Funzioni di prima classe e chiusure

In un linguaggio funzionale tendenzialmente le funzioni sono **entità di prima classe**.
Un'entità di prima classe può essere:

- **Argomento** di una funzione.
- **Risultato** di una funzione.
- **Definita** dentro una funzione.
- **Assegnata** a una variabile.
- **Memorizzata** in una struttura dati (array, liste, alberi, ...).

In generale, un'entità di prima classe in un *linguaggio tipato* ha un **tipo che la descrive**, andando a descrivere il tipo del dominio (*input*) e del codominio (*output*).

Vengono ora osservati diversi linguaggi, per determinare se sono linguaggi funzionali o meno:

- **C**: il linguaggio C **non** è un linguaggio funzionale.

  In C esistono dei **puntatori a funzioni**, ma non è possibile definire funzioni dentro funzioni.

- **Pascal**: anche il linguaggio Pascal **non** è un linguaggio funzionale.

  In Pascal è possibile definire **funzioni dentro funzioni**, ma non c'è un tipo `funzione`.

- **Haskell**: **è** un linguaggio funzionale.

  Le funzioni hanno un tipo, possono essere definite dentro ad altre funzioni e possono essere risultati di altre funzioni.

- **Java, Python, C++, ...**: non sono pienamente linguaggi funzionali.

  Le funzioni vengono modellate come oggetti (e incorporate a posteriori). Viene applicato dello zucchero sintattico per le λ espressioni (ad esempio, in Java: `(a, b) -> ...;`). Inoltre, vengono utilizzate delle interfacce funzionali, come il `Comparator` in Java.

Un semplice test per comprendere se all'interno di un linguaggio le funzioni sono entità di prima classe è il **definire la composizione funzionale** $(f \circ g)(x) = f(g(x))$.

Viene scelto questo test dato che la composizione funzionale prende in input due funzioni e ne restituisce una in output.

Vediamo ora due esempi di implementazione.
In Haskell, preso dalla libreria standard:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g = \x -> f (g x)
```

Nota: `f (g x)` è equivalente al solito `f(g(x))`. Viene scelta questa notazione per semplificare l'utilizzo del linguaggio, evitando l'uso eccessivo di parentesi.
Inoltre, la notazione `\x` indica una nuova funzione che si aspetta in input l'argomento `x`.
Il \ vuole richiamare ad una λ stilizzata.

In Java:

```
public static <A,B,C>
Function<A,C> compose(Function<B,C> f, Function<A,B> g) {
    return x -> f.apply(g.apply(x));
}
```

Il metodo si aspetta due funzioni che andranno a produrre una funzione come output. Tutte queste funzioni sono in realtà oggetti che implementano una determinata interfaccia (`Function`).

Proviamo invece ora ad implementarlo in C:

```
int (*)(int) compose(int (*f)(int), int (*g)(int)) {
    int aux(int x) {
        return f(g(x));
    }
    return aux;
}

int main() {
    int (*plus2)(int) = compose(succ, succ);
    printf("%d", plus2(1));
}
```

Ci sono però dei problemi. Stiamo richiamando valori al di fuori della funzione `aux` (`f` e `g` che appartengono a `compose`).

Quando viene eseguito il corpo di `aux`, gli slot che contengono i valori di `f` e `g` non esistono più. Una funzione è una *computazione ritardata* che può accedere a nomi definiti all'esterno del suo corpo, ma che potrebbero non esistere più nel momento in cui il corpo viene eseguito.

Proviamo ora ad implementarlo anche in Pascal:

```
function E(x: real): real;
    function F(y: real): real;
    begin
        F := x + y
    end;
begin
    E := F(3) + F(4)
end;
```

In C le funzioni non possono essere annidate, in Pascal si. Se una funzione accede a nomi definiti all'esterno del suo corpo, deve trattarsi di variabili **globali** (o di altre funzioni **globali**). Variabili/funzioni globali esistono per tutta la durata del programma, quindi siamo salvi.

In Pascal le funzioni possono essere annidate, ma non restituite come risultato. Se una funzione accede a nomi definiti all'esterno del suo corpo, deve trattarsi di variabili **globali** o di **variabili locali ancora esistenti**.

Viene quindi utilizzato un meccanismo chiamato **puntatore di catena statica**. Il puntatore di catena statica è il collegamento che ogni activation record (di una funzione annidata) mantiene con il proprio ambiente di definizione. Questo collegamento è fondamentale per permettere alla funzione annidata di accedere correttamente alle variabili definite nei livelli esterni. È una soluzione elegante per gestire la visibilità e la durata delle variabili in presenza di funzioni annidate, garantendo che le variabili locali di un contesto esterno siano ancora accessibili fintanto che ne esiste un riferimento valido.

Chiusure

In un linguaggio funzionale una funzione può essere invocata in un luogo e in un tempo molto diversi da quelli in cui la funzione è stata definita.

Per assicurare che l'esecuzione del corpo della funzione proceda senza intoppi (ovvero che tutti i dati di cui ha bisogno esistono ancora) tali dati vengono copiati e "impacchettati" insieme al codice della funzione in una cosiddetta **chiusura**.

Chiusura = **codice della funzione** + **(valori delle) variabili libere**.

Le chiusure possono avvere molti utilizzi. Un possibile caso d'uso è il **currying**.

In molti linguaggi funzionali, le funzioni "a più argomenti" sono "cascate" di funzioni a un singolo argomento. Due possibili esempi possono essere:

```
add :: Int -> Int -> Int
add = \x -> \y -> x + y
```

```
leq :: Int -> Int -> Bool
leq = \x -> \y -> x <= y
```

Questo consente di non avere una nozione nativa di "funzione a più argomenti", rendendo quindi il linguaggio più semplice.

Inoltre, le funzioni "currificate" possono essere **specializzate**. Un possibile esempio dove si specializza una funzione è il seguente, riprendendo la funzione `add` definita precedentemente:

```
>let f = add 1
>f 5
6
>f 7
8
```

Si può notare quindi come la funzione `add` sia stata specializzata come una funzione che somma `1` al numero passato come argomento.

Un ulteriore utilizzo delle chiusure è quello di **incapsulare** un dato sensibile creato localmente da un'altra funzione.

Un possibile esempio di quest'applicazione può essere, in linguaggio OCaml:

```
let make_counter () =
    let c = ref 0 in
    (fun () -> !c),
    (fun () -> c := !c + 1),
    (fun () -> c := 0)

let get, inc, reset = make_counter ()
```

In questo caso, `make_counter` crea un riferimento `c` e tre funzioni per leggere, incrementare e resettare il contenuto di `c`.

Il contatore è incapsulato nella chiusura di (ed utilizzabile solo attraverso) queste funzioni.

Altro caso d'uso comune per le chiusure è il loro utilizzo per **impacchettare** espressioni che non devono essere valutate subito, ma solo se necessario.

Questo concetto è allineato con i linguaggi **lazy**, come Haskell, dove gli argomenti di una funzione vengono valutati al massimo una volta e solo se necessario.

Un possibile esempio di quest'applicazione può essere:

```
zif f []       _        = []
zif f _        []       = []
zif f (x : xs) (y : ys) = f x y : zif f xs ys

sorted xs = and (zif (<=) xs (tail xs))
```

La funzione `zif` permette di controllare se una lista è ordinata. Un modo elegante di effettuare questo controllo è quello di confrontare le coppie che si ottengono unendo un valore con il suo successivo all'interno della lista.

`(zif (<=) xs (tail xs))` permette quindi di confrontare la lista con la sua coda (la lista senza il primo elemento).

`(tail xs)` potrebbe però provocare dei problemi, non essendo definita in caso di `xs` vuota.

Essendo però Haskell un linguaggio lazy, `sorted` è corretta perché quando `xs` è vuota il secondo argomento di `zif` non viene valutato.

Questa logica in Haskell viene anche applicata in contesti di base, come l'implementazione degli operatori logici (come l'operatore `AND &&`) e dell'if-else. Non vengono quindi valutati gli argomenti non necessari.

Chiusure in Java

Altri linguaggi moderni hanno implementato costrutti della programmazione funzionale all'interno del proprio linguaggio. Un esempio sono le chiusure implementate in Java.

Un possibile esempio di chiusura in Java può essere:

```
public static Function<Integer,Integer> add(int x) {
    return y -> x + y;
}
```

In questo caso, la variabile `x` è **effectively final**. Il valore di `x` non deve essere alterato all'interno del codice della chiusura.

La chiusura contiene **copie** dei valori delle variabili libere. Se tali variabili sono modificate, la semantica della chiusura può risultare oscura.
Se `x` fosse un riferimento ad un oggetto la situazione si andrebbe a complicare, potendo modificare lo stato interno dell'oggetto successivamente.

In un linguaggio funzionale **puro** il problema non si pone perché non c'è assegnamento.

Volendo quindi confrontare *oggetti* e *chiusure*:

- All'interno di oggetti sono presenti **(campi + riferimenti a metodi)**, mentre nelle chiusure sono presenti **(valori della variabili libere + riferimento a funzione)**.
- Negli oggetti i **campi** sono **mutabili**, mentre nelle chiusure i **valori** sono **immutabili**.
- Negli oggetti i metodi hanno un parametro implicito (tipicamente `self` o `this`) con riferimento all'oggetto ricevente, mentre nelle chiusure la funzione ha un parametro implicito con riferimento alla chiusura in cui la funzione trova i valori delle variabili libere.