

Applicazione parziale

- Le funzioni curried permettono l'applicazione parziale, cioè gli argomenti possono essere passati uno alla volta in ordine fisso.
- Le funzioni uncurried non permettono l'applicazione parziale: gli argomenti devono essere passati tutti insieme.

Example

```
let curAdd x y = x+y (* curried, int -> int -> int *)
```

```
let uncurAdd (x,y) = x+y (* uncurried, int * int -> int *)
```

uncurAdd(1,2) returns 3, the arguments **must** be passed **together**

curAdd 1 returns a function, the 1-st argument can be passed **without** the 2-nd argument

curAdd 1 2 returns 3, the arguments are passed together

Remark

curAdd 1 2 is equivalent to (curAdd 1) 2

Dichiarazioni

Syntax for declarations (reminder)

```
Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp
Pat ::= ID | '(' ')' | Pat (',' Pat)+
Exp ::= Dec* Exp | 'fun' Pat '->' Exp | ...
```

Promemoria

- **let Pat = Exp** funziona per valori di qualsiasi tipo (incluse le funzioni).
- **let ID Pat+ = Exp** è una comoda abbreviazione per dichiarazioni di funzioni.
- **let f (x,y) z = (z+x, z+y)** è un'abbreviazione per **let f = fun (x, y) -> fun z -> (z + x, z + y)**, dove il parametro z è dichiarato.

Nota Le tuple possono essere utilizzate nei pattern.

Simple examples

```
let x = 2
```

x is a constant of type `int` with value 2

```
let y = x+40
```

y is a constant of type `int` with value 42

the value of `x+y` is 44

```
let inc x = x+1 (* this is actually let inc = fun x -> x+1 *)
```

inc is a constant of type `int -> int`

x is a parameter of type `int`

the value of `inc y` is 43

Esempi più elaborati

```
let x, y = 2, 42
```

- x, y è un pattern di tipo `int * int`.
- x è una costante di tipo `int` con valore 2.
- y è una costante di tipo `int` con valore 42.
- Il valore di `x + y` è 44.

```
let pair = 4, 2
```

- pair è una costante di tipo `int * int` con valore 4, 2.
- `fst` e `snd` sono costanti di tipo `int * int -> int`, predefinite, quindi non necessitano di essere dichiarate.
- `fst` restituisce il primo valore di una coppia.
- `snd` restituisce il secondo valore di una coppia.
- Il valore di `fst pair` è 4.
- Il valore di `snd pair` è 2.

Dichiarazioni nidificate

Le dichiarazioni possono essere nidificate a vari livelli all'interno di altre dichiarazioni:

- **Livello 0:** dichiarazioni di alto livello non contenute in altre dichiarazioni.
- **Livello nidificato n + 1** con $n \geq 0$: se una dichiarazione d è direttamente all'interno di una dichiarazione di livello n, allora d è nidificata a livello n + 1.

"Direttamente all'interno" significa:

- nel corpo di una definizione di funzione
- nella definizione di una costante.

Example of nested declarations

```
let retFun i =  
  let x = 2 * i (* declares x *)  
  fun y -> y + x (* returns a function *)
```

Ambito (scope) di una dichiarazione

Si riferisce alla porzione di programma in cui la dichiarazione è efficace.

Dichiarazioni nidificate e shadowing In alcuni linguaggi, una dichiarazione d1 può sovrascrivere (shadow) una dichiarazione d2 se d1 è all'interno dell'ambito di d2 e dichiara lo stesso nome di d2.

Esempio:

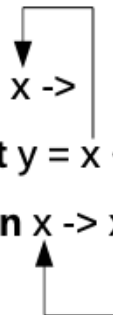
```
let f =  
  fun x ->  
    let y = x + 1  
    fun x -> x * y
```

La dichiarazione del parametro x a livello 2 sovrascrive la dichiarazione del parametro x a livello 1

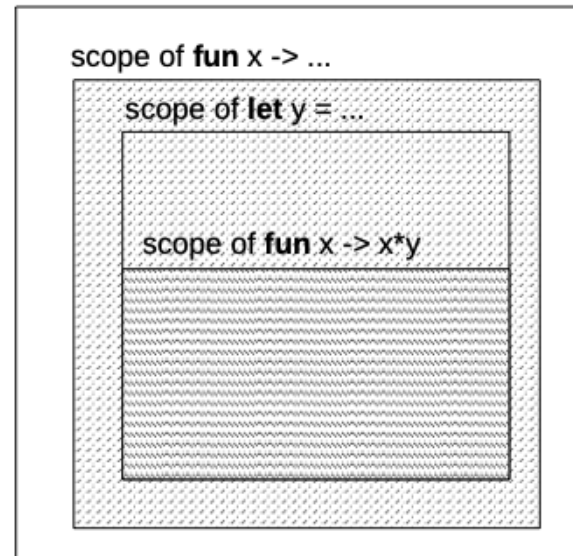
Example of shadowing

the inner declaration of x is in the scope of the outer declaration of x

```
let f =  
  fun x ->  
    let y = x + 1  
    fun x -> x * y
```



scope of **let f = ...**

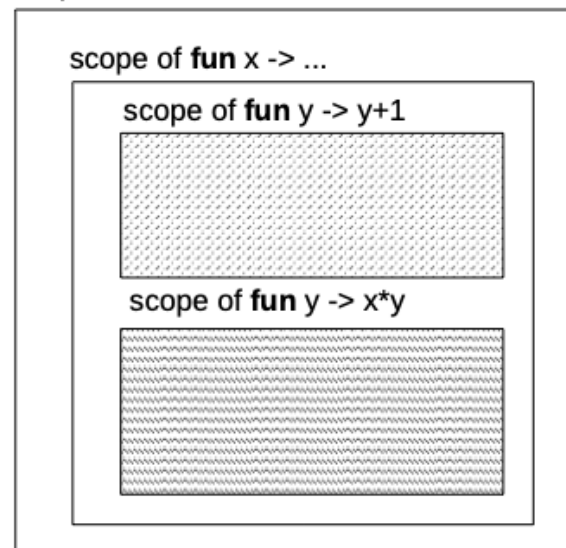


Example of no shadowing

the declarations of the two parameters y have disjoint scopes

```
let f =  
  fun x ->  
    fun y -> y + 1  
    fun y -> x * y
```

scope of **let f = ...**



Valori Booleani

Syntax

```
Exp ::= BOOL | 'not' Exp | Exp '&&' Exp | Exp '||' Exp  
Type ::= 'bool'
```

BOOL è definito dalla espressione regolare: false | true.

Regole sintattiche standard

- Associazione sintattica a sinistra per && e ||.
- not ha una precedenza maggiore di &&.
- && ha una precedenza maggiore di ||.

Semantica statica

- false e true sono corretti staticamente e hanno tipo bool.
- not e è corretto staticamente e ha tipo bool se e solo se e è corretto staticamente e ha tipo bool.
- e1 && e2 e e1 || e2 sono corretti staticamente e hanno tipo bool se e solo se e1 e e2 sono corretti staticamente e hanno tipo bool.

Semantica dinamica

- Gli operandi di && e || vengono valutati da sinistra a destra con **short circuit**.
- **Short circuit** significa che non sempre il secondo operando viene valutato:
 - Se e1 si valuta a false, allora e1 && e2 si valuta a false e e2 non viene valutato.
 - Se e1 si valuta a true, allora e1 && e2 si valuta come il valore di e2.
 - Se e1 si valuta a true, allora e1 || e2 si valuta a true e e2 non viene valutato.
 - Se e1 si valuta a false, allora e1 || e2 si valuta come il valore di e2.

Example

```
1<0 && 0/0>0 returns false
```

```
0/0>0 && 1<0 fails with a dynamic error
```

```
System.DivideByZeroException: Attempted to divide by zero.
```

Dichiarazioni ricorsive

Syntax for recursive declarations

```
Dec ::= 'let' 'rec' Def ('and' Def) *  
Def ::= Pat '=' Exp | ID Pat+ '=' Exp  
Pat ::= ID | '(' Pat ')' | Pat (',' Pat) +
```

Nota

- La parola chiave **'rec'** indica che la dichiarazione è consentita essere ricorsiva.
- L'uso delle parole chiave **'rec'**, **'and'** supporta dichiarazioni mutuamente ricorsive.
- Le dichiarazioni ricorsive sono consentite solo per i tipi di funzione.