Concetti Base

Antenati di OCaml

ML (Meta Language)
Linguaggio funzionale fortemente tipato con avanzate funzionalità di “type inferece”, prevede funzionalità caratteristiche della programmazione funzionale:

funzioni come valori, variabili immutabili, polimorfismo, pattern matching

Caml (Categorical Abstract Machine Language)

Versione di ML dotata di una implementazione particolarmente efficiente, introduce la possibilità di definire moduli e fornisce una serie di librerie di programmazione

OCaml (Objective Caml)

Estende Caml con funzionalità di programmazione orientata agli oggetti

Struttura di una dichiarazione

```
let x = 10 ;;
```

```
val x : int = 10
```

`let` crea un binding, ossia un legame, tra un nome e un valore. Caratteristica fondamentale della programmazione funzionale è che anche le funzioni sono valori.
Quindi let potrà essere usato anche per dichiarare funzioni.

Dopo il simbolo `:` abbiamo il tipo dell’espressione che OCaml ha inferito

```
x + 5 ;;
```

```
- : int = 15
```

Il terminatore `;;` informa l’interprete che l’espressione è finita

Formato delle variabili

I nomi (o identificatori) delle variabili dichiarate tramite il costrutto let:

- possono contenere `lettere`, `numeri` e i `caratteri` `_` e `'`
- devono iniziare con una lettera minuscola o con `_`

```
let x = 3 + 4 ;;
```

```
val x : int = 7
```

Esempio errato:

```
let Seven = 3 + 4 ;;
```

```
File "[6]", line 1, characters 4-9:
1 | let Seven = 3 + 4 ;;
        ^^^^^
Error: Unbound constructor Seven
```

📎 In OCaml tutto è un espressione e **non** un comando

Tipi di base

- `Int`: Numeri interi
- `float`: Numeri frazionari
- `bool`: Valori booleani
- `char`: Singoli caratteri (racchiusi tra apici)
- `string`: Stringhe (racchiuse tra virgolette)
- `unit`: tipo simile al *void* di altri linguaggi

Operazioni su `Int`

```
3 + 5 ;; (* operazione su interi: tipo int *)
```

```
- : int = 8
```

Operazioni su `float`

```
3.2 +. 5.2 ;; (* operazione su frazionari: tipo float *)
```

```
- : float = 8.4
```

☝🏼 Il tipo degli operandi deve essere coerente con il tipo dell'operazione

Conversione di tipo

Poiché OCaml non prevedere conversione di tipo automatiche (3 al posto di 3.0) , allora per le conversioni devono essere esplicitate usando `float_of_int` e `int_of_float`

```
float_of_int 3 ;;
int_of_float 3.0 ;;
```

```
- : float = 3.0
- : int = 3
```

```
(float_of_int 3) +. 2.4 ;;
```

```
- : float = 5.4
```

Operazioni su `bool`

```
let x = true;;
```

```
val x : bool = true
```

```
x && false ;;
x || false ;;

not x;; (* not è l'operazione di negazione -- come ! in molti altri linguag
```

```
- : bool = false

- : bool = false

- : bool = true
```

```
1 = 2 ;; (* ATTENZIONE, il confronto si fa con =, non con == ! *)
1 <> 2 ;; (* mentre per vedere se due espressioni sono diverse si usa <>, n
```

```
- : bool = false

- : bool = true
```

Operazioni su `char` e `string`

```
let c = 'a' ;;
int_of_char c ;; (* conversione esplicita, esiste anche char_of_int *)
```

```
val c : char = 'a'

- : int = 97
```

```
let h = "hello" ;;
let hw = h ^ "world" ;; (* concatenazione con simbolo ^ *)
```

```
val h : string = "hello"

val hw : string = "helloworld"
```

```
int_of_string "10";; (* conversione esplicita, esistono anche per float, bo
```

```
- : int = 10
```

I tipi tupla

Una tupla (o ennupla) è una sequenza di valori separati da virgole.

```
let t = (10,"hello",12.5) ;;
```

```
val t : int * string * float = (10, "hello", 12.5)
```

Il tipo di una tupla è il prodotto dei tipi degli elementi che la compongono

Funzioni

In coerenza con il paradigma funzionale, in OCaml le funzioni sono valori:

- possiamo ottenere una funzione come risultato della valutazione di un'espressione
- possiamo passare una funzione come parametro ad un'altra funzione
- una funzione può essere restituita come risultato di un'altra funzione

λ-calcolo e OCaml

Il modo "base" di definire una funzione riprende quanto visto nel l-calcolo, ad esempio la funzione `λx.x + 1` può essere definita in OCaml come segue:

```
fun x -> x + 1 ;;
```

```
- : int -> int = <fun>
```

e la sua applicazione `(λx.x + 1)10` diventa:

```
(fun x -> x + 1) 10 ;;
```

```
- : int = 11
```

Ancora un esempio:

$$(\lambda f.\lambda n.fn+1)(\lambda x.2x)10$$

```
(fun f -> fun n -> f n + 1)(fun x -> 2 * x) 10 ;;
```

Questo è un esempio di funzione higher-order (di ordine superiore), ossia una funzione che prende un'altra funzione (f) come parametro e la usa nel proprio corpo.

> il valore calcolato da OCaml è lo stesso che si ottiene valutando l'espressione del l-calcolo tramite la β-riduzione:

Richiamiamo le regole della β-riduzione:

$$(\lambda x.e1)e2 \to e1[x/e2]$$

$$\frac{e_1 \to e'}{e_1 e_2 \to e' e_2}$$

$$\frac{e_2 \to e'}{e_1 e_2 \to e_1 e'}$$

$$\frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

Ne segue che l'espressione $(\lambda f.\lambda n.fn+1)(\lambda x.2x)10$ abbia la seguente semantica:

(λ f .λn. f n + 1)(λx.2x)10 *ricordando che* e1 e2 e3 = (e1 e2)e3
→ (λn.(λx.2x)n + 1)10
→ (λn.2n + 1)10
→ 20 + 1 = 21

Funzioni con più parametri

Il costrutto fun può prevedere più di un parametro

```
(fun x y -> x+y) 3 4 ;; (* equivale a ( fun x -> fun y -> x+y ) 3 4 *)
```

```
- int = 7
```

In generale, con il termine zucchero sintattico (***o syntactic sugar***) si intende dire che un certo costrutto è stato previsto in un linguaggio di programmazione solo per semplificare la scrittura di certe operazioni.

Le stesse operazioni possono essere scritte usando anche altri costrutti del linguaggio ottenendo comunque lo stesso identico risultato.

λ-calcolo in OCaml (sintatticamente)

Dato un qualunque termine del λ-calcolo T, la corrispondente espressione in OCaml è $[[T]]$, dove $[[...]]$ è la funzione definita ricorsivamente sulla struttura dei λ-termini come segue:

$$[\![n]\!] = n$$

$$[\![x]\!] = x$$

$$[\![T' op T'']\!] = [\![T']\!] op [\![T'']\!]$$

$$[\![\lambda x.T']\!] = fun x -> [\![T]\!]$$

$$[\![T'T'']\!] = [\![T']\!][\![T'']\!]$$

Dove n è un qualunque numero intero, x è un qualunque identificatore (nome) e op è un operatore artimetico ($+, -, *, /, ...$)

📎 La funzione $[\![...]\!]$ ha come dominio l'insieme delle espressioni (o termini) del λ-calcolo e come codominio le espressioni OCaml

In realtà $[\![...]\!]$è una funzione molto semplice, che traduce letteralmente l'espressione del

λ-calcolo trasformando i vari $\lambda x.$ in `fun x ->` e lasciando tutto il resto sostanzialmente inalterato.

esempi:

La seguente λ-espressione

$$[\![\lambda x.\lambda y.x + y]\!]$$

può essere tradotta passo passo in OCaml:

`fun x ->` $[\![\lambda y.x + y]\!]$

`fun x -> fun y ->` $[\![x + y]\!]$

`fun x -> fun y ->` $[\![x]\!] + [\![y]\!]$

`fun x -> fun y -> x +` $[\![y]\!]$

`fun x -> fun y -> x + y`

Definire funzioni tramite `let`

Le funzioni essendo valori possono essere usate in una dichiarazione tramite `let`

```
let somma = fun x y -> x+y ;;
somma 3 4;;
```

```
val somma : int -> int -> int = <fun>
- : int = 7
```

Utilizzando lo ***zucchero sintattico*** si può evitare di scrivere `fun` e `->`

```
let somma = x y = x + y ;;
somma 3 4;;
```

```
val somma : int -> int -> int = <fun>
- : int = 7
```

☝🏻 L'utilizzo dello zucchero sintattico nella dichiarazione di funzioni è molto comune

Si può anche utilizzare "`function`" per definire una funzione ma in questo caso si può prevedere un unico parametro

```
function x -> x + 1 ;;
```

```
- : int -> int = <fun>
```

Siccome il corpo di una funzione in OCaml è un espressione non esiste il "`return`"

```
let media x y =
    x +. y /. 2.0
```

```
val media : float -> float -> float = <fun>
```

Il comando return ha senso nella *programmazione imperativa*, in cui il corpo di una funzione è una sequenza di comandi, ad esempio in javascript:

```
function area_triangolo ( x , y ) {
    let somma = x + y;
    return somma / 2;
}
```

Scope

Una dichiarazione di variabile aggiorna l'ambiente globale gestito dal toplevel di OCaml e diventa utilizzabile da tutte le espressioni successive

```
let x = 10 ;;
let y = 5 * x ;;
```

```
val x : int = 10
val y : int = 50
```

costrutto `let . . . in`

Questo costrutto dichiara variabili il cui scope è una sola espressione

```
let z = 10 in z * z / 2 ;; (* z è utilizzabile solo qui... *)
let w = 5 * z ;; (* mentre qui no! *)
```

```
File "[33]", line 2, characters 12-13:
2 | let w = 5 * z ;; (* mentre qui no! *)
                                ^
Error: Unbound value z
```

Le dichiarazioni di variabili con scope limitato (variabili locali) è ottenuto in molti altri linguaggi (come JavaScript, C, Java) creando blocchi di comandi `{ ... }` e dichiarando la variabile all'interno del blocco

Funzioni *curryed*

Il nome della funzione e i vari parametri sono separati da spazi, senza parentesi e virgole (diversamente da molti altri linguaggi)

```
let somma x y = x+y ;;
somma 3 4 ;;
```

```
val somma : int -> int -> int = <fun>
- : int = 7
```

questa tipologia di funzione permette di non usare parentesi e virgole per i parametri, ma ciò non vieta però di utilizzarle, ad esempio:

```
let somma (x, y) = x+y ;;
somma (3,4);;
```

```
val somma : int -> int -> int = <fun>
- : int = 7
```

ma in questo caso stiamo scrivendo una funzione con un unico parametro di tipo int * int ossia una coppia.

La notazione *Curryed* (cioè senza virgole) è quella comunemente usata per le funzioni in OCaml Anche la notazione
Curryed è in realtà un richiamo al λ-calcolo, in cui l'applicazione di funzione si

scrive f x senza parentesi ne virgole.

Applicazione parziale di funzione

La notazione Curryed consente di applicare una funzione parzialmente, passandole solo una parte dei parametri

```
let somma x y = x+y ;;
somma 10;;
```

```
val somma : int -> int -> int = <fun>
- : int -> int = <fun>
```

Questo meccanismo non va confuso con la possibilità (presente in molti linguaggi) di definire funzioni con parametri opzionali, o funzioni con parametri a cui è associato un valore di default.

In questi casi, infatti, la funzione che applichiamo è sempre la stessa, ma avremo semplicemente la possibilità di evitare di scrivere qualche parametro il cui valore non è particolarmente rilevante.

Controllo del flusso di esecuzione

Siccome OCaml è un linguaggio funzione e i costrutti `if` e `while` sono controlli del flusso di linguaggi imperativi, possiamo utilizzare:

- una espressione di scelta condizonale (`if`)
- la ricorsione

Espressione di scelta condizionale if

In OCaml `if` è un'espressione, e deve sempre prevedere due rami: `then` e `else`

- I due rami dovranno contenere a loro volta espressioni dello stesso tipo
- L'if potrà essere usato "a destra" di una dichiarazione

```
let x = 23 ;;
let y = if x>10 then 1 else 0 ;;
```

```
val x : int = 23
val y : int = 1
```

💡 L'if in OCaml assomiglia all'operatore di scelta condizionale `... ? ... : ...`

Ricorsione

Per definire funzioni ricorsive è necessario utilizzare il costrutto `let rec` altrimenti OCaml restituisce un errore, ad esempio:

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

```
File "[48]", line 2, characters 28-32:
2 | if n<=0 then 1 else n * fact (n-1) ;;
                                ^^^^
Error: Unbound value fact
Hint: If this is a recursive definition,
you should add the 'rec' keyword on line 1
```

si può evitare ciò tramite il costrutto dedicato alla ricorsione:

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

```
val fact : int -> int = <fun>
```

Esempio di funzione ricorsiva:

```
let rec fibonacci n =
    if n=0 || n=1 then n
    else fibonacci (n-1) + fibonacci (n-2);;
```

```
val fibonacci : int -> int = <fun>
```

Funzioni mutuamente ricorsive

Ogni funzione può richiamare solo funzioni definte precedentemente.
Funzioni mutuamente ricorsive vanno definite una dopo l'altra separate da `and`

```
let rec pari n =
    match n with
```

```
     | 0 -> true
     | x -> dispari (x-1)

and dispari n =
    match n with
     | 0 -> false
     | x -> pari (x-1) ;;
```

```
val pari : int -> bool = <fun>
val dispari : int -> bool = <fun>
```

```
pari 6;;
dispari 6;;
```

```
- : bool = true
- : bool = true
```

Type Inference

In OCaml il tipo di una espressione viene inferito analizzando:

- i valori inseriti
- gli operatori usati

I valori (letterali) presenti hanno un tipo ben definito, e anche gli operatori hanno dei tipi specifici a cui possono essere applicati.
A partire da queste certezze, il compilatore di OCaml può dedurre il tipo delle variabili utilizzate nell'espressione. Ad esempio, in
`x+y` le variabili `x` e `y` devono necessariamente avere tipo `int` perché sono operandi di un `+`

Esaminando l'espressione partendo dai singoli valori e dalle singole variabili e facendo dei passi di deduzione che riguardano via via sottoespressioni sempre più ampie (che corrisponde a "risalire" l'albero di sintassi astratta dell'espressione) il compilatore procede creandosi un elenco di associazioni "*variabile → tipo*" che si porta dietro durante l'analisi dell'espressione.

Type Inference e funzioni

Nel caso di espressioni che definiscano (o utilizzino) funzioni, inferire il tipo è un po' più complicato, anche se l'approccio rimane lo stesso

Si analizzano:

- i valori inseriti
- gli operatori usati
- gli argomenti/valori di ritorno delle funzioni

```
let f x = x+1 ;;
```

```
val f : int -> int = <fun>
```

```
let sum_if_true test first second =
    (if test first then first else 0)
    + (if test second then second else 0);;
```

Questa è una funzione higher order che prende una funzione *test* che effettua un controllo sui due parametri first e second e calcola una somma.
I parametri first e second prendono parte alla somma solo se superano il controllo operato dalla funzione test (ossia solo se il risultato di test è true) altrimenti vengono sostituiti da 0.

```
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

Il tipo inferito è `(int -> bool) -> int -> int -> int` poiché:

- i due rami dell'if devono avere lo stesso tipo (quindi first e second sono int come 0)
- essendo usata nella guardia di un `if` la funzione test deve restituire un `bool`
- il risultato di `+` è sempre di tipo `int`

Vediamo qualche esempio di uso di `sum_if_true`:

```
sum_if_true (fun x -> x mod 2=0) 3 4;; (* somma se pari -- mod calcola il m
```

```
- : int = 4
```

```
sum_if_true (fun x -> x>0) 3 (-4);; (* somma se positivo *)
```

```
- : int = 3
```

```
let rec potenza_di_due x =
    if x = 1 then true
    else if x < 1 || x mod 2 = 1 then false
        else potenza_di_due (x/2);;

sum_if_true potenza_di_due 3 4;; (* somma se potenza di due *)
```

```
val potenza_di_due : int -> bool = <fun>
- : int = 4
```

L'ultimo esempio, potenza_di_due, mostra innazitutto che le funzioni passate come parametro non deve necessariamente essere anonime, ma possono essere qualunque funzione definibile in OCaml.

Type Inference e polimorfismo

In certi casi il tipo di una funzione non può essere identificato in modo concreto (ossia facendo riferimento ai tipi di base
`int`, `float`, ...)

```
let id x = x ;; (* funzione identità *)
```

```
val id : 'a -> 'a = <fun>
```

la funzione id è polimorfa: si può applicare ad argomenti di tipi diversi

- il risultato avrà per certo lo stesso tipo dell'argomento passato (qualunque sia)

Il costrutto 'a è una variabile di tipo:

- indica che possono essere usati valori di qualunque tipo
- se presente più volte, i valori corrispondenti dovranno avere tutti lo stesso tipo

```
id 4 ;;
```

```
- : int = 4
```

```
id 'a' ;;
```

```
- : char = 'a'
```

```
id "ciao" ;;
```

```
- : string = "ciao"
```

Un altro esempio di funzione polimorfa:

funzione maggiore

```
let maggiore x y =
    if x > y then x else y ;;
```

```
 val maggiore : 'a -> 'a -> 'a = <fun>
```

```
maggiore 6 5 ;;
```

```
- : int = 6
```

```
maggiore 3.5 9.4
```

```
- : float = 9.4
```

```
maggiore "albero" "gatto";;
```

```
- : string = "gatto"
```

```
maggiore true false ;;
```

```
- : bool = true
```

Se ad esempio proviamo ad applicare la funzione *maggiore* a due funzioni, si propaga un errore:

```
maggiore (fun x -> x) (fun x -> x+1) ;;
```

```
Exception: Invalid_argument "compare: functional value".
Raised by primitive operation at file "[66]", line 2, characters 7-12
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Annotazioni di tipo

Tramite questo metodo è possibile restringere o eliminare il polimorfismo

```
let maggiore (x : int) (y : int) : int =
    if x > y then x else y ;;
```

```
val maggiore : int -> int -> int = <fun>
```

Grazie a ciò ci si può assicurare che non vengano confrontate due funzioni

Differenze tra OCaml e Typescript

Il linguaggio TypeScript ha un sistema di type inference con tipi polimorfi simile a quello di OCaml ma esistono alcune differenze importanti, ad esempio:

Tipi di base

In OCaml:

- `int`
- `float`
- `bool`
- `...`

In Typescript:

- `number`
- `boolean`
- `any`
- `unknown`
- `...`

In linea generale però OCaml necessita di inferire con precisione il tipo di ogni espressione, invece TypeScript tende a essere più permissivo (vedi tipi `any` e `unknown`)

Ad esempio, la funzione identità

```
let id x = x ;;
```

scritta in OCaml, in Typescript può essere scritta utilizzando `any`

```
function id (x: any) : any {
    return x
}
```

Ma in questo caso, utilizzando `any` il compilatore accetterà tutto, quindi sarà possibile effettuare anche operazioni del tipo: `12 * id("hello")`

In generale, agli occhi del compilatore di TypeScript la funzione id
prende un parametro di qualunque tipo e restituisce un risultato di qualunque tipo (anche diverso dal precedente).

In OCaml, invece, dal momento che il tipo inferito è `'a -> 'a`, il compilatore è consapevole che qualunque sia il tipo del parametro passato alla funzione, il risultato che si otterrà sarà dello stesso tipo.

In modo migliore, in Typescript si possono utilizzare i ***generics***:

```
function id<T> (x: T) : T {
    return x
}
```