Domande teoriche LP
II Parziale

by KatzuCurry 

Linguaggi funzionali

- **Perché i linguaggi funzionali puri ammettono il passaggio dei parametri ad una funzione solo per valore e non, ad esempio, per riferimento?**
  Questo perchè i linguaggi funzionali puri rispettano la trasparenza referenziale, ovvero una funzione è considerata trasparente referenziale se il risultato di una chiamata alla funzione dipende solo dai suoi argomenti (che sono immutabili) e non da stati globali o effetti collaterali. Per rispettare la trasparenza referenziale è necessario che il passaggio dei parametri sia solo per valore e mai per riferimento, poiché la scelta dell'utilizzo di quest'ultimo può portare ad una serie di effetti collaterali come modifiche a variabili globali o modifiche ai parametri passati per riferimento, modificando direttamente il valore di una variabile con effetti che vanno oltre il contesto della funzione stessa. Se questo accadesse, non sarebbe più possibile prevedere il risultato di una funzione poichè all'interno di essa potrebbero esserci modifiche dello stato del sistema che porta una serie di cambiamenti che possono modificare il risultato atteso di suddetta funzione.

- **Caratteristiche dei paradigmi di programmazione funzionale**
  Funzioni di prima classe (possono essere parte di una struttura dati oppure possono essere costruiti a runtime e ritornare come valore di un'altra funzione), dati immutabili, ricorsione al posto dei loop, mancanza di effetti collaterali, funzioni di ordine superiore (prendere altre funzioni come argomenti e ritornare altre funzioni in modo generale).
  Programma = funzione matematica, poichè l'output di una funzione dipende solo dai suoi argomenti e non da stati esterni o variabili globali (Stessa uscita per gli stessi input, assenza di effetti collaterali).

- **Tre fasi per applicare paradigma funzionale: READ-EVAL-PRINT**
  READ: lettura di input, rappresentati in strutture dati appropriate.
  EVAL: valutazione della rappresentazione interna al fine di produrre un valore o più.
  PRINT: il valore ottenuto viene stampato.
  E poi loop.

- **Definizione dello Heap e importanza in linguaggi funzionali e logici.**
  Lo heap è una regione di memoria utilizzata per l'allocazione dinamica di dati durante l'esecuzione di un programma. È utilizzata per la gestione automatica della memoria a breve termine e consente di allocare dati a vita più lunga e dinamica. Grazie a questo, è possibile avere una allocazione dinamica di dati e strutture dati complesse a runtime e una gestione automatica della memoria, in sintesi avere la garbage collector. Inoltre, avendo i dati immutabili nei linguaggi funzionali e logici, lo heap è spesso coinvolto nella gestione di dati immutabili che persistono durante l'esecuzione del programma poichè fornisce uno spazio per la loro allocazione.

Common Lisp

- **Caratteristiche di LISP**
  Paradigma: Funzionale dichiarativa.
  Rispetta alcune caratteristiche dei paradigmi di programmazione funzionale: funzioni di prima classe (possono essere parte di una struttura dati oppure possono essere costruiti a runtime e ritornare come valore di un'altra funzione) , dati immutabili, mancanza di effetti collaterali, funzioni di ordine superiore (prendere altre funzioni come argomenti e ritornare altre funzioni in modo generale). Programma = funzione matematica, poichè l'output di una funzione dipende solo dai suoi argomenti e non da stati esterni o variabili globali (Stessa uscita per gli stessi input, assenza di effetti collaterali), ma essendoci i loop non è considerato puro.
  Tre fasi per applicare paradigma funzionale: READ-EVAL-PRINT-LOOP.

- **Quali sono le espressioni auto-valutanti in LISP? Che cosa fa l'interprete LISP?**
  Le espressioni auto-valutanti sono quelle il cui valore è la propria rappresentazione testuale. Ciò significa che non è necessario valutare ulteriormente queste espressioni, in quanto il risultato è già noto dalla rappresentazione stessa. Le liste non sono autovalutanti perchè richiedono una valutazione per determinare il loro valore (il primo elemento potrebbe essere il nome di una operazione speciale, una funzione definita nell'ambiente oppure una lambda).
  Le espressioni auto-valutanti sono numeri, simboli, caratteri, stringhe, booleani, atomi. Se un simbolo ha un valore nell'ambiente attuale, ritorna il suo valore, altrimenti segnala un errore.
  Se non sono auto-valutanti, ovvero sono cons (o A1 A2 … An), se o è un operatore speciale, viene evaluato in un modo speciale. Se o è un simbolo che denota una funzione nell'ambiente attuale, allora la funzione è applicata alla lista. Se o è una lambda, allora viene applicato alla lista. Altrimenti viene segnalato un errore.

- **Dare una definizione di cons-cells. Quale funzionalità deve fornire l'ambiente in Common LISP a corredo di quest'operazione fondamentale? Ovvero, senza questa funzionalità, che potrebbe accadere?**
  L'ambiente Lisp deve assolutamente offrire la funzionalità di garbage collection. Questo perché (Common) Lisp è un linguaggio ad altissimo livello la cui filosofia reputa la gestione della memoria troppo importante per essere lasciata al programmatore. C'è quindi bisogno di poter disallocare la memoria allocata!
  Potrebbe esserci un memory leak, ovvero potremmo "finire" la memoria!
  L'operazione cons [dalla quale deriva la append], infatti, alloca spazio in memoria: serve a creare cons-cells, strutture dati formate da una coppia di puntatori. Possiamo quindi allocare moltissimo spazio, ma non potendo liberare la memoria (lato programmatore) prima o poi la memoria centrale a disposizione termina. C'è quindi bisogno di un'entità che si curi di liberare le celle di memoria che non servono più al posto del programmatore: questa è il garbage collector.

- **Implementare LISP o Prolog senza garbage collector**
  È possibile, ma sarebbe molto complicato e potrebbe portare una serie di problemi di sicurezza, instabilità del programma e difficoltà di debugging.

- **A cosa si riferisce il concetto di closure in LISP? Darne una definizione.**
  Una closure è una combinazione di una funzione e l'ambiente in cui è stata definita. È una funzione che cattura le variabili dell'ambiente circostante in cui è stata dichiarata, permettendo loro di essere referenziate anche dopo che l'ambiente originale è uscito dallo scope. Quando viene creata una closure, la funzione memorizza l'ambiente in cui è stata creata, inclusi i valori delle variabili circostanti. Questo significa che può "ricordare" e utilizzare quei valori anche dopo che l'ambiente originale è uscito dallo scope.
  Ci permette di creare anche le funzioni lambda.

- **A cosa serve il quote in LISP**
  Operatore speciale utilizzato per evitare la valutazione di una forma. Quando una forma è preceduta da una QUOTE, la forma viene restituita senza essere valutata. È utile quando si desidera trattare un'espressione come dati letterali che come un'espressione da valutare.

- **Quali sono le operazioni principali che allocano memoria in Lisp (nella versione minimale dell'implementazione del linguaggio)?**
  cons è utilizzata per creare una nuova coppia (cons cells) contenente due elementi che può essere utilizzata per costruire strutture dati più complesse come liste e alberi.
  list può essere utilizzato per creare liste di lunghezza fissa.

C

- **Memoria statica e dinamica in C**
  Le memoria statica è allocata a tempo di compilazione e persiste per l'intera durata del programma.
  Le variabili a memoria statica sono dichiarate utilizzando le parole chiave 'static' (dichiarazione di variabili o funzione con visibilità limitata all'interno del file in cui sono definite) o 'extern' (dichiarazione di variabile o funzione definita altrove nel programma) e vengono gestite dal sistema operativo e viene riservata una volta che il programma viene caricato in memoria. Le variabili a memoria statica mantengono il loro valore tra le chiamate di funzione. È più semplice da gestire, ma può essere limitante.
  La memoria dinamica è allocata a tempo di esecuzione e può essere gestita manualmente dal programmatore.
  L'allocazione della memoria dinamica avviene utilizzando le funzioni come 'malloc', 'calloc' e 'realloc'. La liberazione della memoria viene eseguita con 'free'. Viene spesso utilizzata per strutture dati di dimensioni variabili o quando la dimensione della memoria non è nota a tempo di compilazione. Offre maggiore flessibilità, ma richiede una gestione più attenta per evitare memory leaks o accessi illegali alla memoria.

- **Descrivete almeno due usi delle variabili dichiarate static in C.**
  In C la parola chiave "static" può essere utilizzata per dichiarare variabili limitandone la loro visibilità e mantenere lo stato tra le chiamate di funzioni specifiche. Può essere utilizzata per mantenere il valore di una certa variabile tra le chiamate (e non inizializzarle nuovamente ogni volta) oppure per limitare la visibilità alle funzioni del file. Per esempio se una variabile static è dichiarata all'esterno di qualsiasi funzione, il suo ambito di visibilità sarà limitato al file corrente. Questo significa che la variabile non sarà visibile o accessibile da altri file sorgente. Viene usato per dichiarare costanti in un file.

- **Che differenze ci sono all'atto pratico tra usare il pre-processore C per definire delle costanti ed usare invece le (più nuove) funzionalità previste dallo standard del linguaggio per definirle? (Ad esempio (#define E 2.712) rispetto a (const float E = 2.712).**
  Con #define non viene specificato un tipo di dati ma è possibile definire qualsiasi cosa, anche funzioni, e viene fatta una sostituzione del testo da parte del pre-processore. Con 'const' si può specificare un tipo di dati e viene fatto anche un controllo del tipo, dando errore in caso di tipi incompatibili. È possibile regolare la visibilità delle 'const' (se viene messo dentro una funzione è visibile solo a quella funzione), mentre invece le #define hanno visibilità globale e non è possibile modificarlo.

- **Che sottosistema del "runtime" è normalmente presente in Lisp (e Java, Javascript, Prolog, C#, Haskell, Julia…) ma non in C/C++?**
  Garbage collector che si occupa di allocare e deallocare automaticamente la memoria a runtime. In C/C++ è necessario dover allocare e deallocare manualmente la memoria, altrimenti si creano problemi di memory leak.

- **Inclusioni multiple in C.**
  Le inclusioni multiple in C si verificano quando un file di intestazione (header) viene incluso più volte in un file sorgente durante la compilazione. Per prevenire problemi di duplicazione e di dichiarazioni ridondanti, è comune utilizzare le direttive di pre compilazione per garantire che un file di intestazione venga incluso una sola volta.
  ESEMPIO:

```
#ifndef EXAMPLE_H
#define EXAMPLE_H

// Contenuto del file di intestazione

#endif
```

Haskell

- **Differenza tra il linguaggio eager e lazy. Grazie al fatto di avere una valutazione pigra, con Haskell è possibile rappresentare infinite strutture dati.**
  Le loro differenze riguardano principalmente il modo in cui le espressioni vengono valutate e quando viene effettuata l'evaluazione delle espressioni.
  Nei linguaggi con "eager evaluation", le espressioni vengono valutate non appena vengono incontrate e tutte le espressioni vengono valutate in modo sequenziale e il risultato di ciascuna espressione è disponibile prima di procedere all'esecuzione successiva.
  Nei linguaggi con "lazy evaluation", le espressioni vengono valutate solo quando il loro risultato è effettivamente necessario. Questo significa che è possibile ritardare l'evaluazione di un'espressione fino al momento in cui il suo valore è richiesto. Questo è comune in linguaggi di programmazione funzionali come Haskell.

- **Cos'è il type inference. Elenca i vantaggi e svantaggi.**
  Il type inference è una caratteristica di alcuni linguaggi di programmazione che consente al compilatore o all'interprete di dedurre automaticamente il tipo di una variabile senza richiedere una dichiarazione esplicita da parte del programmatore. Il sistema di tipo del linguaggio è in grado di determinare il tipo di una variabile in base al contesto e alle operazioni che vengono svolte su di essa. Quindi in un linguaggio con type inference, il programmatore non è obbligato a specificare il tipo di ogni variabile in modo esplicito, poichè il compilatore o l'interprete è in grado di dedurrò automaticamente. Un esempio di linguaggio con il type inference è Haskell. Più lento e non flessibile.

Julia

- **Function broadcasting or vectorization in Julia.**
  È una caratteristica che permette di applicare operazioni elemento per elemento su array o collezioni di dati, senza dover utilizzare cicli espliciti. Funziona come il mapcar.

- **Differenza tra static dispatch e dynamic dispatch.**
  Sono due approcci distinti per la gestione delle chiamate di funzione in un programma.
  Static dispatch: la decisione su quale funzione chiamare viene presa durante il tempo di compilazione poiché il tipo del destinatario della chiamata di funzione è noto a tempo di compilazione, e il compilatore può generare errore.
  Dynamic dispatch: la decisione su quale funzione chiamare viene presa a tempo di esecuzione, basandosi sui tipi effettivi degli argomenti della funzione. Consente di scrivere funzioni più generiche che possono essere utilizzare con diversi tipi di dati a tempo di esecuzione. Approccio tipico dei linguaggi di programmazione con tipi dinamici o supportano la generalità come Julia.

- **Composizione e piping in Julia.**
  Permettono di combinare funzioni in modo sequenziale.
  Composizione: tecnica che consente di combinare due o più funzioni in una nuova funzione, dove il risultato di una funzione diventa l'input della successiva. Viene usato l'operatore '∘'.
  Piping: tecnica che consente di concatenare funzioni in modo più leggibile, con il risultato di una funzione che diventa l'argomento della successiva. Viene usato l'operatore 'l>'.

Ricorsione

- **Cosa s'intende per ritorsione in coda e quali tipi di programmi essa denota? Che effetto ha una chiamata ricorsiva in coda sullo stack degli activation frames? (Ovvero cosa può fare il compilatore?) Fornire un esempio di codice con ricorsione in coda e un esempio di funzione che non può essere definita mediante una ricorsione in coda.**
  È uno specifico tipo di ricorsione in cui la chiamata ricorsiva è l'ultima operazione eseguita dalla funzione e il risultato non richiede elaborazioni. Inoltre queste funzioni hanno una struttura molto simile ad un ciclo, infatti è possibile convertirli in un loop. Questa struttura rende possibile per alcuni compilatori ottimizzare il codice convertendo la chiamata ricorsiva in una semplice istruzione jump (al livello assembly). Un esempio di ricorsione in coda è il fattoriale e una funzione che non può essere definita mediante una ricorsione in coda è Fibonacci.

- **Ricorsione semplice, doppia (o multipla) e in coda.**
  Semplice: funziona richiama sè stesso direttamente (fattoriale);
  Doppia (o multipla): richiama sè stesso più di una volta in una singola chiamata;
  In coda: la chiamata ricorsiva è l'ultima operazione eseguita all'interno della funzione. Viene ottimizzata dai compilatori, sfruttando una jump.

- **È sempre possibile convertire un programma ricorsivo in un programma con cicli?**
  Sì, tuttavia la conversione può richiedere delle modifiche sostanziali alla struttura del codice oltre che a non essere direttamente intuitiva. Le funzioni tali-recursive possono essere facilmente reversibili in un loop. Per le altre tipologie di ricorsione potrebbe non essere così intuitivo.

- **Quanti activation record sono inseriti sullo stack di runtime di un linguaggio quando viene eseguita una funzione ricorsiva non tail-ricorsiva? Quanti ne sono invece inseriti (sullo stack di runtime) quando viene eseguita una funzione propriamente tail-ricorsiva?**
  In una funzione ricorsiva non tail-ricorsiva, ogni chiamata ricorsiva genera un nuovo activation record nello stack di runtime, poichè la chiamata ricorsiva deve mantenere lo stato corrente della funzione.
  In una funzione ricorsiva tail-ricorsiva, la chiamata ricorsiva è l'ultima operazione eseguita dalla funzione e non richiede alcuna operazione dopo il suo completamento. Grazie a questo, alcuni compilatori possono effettuare un processo di ottimizzazione che permette all'ultima chiamata ricorsiva di utilizzare lo stesso activation record della funzione chiamante, evitando l'accumulo di record di attivazione per ogni chiamata ricorsiva.

DOMANDE RANDOM

- **Cosa sono gli effetti collaterali**
  Gli effetti collaterali nella programmazione si verificano quando una funzione modifica uno stato al di fuori del suo ambito locale o produce un risultato che va oltre il valore restituito dalla funzione stessa. Gli effetti collaterali possono influenzare il comportamento di altre parti del programma, portando a comportamenti imprevisti o complessità aggiuntiva nella gestione del codice.
  Un esempio potrebbe essere quello di modificare il valore di una variabile fuori dallo scope locale o modifiche di variabili globali.

- **Cos'è una lambda expression e cosa è necessario per valutare una lambda expression? Quali sono i vantaggi della lambda expression?**
  Una lambda expression è una forma di espressione anonima utilizzata in molti linguaggi di programmazione per definire funzioni senza dover assegnare loro un nome esplicito. Per valutare una lambda è necessario passare gli argomenti necessari in coda. Il vantaggio è che possono essere utilizzare senza dover assegnare un nome, possono essere usate in "loco" quando è necessario un comportamento specifico solo in un punto della funzione. Si integrano bene con i concetti della programmazione funzionale, come il passaggio di funzioni come argomenti ad altre funzioni. Possiamo creare funzioni che costruiscono altre funzioni mediante lambda e ritornarli come valori. È possibile anche riscrivere in modo più elegante e conciso una funzione. È possibile trasformare una lambda in una let e viceversa.

- **Activation frame: cos'è? Dove si trova? Com'è fatto? Dare una definizione di "activation frame". In che tipo di struttura è inserito durante l'esecuzione di un programma?**
  È lo spazio necessario per salvare tutti i dati necessari per eseguire una procedura F e riprendere l'esecuzione al chiamante dopo la fine dell'esecuzione di tale procedura, in modo particolare per:
- Salvare i valori contenuti nei registri del processore;
- Salvare l'indirizzo di ritorno (nel corpo del chiamante);
- Variabili locali, definizioni e valori di ritorno
- Binding valore-argomento
- Static link: informazioni su dove è definito una procedura;
- Dynamic link: informazioni su come è chiamata una procedura.
  Una chiamata di funzione porta ad effettuare una push di un activation frame sullo stack chiamato evaluation stack e rimane fino a quando non termina l'esecuzione, ovvero quando verrà effettuata una pop.
  In generale, l' activation frame è fondamentale per mantenere lo stato e la coerenza durante l'esecuzione di un programma, gestendo la catena di invocazioni delle funzioni.

- **Cos'è una funzione di ordine superiore**
  Funzione che accetta altre funzioni come argomenti o restituisce una funzione come risultato. Rende possibile trattare le funzioni come dati manipolabili (LISP).

- **Cos'è un puntatore**
  Variabile che contiene l'indirizzo di memoria di un'altra variabile. Un puntatore "punta" a una posizione specifica nella memoria del computer. Consentono di gestire la memoria in modo più efficiente e di manipolare dati in modi avanzati (C).