# Informatica 3 - prova intermedia dell'8 maggio 2003 – soluzioni

**Esercizio 1**

Si consideri il seguente frammento di programma scritto in un linguaggio ipotetico che consente di dichiarare

- variabili globali
- funzioni all'interno di funzioni

e che adotta regole di visibilità (scope) statiche:

```
integer x, y, z; //globali
void fun3 (); //prototipo della funzione, necessario perchè fun1
possa chiamare fun3
void function fun1 (integer m){ // NB: il parametro m è passato per
valore
          integer x,  n;
     void fun2 ( ) {
                              integer z;
                              x = z + m + y;    (*)
     };
     …….
     m = x + n + z;                        (**)
     ……
};
void fun3 () {
          integer x, m, z, w;
          …….
          fun1(z);
          …….
};
……..
void main {
          …….
          fun3();
          …….
};
```

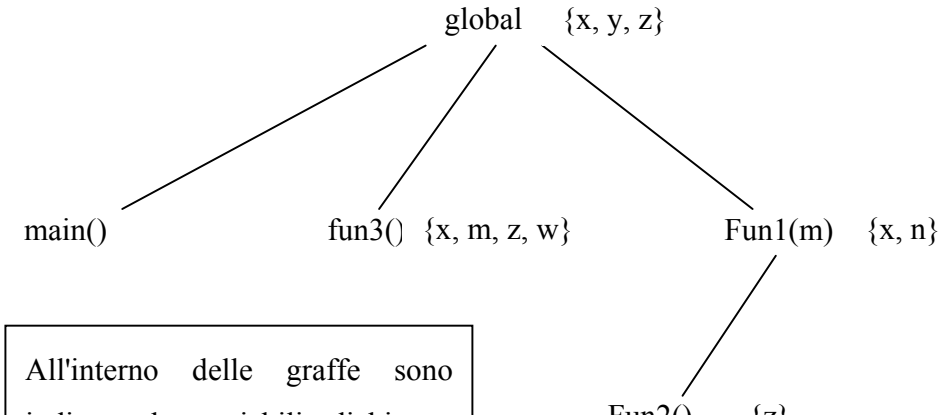
(a) calcolare l'attributo (distanza, offset) per le variabili che appaiono nell'istruzione (*);

(b) calcolare l'attributo (distanza, offset) per le variabili che appaiono nell'istruzione (**);

(c) schizzare lo stato della memoria (mostrando i links statici e dinamici la disposizione delle variabili) nel caso in cui main chiami fun3, che chiama fun1, che chiama fun2, che chiama fun3, che chiama fun1.

Soluzione Esercizio 1

Static nesting tree:




All'interno delle graffe sono indicate le variabili dichiarate localmente alle routine

a) x (1, 4)

   z (0, 3)

   m (1, 3)

   y (2, 1)

b) m (0, 3)

   x (0, 4)

   n (0, 5)

   z (1, 2)

c)

| | | |
|---|---|---|
| | 0 | Current |
| | 1 | Free |
| global | 2 | x in global |
| | 3 | y in global |
| | 4 | z in global |
| main | 5 | RP # |
| | 6 | DL # |
| | 7 | SL 2 |
| fun3 | 8 | RP # |
| | 9 | DL 5 |
| | 10 | SL 2 |
| | 11 | x |
| | 12 | m |
| | 13 | z |
| | 14 | w |
| fun1 | 15 | RP # |
| | 16 | DL 8 |
| | 17 | SL 2 |
| | 18 | m |

| | | |
|---|---|---|
| | 19 | x |
| | 20 | n |
| fun2 | 21 | RP # |
| | 22 | DL 15 |
| | 23 | SL 15 |
| | 24 | z |
| fun3 | 25 | RP # |
| | 26 | DL 21 |
| | 27 | SL 2 |
| | 28 | x |
| | 29 | m |
| | 30 | z |
| | 31 | w |
| fun1 | 32 | RP # |
| | 33 | DL 25 |
| | 34 | SL 2 |
| | 35 | m |
| | 36 | x |
| | 37 | n |

Vediamo lo stesso problema risolto inserendo spazio anche per i return value

| | | |
|---|---|---|
| | 0 | Current |
| | 1 | Free |
| global | 2 | x in global |
| | 3 | y in global |
| | 4 | z in global |
| main | 5 | RP # |
| | 6 | DL # |
| | 7 | SL 2 |
| | 8 | RV |
| fun3 | 9 | RP # |
| | 10 | DL 5 |
| | 11 | SL 2 |
| | 12 | x |
| | 13 | m |
| | 14 | z |
| | 15 | w |
| | 16 | RV |
| fun1 | 17 | RP # |
| | 18 | DL 9 |
| | 19 | SL 2 |
| | 20 | m |
| | 21 | x |
| | 22 | n |
| | 23 | RV |
| fun2 | 24 | RP # |
| | 25 | DL 17 |
| | 26 | SL 17 |
| | 27 | z |
| | 28 | RV |
| fun3 | 29 | RP # |
| | 30 | DL 24 |
| | 31 | SL 2 |
| | 32 | x |
| | 33 | m |
| | 34 | z |
| | 35 | w |
| | 36 | RV |
| fun1 | 37 | RP # |
| | 38 | DL 29 |
| | 39 | SL 2 |
| | 40 | m |
| | 41 | x |
| | 42 | n |

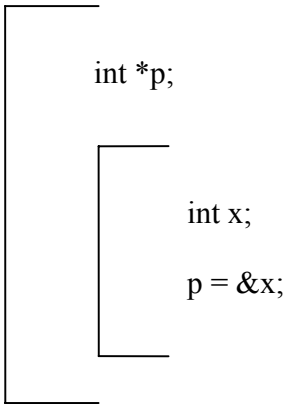
**Esercizio 2**

**1. Si consideri la seguente istruzione di un programma C:** `p=&x;`

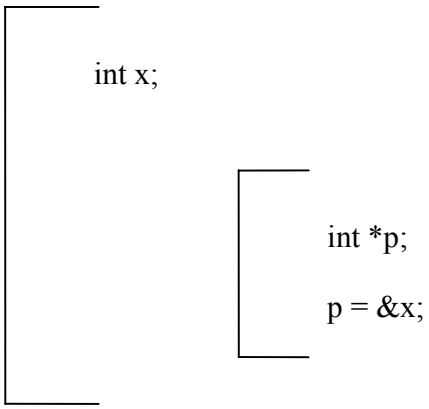
**In quale dei seguenti due casi si genera un "dangling reference"?**

a) **la dichiarazione int x; appare nello stesso "scope" in cui si trova l'istruzione e questo scope risulta interno a quello in cui appare la dichiarazione int * p;**
b) **la dichiarazione int * p; appare nello stesso "scope" in cui si trova l'istruzione e questo scope risulta interno a quello in cui appare la dichiarazione int x;**

Il caso a) corrisponde alla seguente situazione:




Il caso b) corrisponde a quest'altra situazione:



Il dangling reference si può originare solo nel caso a) quando

- nello statement in cui x è dichiarato si fa puntare p ad x
- si esce dallo statement in cui x è dichiarato (quindi x viene deallocato)
- fuori dallo statement in cui x è dichiarato (fuori dallo scope di x) si cerca di accedere alla cella puntata da p.

**2. In Java un attributo di tipo private è visibile all'interno del package in cui la classe è dichiarata. Vero o falso?**

Falso: solo gli attributi protected e package sono visibili all'interno del package in cui la classe è dichiarata.

**3. Supponendo che in un linguaggio simil-Java le istruzioni catch per la gestione di eccezioni possano solo essere associate al corpo dei metodi, quale delle seguenti informazioni deve essere presente nel record di attivazione per permettere la propagazione dell'eccezione (qualora essa non venga gestita dal metodo)?**

**a) link dinamico al chiamante;**

**b) link statico all'ambiente non locale;**

La risposta corretta è la a): un'eccezione deve essere notificata al metodo chiamante (risalendo di un passo la catena dinamica).

Il metodo che include staticamente il metodo chiamato (quello che si ottiene risalendo di un passo la catena statica) non deve gestire le eccezioni sollevate quando il metodo staticamente incluso è stato chiamato da altri.

**Esercizio 3**

**Che cosa calcola la seguente funzione LISP, e sotto quale ipotesi?**

```
(defun guess (lis) (if (null lis)
                       1
                       (* (car lis) (guess (cdr lis))) ) )
```

La funzione calcola la produttoria della lista sotto le seguenti ipotesi:

- che la lista passata come parametro sia piatta
- che la lista sia o nulla o composta da atomi non nulli
- che gli atomi della lista siano numeri

**Trasformare la definizione della funzione `guess` in una forma equivalente, definendo un'opportuna funzione *tail-recursive* `guessTail` come suggerito nel seguito, completando le parti mancanti**

```
(defun guess (lis) (guessTail lis 1) )

(defun guessTail (lis n) (if (      null lis      )
      n
      (guessTail (   cdr lis   ) (   *      (car lis)      n    )) ) )
```

**Trasformare la funzione `guessTail` in un'equivalente funzione `guessIter` (anch'essa con due parametri come `guessTail`) nella quale la chiamata ricorsiva è sostituita da un'istruzione iterativa, come suggerito nel seguito, completando le parti mancanti**

```
(defun guess (lis) (guessIter lis 1) )

(defun guessIter (lis n)
  (do    ((lis1   lis    (   cdr lis1    ))
          (n1     n     ( * (car lis1)  n1  )) )
      (( null lis1  )     n1)  )  )
```

**Esercizio 4**

**Un tipico problema di programmazione concorrente considera un conto corrente bancario (modellato attraverso un *task* del linguaggio Ada o una classe di Java) al quale più utenti (modellati anch'essi da *task* Ada e da classi Java) possono accedere in modo concorrente per effettuare operazioni di versamento e di prelievo (realizzate mediante *entry* in Ada e metodi in Java aventi, a esempio, il nome di "deposita" e di "ritira").**

**Si considerino le seguenti due varianti del problema.**

a) **È definito un importo massimo prelevabile, e l'accesso al conto corrente da parte di un utente che vuole effettuare un prelievo è possibile solo se, indipendentemente dall'importo che l'utente intende effettivamente prelevare, nel conto corrente è presente un importo tale da rendere possibile il prelievo della cifra massima senza "andare in rosso"; in caso contrario il processo che intende effettuare il prelievo viene sospeso fino a quando la somma disponibile sul conto corrente non raggiunge un valore adeguato a permettere il prelievo dell'importo massimo.**

b) **L'accesso all'utente che vuole effettuare un prelievo è permesso se l'importo che si intende prelevare è inferiore all'attuale disponibilità, altrimenti il processo cliente viene sospeso fino a quando l'importo disponibile sul conto corrente raggiunge il valore del prelievo che si intende effettuare.**

**In entrambe le varianti del problema, le operazioni di versamento sono sempre ammesse (pur dovendo ovviamente garantire per ognuna di esse la mutua esclusione con qualsiasi altra operazione).**

**Si dica quali delle soluzioni applicative (a) o (b) sono realizzabili con semplici programmi in Ada o in Java, motivando sinteticamente la risposta nel caso negativo. Ove possibile, tratteggiare, per le sole parti relativa alla sincronizzazione tra i processi, i programmi che realizzano le operazioni di versamento e prelievo.**

**NB: Non è necessario che le soluzioni siano rispettose della sintassi del linguaggio (in particolare, ciò vale per Ada). E' invece importante che la soluzione fornita sia rispettosa della semantica dei costrutti del linguaggio.**

Soluzione Esercizio 4.

```
public clsss ContoCor {                                  // versione (a) Java
      private int disponibile;
      private int soglia;

      ContoCor () {... disponibile=0; soglia=2000; ...}

      public synchronized void deposita(int importo){
            disponibile += importo;
            notifyAll();
      }

      public synchronized void ritira(int importo){
            while (!(disponibile >= soglia) )
                  try { wait(); }   //aspetta di raggiungere la soglia
                  catch(InterruptedException e) {}
            disponibile -= importo;
      }
}
```

```
task ContoCor is                                  -- versione (a) Ada
      entry deposita(importo: in INTEGER);
      entry ritira(importo: in INTEGER);
end ContoCor;

task body ContoCor is
      disponibile: INTEGER := 0;          soglia := 2000;
      loop
            select
                  when disponibile >= soglia accept ritira(importo: in
INTEGER) do
                              disponibile := disponibile – importo;
                        end ritira;
                  or
                  when true accept deposita(importo: in INTEGER) do
                              disponibile := disponibile + importo;
                        end deposita;
            end select;
      end loop;
end ContoCor;
```

In Ada non esiste una semplice variante del caso (b) perché nella clausola *when* di una *select* non è possibile valutare il parametro della *entry* citata nella corrispondente *accept*; il valore del parametro risulta noto solo DOPO l'esecuzione della *accept* e quindi non può essere utilizzato per decidere l'esecuzione dell'*accept* stessa.

```
public clsss ContoCor {                    //versione (b) Java
      ... TUTTO COME SOPRA ...

      public synchronized void ritira(int importo){
            while (!(disponibile >= importo) )
                  try { wait(); }   //aspetta di raggiungere l'importo
                  catch(InterruptedException e) {}
            disponibile -= importo;
      }
}
```