# Linguaggi di Programmazione - prove d'esame passate e possibili domande

## Esame del 18/06/2013

**Esercizio 1**

*Supponete di avere un header file C su cui avete il controllo: my_stuff.h. Il file my_stuff.h #include il file API.h, il quale #include il file basicAPI.h. Mostrate come devono essere scritti i files my_stuff.h, API.h ed un vostro altro file main.c al fine di evitare inclusioni multiple di header files. Il file main.c deve includere separatamente ed esplicitamente i due files basicAPI.h e my_stuff.h. Notate che il file basicAPI.h si trova nelle cartelle controllate dal sistema, mentre il file my_stuff.h si trova nella cartella dove state sviluppando il progetto.*

Richieste:

- my_stuff.h: #include API.h
- API.h: #include basicAPI.h
- main.c: #include <basicAPI.h> e #include "my_stuff.h"

main.c

```
#include "my_stuff.h"
#include <basicAPI.h>
```

my_stuff.h

```
#ifndef _MY_STUFF_H
#define _MY_STUFF_H
#include "API.h"
#endif
```

API.h

```
#ifndef _API_H
#define _API_H
#include <basicAPI.h>
#endif
```

**Esercizio 2**

*Considerate la seguente espressione aritmetica:*
*tan(pi * 4.2) / (12 - e * 3) + 1024*
*1) Riscrivetela come S-expression*
*2) Scrivete una funzione value che ritorni il valore dell'espressione rappresentata da S-espressioni di questo tipo: considerate solo le costanti "pi" ed "e", e gli operatori + - * / sin cos tan. Se vi servono funzioni ausiliarie, scrivetele pure.*

1)

```
(+ (/ (tan (* pi 4.2)) (- 12 (* 3 e))) 1024)
```

2)

```
(defun value (lista)
  (cond ((null lista) '())
        ((equal lista 'e) 2.17)
        ((equal lista 'pi) 3.14)
        ((atom lista) lista)
        (t (let ((operazione (first lista)))
             (if ((lambda (x)
                    (if (or (equal x '+)
                            (equal x '-)
                            (equal x '*)
                            (equal x '/))
                        T
                      '())) operazione)
                 (funcall operazione
                          (value (second lista))
                          (value (third lista)))
                 (funcall operazione
                          (value (second lista)))))))))
```

**Esercizio 3**

*Una funzione fondamentale in (Common) Lisp, con varianti equivalenti in moltissimi altri linguaggi funzionali, logici e non, è la CONS (in Java e C++ new, ...)*
*1) Quale funzionalità deve fornire l'ambiente Lisp a corredo di quest'operazione fondamentale?*
*2) Ovvero, senza questa funzionalità, che potrebbe accadere durante l'esecuzione di questo programma?*

```
(defun giro-giro-tondo (n l1 l2)
  (when (plusp n)
    (append l1 l2)
    (frulla-frulla (1- n) l1 l2)))
> (giro-giro-tondo quarantadue-fantastiliardi '(4) '(2))
```

1) 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!
2) 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. 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.

# Esame A del 26/01/2015

**Esercizio 1**

*Definire in Common Lisp una funzione nopred che ha per argomenti un predicato e una lista e che restituisce come valore la lista data come argomento SENZA gli atomi -a qualsiasi livello di profondità- che soddisfano il predicato. Attenzione ai NIL per mantenere la struttura della lista*

```
(defun nopred(p l)
  (cond ((null l) NIL)
        ((listp (first l))
            (cons (nopred p  (first l))
                  (nopred p (rest l))))
        ((ignore-errors (funcall p (first l)))
            (nopred p (rest l)))
        (T (cons (first l)
                 (nopred p (rest l))))))
```

**Esercizio 2**

*Dato il seguente programma Common Lisp scriverne uno equivalente che usi una lambda expression*

```
(defun f (x y)
  (let ((a (* (+ (* 2 x y)) 42))
        (b (+ 2 a)))
    (+ (* a (quadrato x))
       (+ b y)
       (* a b))))
```

```
(defun f (x y)
  ((lambda (a b)
    (+ (* a (* x x))
      (+ b y)
      (* a b)))
   (* (+ (* 2 x y)) 42) (+ 2 (* (+ (* 2 x y)) 42))))
```

**Esercizio 3**

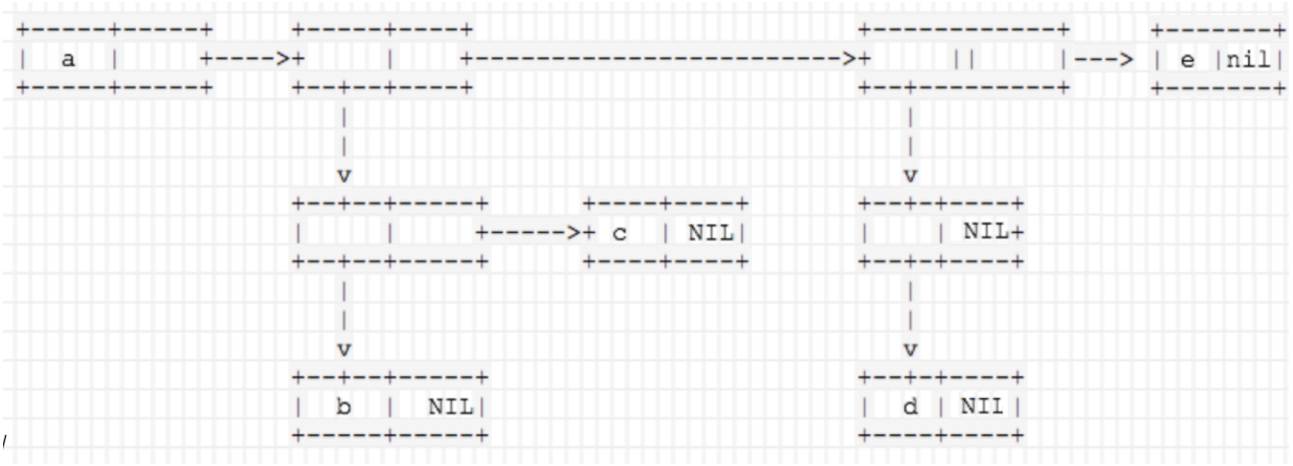
*Dare una definizione di cons-cell*

*Dare una rappresentazione grafica in termini di cons-cell della seguente espressione simbolica*

*(a ((b) c) ((d)) e)*

Una cons-cell è una delle strutture dati chiave (Common) Lisp. Consiste in una coppia di puntatori a due elementi, a cui si accede con car e cdr.

*(cons 'a (cons (cons (cons 'b nil) (cons 'c nil)) (cons (cons (cons 'd nil) nil) (cons 'e nil))))*




**Esercizio 4**

*Dato il seguente programma Common Lisp*

```
(defun what (e n list)
  (cond
      ((zerop n) (cons e list))
      ((null list) (list e))
      (t (cons (car list) (what e (- n 1) (rest list))))))
```

*descriverne il comportamento e calcolarne il valore prodotto dalla valutazione della seguente applicazione:*

*> (what 'h 3 '(a b c d e))*

La funzione what prende in input un atomo e, un numero n e una lista list.
Come risultato, "inserisce" l'atomo e in posizione n in list (partendo a contare dalla posizione 0).
Passi di computazione:

- (what 'h 3 '(a b c d e))
- **a** (what 'h 2 '(b c d e))
- **ab** (what 'h 1 '(c d e))
- **abc** (what 'h 0 '(d e))
- **abchde** [n == 0, quindi esegue *(cons e list)* che produce *(h d e)*
- Quindi ritorna la lista **(a b c h d e)**!

**Esercizio 5**

*Cosa si intende per ricorsione in coda e quali tipi di programmi essa denota?*
*Fornire un esempio di codice con ricorsione in coda e un esempio di funzione che non può essere definita mediante una ricorsione in coda.*

Il termine "ricorsione in coda" indica programmi/funzioni che richiamano se stessi come ultima istruzione: non ci sono altre operazioni da eseguire quando arriva il valore della chiamata ricorsiva! Una funzione tail-ricorsiva può essere convertita in un ciclo dal compilatore ottimizzando il codice, evitando quindi la necessità di costruire activation frame, pusharli nello stack, passare il controllo alla funzione chiamata, riottenere il controllo, poppare l'activation frame. Questa tipologia di ricorsione è quindi più performante e evita il rischio di stack overflow.

La funzione fattoriale può essere scritta nel seguente modo

```
(defun fatt-ciclo (n acc)
  (if (= n 0)
    acc
    (fatt-ciclo (- n 1) (* n acc))))
(defun fattoriale (n)
  (fatt-ciclo n 1))
```

Ovvero la funzione fattoriale può essere riscritta in modo tale da riutilizzare uno degli argomenti come un accumulatore! Questa è una funzione tail ricorsiva.

Una funzione non definibile come tail-ricorsiva è quella che ritorna l' *n*-esimo elemento della sequenza di Fibonacci.

Un altro esempio di funzione con ricorsione in testa è il fattoriale senza accumulatore.

```
(defun fattoriale(n)
  (if (= n 0)
   1
   (* n (fattoriale (- n 1)))))
```

I risultati intermedi vengono salvati in uno stack!

**Esercizio 6**

*Scrivere la dichiarazione di un puntatore a una costante di tipo char*

```
const char* pc;
```

## Esame B del 26/01/2015

**Esercizio 1**

*Definire in Common Lisp una funzione filtra che ha per argomenti un predicato e una lista e che restituisce come valore la lista data come argomento CON SOLO gli atomi -a qualsiasi livello di profondità- che soddisfano il predicato. Attenzione ai NIL per mantenere la struttura della lista*

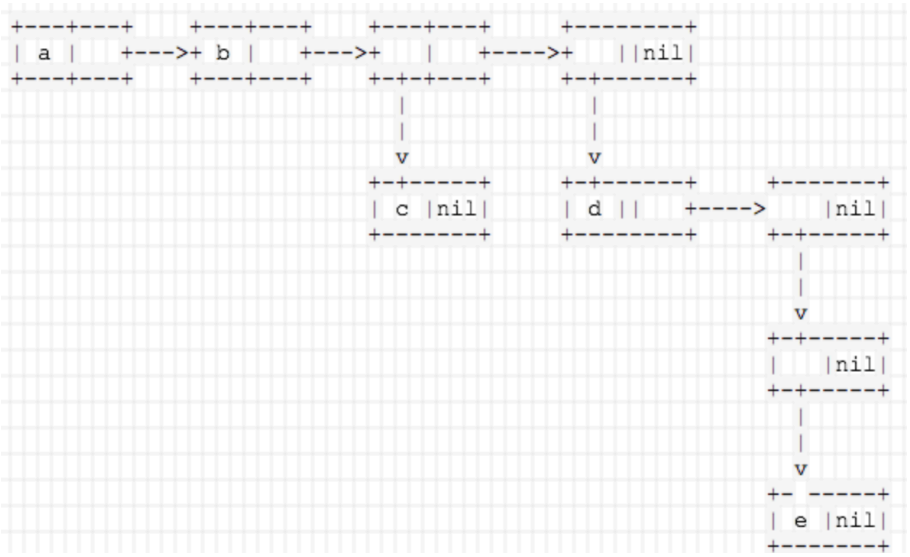
```
(defun filtra (fun lst)
 (cond ((null lst) nil)
      ((listp (first lst))
       (cons (filtra fun (first lst))
            (filtra fun (rest lst))))
      (t
       (if (ignore-errors (funcall fun (first lst)))
          (cons (first lst)
               (filtra fun (rest lst)))
        (filtra fun (rest lst))))))
```

**Esercizio 3**

*Dare una definizione di cons-cell*

*Dare una rappresentazione grafica in termini di cons-cell della seguente espressione simbolica (a b (c) (d ((e))))*

Una cons-cell è una delle strutture dati chiave (Common) Lisp. Consiste in una coppia di puntatori a due elementi, a cui si accede con car e cdr.




**Esercizio 4**
*Data la seguente funzione Common Lisp:*

```
(defun what (functions value)
  (unless (null functions)
    (cons (funcall (first functions) value)
          (what (rest functions) value))))
```

*Descriverne il comportamento e calcolare il valore prodotto chiamando:*
*(what (list 'numberp 'symbolp 'oddp) 3)*

La funzione what applica, una ad una, le funzioni nella lista functions a value e concatena in una lista tutti i risultati ottenuti.
In questo caso otteniamo:
(T NIL T)

**Esercizio 5**
*Dare una definizione di lambda expression e dire in che situazioni è fondamentale utilizzarla*
Una lambda-expression è una funzione anonima. Serve a scrivere una funzione veloce, senza definirla prima tramite defun. Il suo utilizzo è fondamentale in fase di compilazione del codice, in quanto le let vengono convertite in lambda.

**Esercizio 6**
*Scrivere in C le definizione di una costante di tipo puntatore a double*
double * const puntatore_costante_a_double;

# Esame del 21/07/2015

**Esercizio 1**

*Costruite la funzione reduce che prende una funzione binaria (arietà 2) e una lista e produce come risultato la combinazione (associativa a sx) degli elementi della lista. Potete assumere che la lista abbia almeno due elementi*

```
(defun reduce* (fun elements)
  (if (= (length elements) 2)
      (funcall fun
               (first elements)
               (second elements))
    (reduce* fun
             (append (list (funcall fun
                                    (first elements)
                                    (second elements)))
                     (rest (rest elements))))))
```

**Esercizio 2**

*Data la seguente funzione LISP:*

```
 (defun multiplex (functions value)
  (if (null functions)
      nil
    (cons (funcall (first functions) value)
          (multiplex (rest functions) value))))
```

*Descriverne il comportamento e calcolare il valore che si ottiene passandogli i due argomenti seguenti*
*(list 'list 'zerop (lambda (x) (+ x 42)) '/)*
*42*

La funzione multiplex applica, una ad una, le funzioni nella lista functions a value e concatena in una lista tutti i risultati ottenuti.
In questo caso otteniamo:
((42) NIL 84 1/42)

**Esercizio 3**

*Perché i linguaggi funzionali puri ammettono il passaggio dei parametri ad una funzione solo per valore e non, ad esempio, per riferimento?*

Perché alla base dei linguaggi funzionali puri c'è il concetto di trasparenza referenziale, secondo il quale il significato del tutto si può ricavare dal significato delle parti. Passando i parametri per riferimento, questi potrebbero essere modificati e quindi non potremmo essere certi che la funzione chiamante, nel caso li utilizzasse successivamente, ritorni il valore che ci aspettiamo. Il passaggio di parametri solo per valore è uno dei requisiti necessari affinché vi sia la trasparenza referenziale.

## Esame del 29/06/16

**Esercizio 3**

*Scrivete la definizione in C per una "matrice"; la definizione deve essere una struct che contiene il numero di righe, il numero di colonne e l'array di double che contiene ogni elemento della matrice. Definite anche una typedef matrix appropriata.*
*Scrivete la funzione new_matrix che prende il numero di righe e il numero di colonne e restituisce una nuova matrice con tutto lo spazio necessario ottenuto tramite malloc.*

```
#include <stdio.h>
#include <stdlib.h>

struct matrice {  int n_colonne;
                  int n_righe;
                  double *valori;
                };
typedef struct matrice* matrix;

matrix new_matrix(int rows, int columns){
  matrix matrice = (matrix) malloc(sizeof(matrice));
  matrice->n_colonne = columns;
  matrice->n_righe = rows;
  matrice->valori = (double *) malloc(columns * rows * sizeof(double));
  return matrice;
}

void fill_matrix(matrix m, double values[]){
  int contatore = 0;
  for(int i = 0; i < m->n_righe; i++){
    for(int j = 0; j < m->n_colonne; j++){
      m->valori[contatore] = values[contatore];
      printf("%f\t", values[contatore]);
      contatore++;
    }
    printf("\n");
  }
}

int main(){
  matrix matrice_prova = new_matrix(9, 4);
  double lista_valori[50]; /*Se passo più valori, questi vengono ignorati*/
  for(int i = 0; i < 50; i++) lista_valori[i] = i;
  fill_matrix(matrice_prova, lista_valori);
}
```

## Esame del 26/02/2016

**Esercizio 1**

*Considerate le funzioni Common Lisp qui sotto:*

```
(defun foo (l)
   (if (not (null l)) (/ (foo-aux l) (length l))))

(defun foo-aux (l)
   (if (null l)
      0
      (+ (first l) (foo-aux (rest l)))))
```

*(Si noti che foo-aux è una funzione ausiliaria che potrebbe essere definita "localmente" all'interno di foo)*

*1. Che operazione computa foo?*

*2. Che accade quando si sottopone all'interprete CL la seguente espressione?*

```
> (foo ())
```

1. foo computa la media aritmetica dei valori nella lista passata in input.
2. ritorna NIL

**Esercizio 2**

*Scrivete in LISP una funzione controlla che ha per argomento una lista e restituisce T se la lista data non contiene sottoliste, NIL altrimenti.*

```
(defun controlla (x)
  (cond ((null x) t)
        ((listp (first x)) nil)
        (t (controlla (rest x)))))
```

## Un po' di domande random

**1) Activation frame: cos'è? Dove si trova? Com'è fatto?**

**1b) Dare una definizione di "activation frame". In che tipo di struttura è inserito durante l'esecuzione di un programma?**

Un activation frame è un blocco di memoria che si trova sullo stack di sistema. Quando chiamiamo una funzione, viene pushato un activation frame sullo stack.

Questo contiene:

- static link
- dynamic link
- indirizzo di ritorno
- spazio per variabili passate come parametri, variabili e definizioni locali, valori di ritorno
- spazio per i registri da salvare prima di richiamare una sotto-funzione

**2) Definire cos'è una lambda expression**
**2b) Cos'è una lambda expression e a cosa serve?**
Una lambda-expression è una funzione anonima. Serve a scrivere una funzione veloce, senza definirla prima tramite defun. Esempio: *((lambda (x) (+ 42 x)) 0) ---> 42*

**3) Scrivere il codice della funzione Common Lisp che, dati in input una lista e un numero, ritorna la lista senza quel numero**

```
(defun remove-number (number lst)
  (if (null lst) nil
      (if (= (first lst) number)
          (remove-number number (rest lst))
          (append (list (first lst))
                  (remove-number number (rest lst))))))
```

**4) Cosa sono le espressioni autovalutanti in (Common) Lisp?**
Sono espressioni il cui valore ha la stessa rappresentazione tipografica dell'espressione tipografica che li denota

**5) Procedimento generale per trasformare una let in una lambda**
Si definisce una funzione lambda che utilizza *n* parametri, che vengono passati subito dopo la definizione. I valori associati ai parametri con la let vengono scritti alla fine della (i.e. passati alla) lambda!

**6) Quali sono le tre fasi che permettono di applicare il paradigma funzionale?**
Definizione, applicazione, valutazione (???)

**7) Qual è la differenza fra extern e static in C?**
extern: l'oggetto dichiarato ha definizione non locale. Quindi è definito o più avanti in questo file, o in un altro file. Si usa nei files di interfaccia.
static: la dichiarazione/definizione viene "fissata" nello spazio di memoria globale. Non è quindi visibile da altri file! Inoltre, una variabile static mantiene il suo valore fra più chiamate della stessa funzione ("fissa" il suo valore nello scope della funzione). Si usa per definizioni globali in un file.

**8) Dichiarare un puntatore a una funzione che riceve come parametro un intero e una stringa**
(assumo che non ritorni nulla)

```
void (*puntatore_a_funzione) (int, char[]);
puntatore_a_funzione = funzione_puntata;
```

## Un po' di codici random

```
(defun rimuovi-ad-ogni-livello (lista oggetto-brutto)
  (if (null lista) nil
     (let ((head (first lista)))
          (cond ((atom head)
               (if (equal head oggetto-brutto)
                    (rimuovi-ad-ogni-livello (rest lista) oggetto-brutto)
                    (cons head
                         (rimuovi-ad-ogni-livello
                          (rest lista)
                          oggetto-brutto))))
              ((listp head)
               (cons
                   (rimuovi-ad-ogni-livello head oggetto-brutto)
                   (rimuovi-ad-ogni-livello (rest lista) oggetto-brutto)))))))

(defun conta-numeri-in-lista(x)
  (cond ((null x) 0)
       ((numberp (first x)) (1+ (conta-numeri-in-lista (cdr x))))
       (t (conta-numeri-in-lista (cdr x)))))
```