Tuple in F#

Syntax

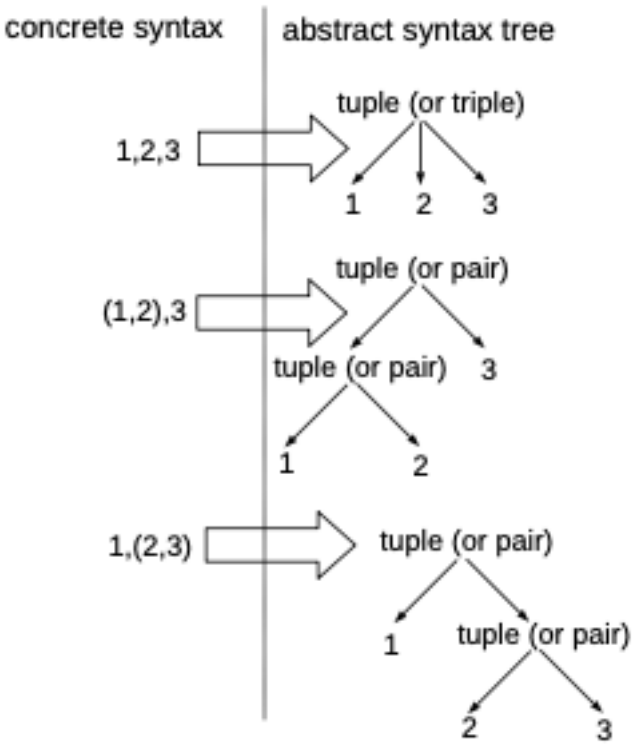
tuple expressions are built with the , (comma) constructor

```
Exp ::= ... | '(' ')' | Exp (',' Exp)+
Pat ::= ID | '(' ')' | Pat (',' Pat)+
```

Regole di precedenza sintattica e di associatività del costruttore di tuple:

- Ha una precedenza maggiore rispetto al costruttore della funzione anonima.
- Ha una precedenza inferiore rispetto agli altri operatori.
- Ha la stessa precedenza degli altri costruttori.
- Non è né associativo a sinistra né a destra.

Esempio: 1, 2, 3 e (1, 2), 3 e 1, (2, 3) hanno alberi di sintassi astratti diversi.



Tipi di tuple in F#

`()` has type `unit` (the `void` type)

`fun` `(x:int) -> ()` has type `int -> unit`

`1,2,3` has type `int * int * int`

`(1,2),3` has type `(int * int) * int`

`1,(2,3)` has type `int * (int * int)`

Precedenza sintattica e regole di associatività per l'operatore

- ***** ha una precedenza maggiore rispetto al costruttore ->.
- *** non è associativo né a sinistra né a destra.**

Esempi

Int * int * int
(int * int) * int
int*(int*int)
hanno alberi di sintassi astratti differenti e quindi sono tipi diversi.

Funzioni curried e uncurried

Gli argomenti multipli possono essere gestiti in due modi differenti:

- **Funzione curried:** una funzione di ordine superiore che restituisce una "catena" di funzioni.

  `fun` pat_1 `->` `fun` pat_2 `->` `...` `fun` pat_n `->` exp

  or with the abbreviated notation

  `fun` pat_1 pat_2 `...` pat_n `->` exp

- **Funzione uncurried:** una funzione che prende come argomento una tupla di grandezza n.

  `fun` $(pat_1, pat_2, \ldots, pat_n)$ `->` exp

  Corrispondenza tra funzioni curried e uncurried
  Una funzione uncurried può essere trasformata nella corrispondente versione curried e viceversa.

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`