressione di tipo $\alpha \times \beta$, ne riportano i componenti (di tipo α e β, rispettivamente).

fst e **snd** sono funzioni polimorfe, ma attenzione:

```
# fst (double,6,'p');;
```

Costruttori e Selettori del tipo coppia

- **costruttore**: **(,)** (insieme di parentesi e virgola).
  Applicato a un'espressione di tipo α e una di tipo β, ne costruisce una di tipo $\alpha \times \beta$.
- **selettori**: **fst, snd**.
  Applicate a un'espressione di tipo $\alpha \times \beta$, ne riportano i componenti (di tipo α e β, rispettivamente).

fst e **snd** sono funzioni polimorfe, ma attenzione:

```
# fst (double,6,'p');;
Characters 4-18:  fst (double,6,'p');;
                      ^^^^^^^^^^^^^^
Error: This expression has type (int -> int) * int * char
       but an expression was expected of type 'a * 'b
```