**Emerging Programming Paradigms**

[ lezioni 06 - 08 - 15 aprile ]

**Futures e promises**

Un future non è un valore, bensì un tipo di dato astratto che **rappresenta** (o descrive) un valore. Questo concetto sarà ripreso dalle monadi in Haskell. Un future viene creato descrivendo la computazione che ne determinerà il valore: x = make\_future(fun() → 100!). yield() prende in input un future e ritorna il risultato della computazione.

Erano stati introdotti negli anni 70 ma non hanno mai preso piede nei linguaggi mainstream. Nei linguaggi logici le variabili sono sempre dei future. Ora stanno di nuovo tornando emergenti per l’utilizzo di interfacce web responsive.

Il valore di un future viene calcolato al più una volta (**memoization**), tipicamente il risultato viene memorizzato in una tabella hash e riutilizzato nelle chiamate future.

Gli eventuali **side-effects** (stampe, eccezioni, ecc) vengono memorizzati anch’essi e risollevati ogni volta che il valore del future viene letto.

La **computazione** può avvenire in qualunque momento tra la creazione del future e la sua lettura.

I linguaggi di programmazione possono specificare particolari **semantiche operazionali**, ovvero come vengono calcolati i futures. Le due più diffuse sono:

- La computazione viene immediatamente lanciata alla creazione del future in un altro thread/processo o su un altro nodo, la lettura è bloccante fino a quando la computazione non è terminata.

- La computazione viene lanciata nel thread corrente, ma non quando il future viene creato, bensì la prima volta che viene chiesta la lettura. Un future con questa semantica viene chiamato **thunk**. Questo approccio permette computazioni lazy in linguaggi eager (linguaggi in cui la valutazione di un'espressione avviene non appena essa viene legata a una variabile).

Un esempio di utilizzo di future in Erlang: [inserire link a codice]

Una **promise** altro non è che un future a cui però non viene specificata la funzione da eseguire all’inizio, bensì separatamente tramite la funzione set().

In JavaScript il metodo **then** di un future crea un nuovo future che inizia facendo **get** del primo. Nelle Promises di Javascript non c’è nessun get e il codice non esce mai dal mondo asincrono.

Un'altra bella feature dei future è il **pipelining.** Si possono definire dipendenze tra i future, i future indipendenti sono eseguiti in parallelo.

Il future pipelining può *ridurre il traffico di rete* perché risultati intermedi non vengono trasferiti ma solo quello finale.

**Chiusure**

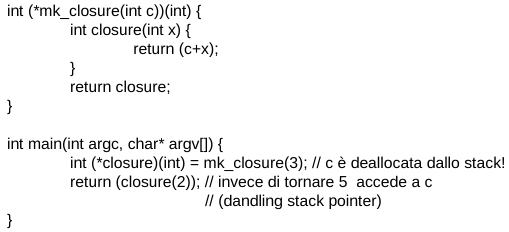
Un “oggetto” è di **prima classe** se può essere usato come un dato qualunque, ovvero preso in input e dato in output da una funzione, memorizzato in variabili, ecc.

Una funzione al top-level, non è definita all'interno di altre funzioni. Le variabili in scope di una funzione sono le variabili locali e globali “viste” dalla funzione. Le variabili locali sono memorizzate nello stack e “sopravvivono” fin tanto che la funzione è in esecuzione. Le variabili globali invece sono memorizzate nello heap e rimangono attive per tutta l’esecuzione.

Un **puntatore ad una funzione** altro non è che l’indirizzo in cui risiede il codice della funzione. Questi in C sono oggetti di prima classe e vengono utilizzati per risolvere vari problemi tra cui il polimorfismo. Il C ammette solo funzioni al top-level.

Una **chiusura** è una funzione di prima classe definita all’interno di uno o più blocchi, ad esempio all’interno del corpo di un’altra funzione. In questo modo la funzione può accedere sia alle proprie variabili locali che a quelle dei blocchi che la contengono (ricorsivamente). Il C non permette l’utilizzo di chiusure.

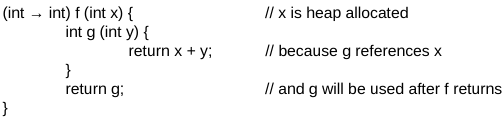
Il motivo per cui il C non permette le chiusure è perché se si allocassero le variabili locali sullo stack e si avessero funzioni annidate allora a runtime si avrebbero problemi. Infatti se le chiusure fossero ammesse le variabili all’interno della chiusura, quando questa viene chiamata altrove, sarebbero già state deallocate dallo stack in quanto usciti dal corpo della funzione, creando errore a runtime. Ad esempio:



Il Pascal ha funzioni annidate ma non ha chiusure. Come fa? L’idea è che una funzione annidata può essere eseguita solo all’interno della funzione in cui è annidata, questo evita il problema presente in C in quanto le variabili usate nella chiusura non sono state già deallocate quando la chiusura viene chiamata.

**Come implementare le chiusure in C**

Un primo problema da risolvere è che le variabili usate da una chiusura non devono essere deallocate fino a quando la chiusura è accessibile. La soluzione può essere allocare i dati nello heap invece che sullo stack.



Un secondo problema è che una chiusura non può essere rappresentata solo da un entry point, perché il codice non saprebbe accedere alle variabili non locali (ovvero quelle nello heap). La soluzione è che la chiusura sia una coppia formata da:

- puntatore a funzione per il codice.

- un record di puntatori ai dati nello heap.



[ a destra la versione del codice a sinistra implementata in C ]

Questa soluzione ricorda gli oggetti, in cui le funzioni prendono implicitamente (es. this in java) o esplicitamente (es. self in python ) il record con i puntatori al codice dei metodi. La scelta di passare il record esplicitamente (tipo python) dà la possibilità di utilizzare un metodo con un record di puntatori differente da quello dell’oggetto che contiene f originariamente, potendo facilmente implementare polimorfismo.

**Chiusure in linguaggi Higher Order**

Un linguaggio è di ordine superiore quando le funzioni (chiusure) sono oggetti di prima classe. Spesso è presente una sintassi semplificata per scrivere funzioni anonime, ovvero espressioni che definiscono una funzione e ne ritornano la chiusura. In questi linguaggi non vi è nessuna differenza sintattica tra funzioni e chiusure.

**Utilizzo delle chiusure**

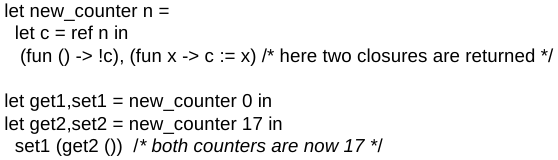
1. Un primo problema è il **data-hiding** ovvero far si che solamente certe parti del codice possano accedere a un determinato dato, ad esempio solo tramite getter/setters.

Benefici:

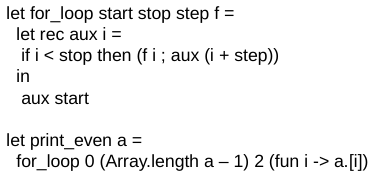
- il debugging è più semplice in quanto meno parti del codice andranno controllate.

- è sufficiente modificare solo porzioni limitate del codice se si vogliono apportare modifiche.

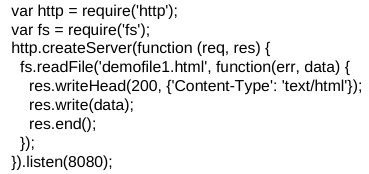
Esempio in OCaml:



2. Un altro interessante utilizzo è l’implementazione di **strutture di controllo**. Tramite funzioni polimorfe è possibile passare in input una funzione (o meglio una chiusura) che esegue del codice in certe condizioni.



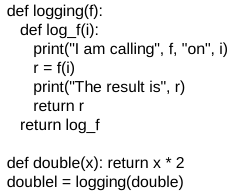
3. È possibile inoltre implementare **callback** nella programmazione asincrona/ad eventi.



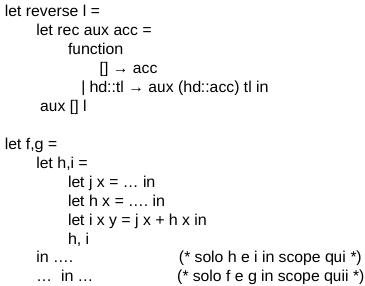
In questo codice si ha una funzione ed una chiusura, **la chiusura per essere tale deve accedere a variabili esterne** (in questo caso res) **ma non globali** (in quanto le globali sono nello heap e sempre accessibili).

NOTA: require(‘...’) altro non è che un record di chiusure.

4. Le chiusure possono essere utilizzate anche per **decorare** altre funzioni:



5. O anche per **limitare l’accessibilità**. Si può limitare una funzione ad avere accesso esclusivamente alle variabili di cui ha bisogno.



Nell’esempio la funzione fa il reverse di una lista. Per fare reverse in modo efficiente viene utilizzato un accumulatore (aux) e non si vuole che sia visibile all’infuori della reverse.

Il secondo esempio è un esempio alla Javascript, anche se la sintassi è del linguaggio ML. Le variabili h e i sono visibili nell’implementazione di f e g (in quanto definite in coppia) ma non all’esterno.

NOTA: le chiusure **non** possono essere utilizzate per limitare l’accessibilità di costrutti non di prima classe. Ad esempio non è possibile usare le chiusure per creare un tipo che non sia poi visibile a tutto il resto del codice (es. eccezioni, o altri tipi di dato).

**In breve**:

- **Una chiusura è una funzione che usa variabili non globali definite negli scope più esterni.**

- Nei linguaggi di alto livello non vi è differenza sintattica fra funzioni e chiusure e le chiusure sono oggetti di prima classe.

- Il linguaggio può supportare una sintassi per definire funzioni anonime (che spesso sono chiusure).

- A *basso livello una chiusura è una coppia (puntatore al codice, record puntatori ai dati)* su cui operare e per certi versi assomiglia ad un oggetto.