Corso di Programmazione

Esame del 25 Giugno 2012

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Procedure con argomenti procedurali

Considera un modello semplificato del processo di *campionamento* e *quantizzazione* di un segnale, simile a quello in uso per codificare un suono in formato *digitale*, per esempio in un CD audio. Originariamente il segale (suono) è di tipo *analogico* ed è rappresentato da una funzione *f* con argomenti e valori reali (andamento nel tempo della pressione dell'aria, a cui l'orecchio è sensibile). I valori *f(t)* vengono campionati ad intervalli regolari *dt* (per i CD la frequenza di campionamento è 44.1 kHz, cioè *dt* = 1/44100 sec). Quindi vengono quantizzati, cioè rappresentati da numeri interi (di 16 bit nel caso dei CD), opportunamente riscalati; tale operazione, che produce l'intero *q(f(t))*, può essere convenientemente rappresentata da una funzione *q* con argomenti reali e valori interi.

Formalizza in Scheme una procedura `samples` che, dati gli argomenti procedurali *f*, *q* con le caratteristiche descritte sopra, dato l'intervallo di campionamento *dt* (numero reale) e dati gli istanti iniziale *t0* e finale *t1* (numeri reali) fra i quali si applica il processo, restituisce la lista dei campioni quantizzati relativi agli istanti *t0*, *t0+dt*, *t0+2dt*, *t0+3dt*, e così via finché non si va oltre *t1*. Nel seguente esempio un periodo di sinusoide viene campionato e quantizzato su 5 bit (`inexact->exact` è la procedura di conversione da *floating-point* a interi e `pi` rappresenta la costante *pi greco*):

```
(samples
  sin
  (lambda (x) (inexact->exact (round (* 15 x))))
  (/ pi 10)
  0  (* 2 pi)
 )
```

$\longrightarrow$

```
'( 0  5  9  12  14  15  14  12  9  5
   0 -5 -9 -12 -14 -15 -14 -12 -9 -5
   0 )
```

2. Memoization

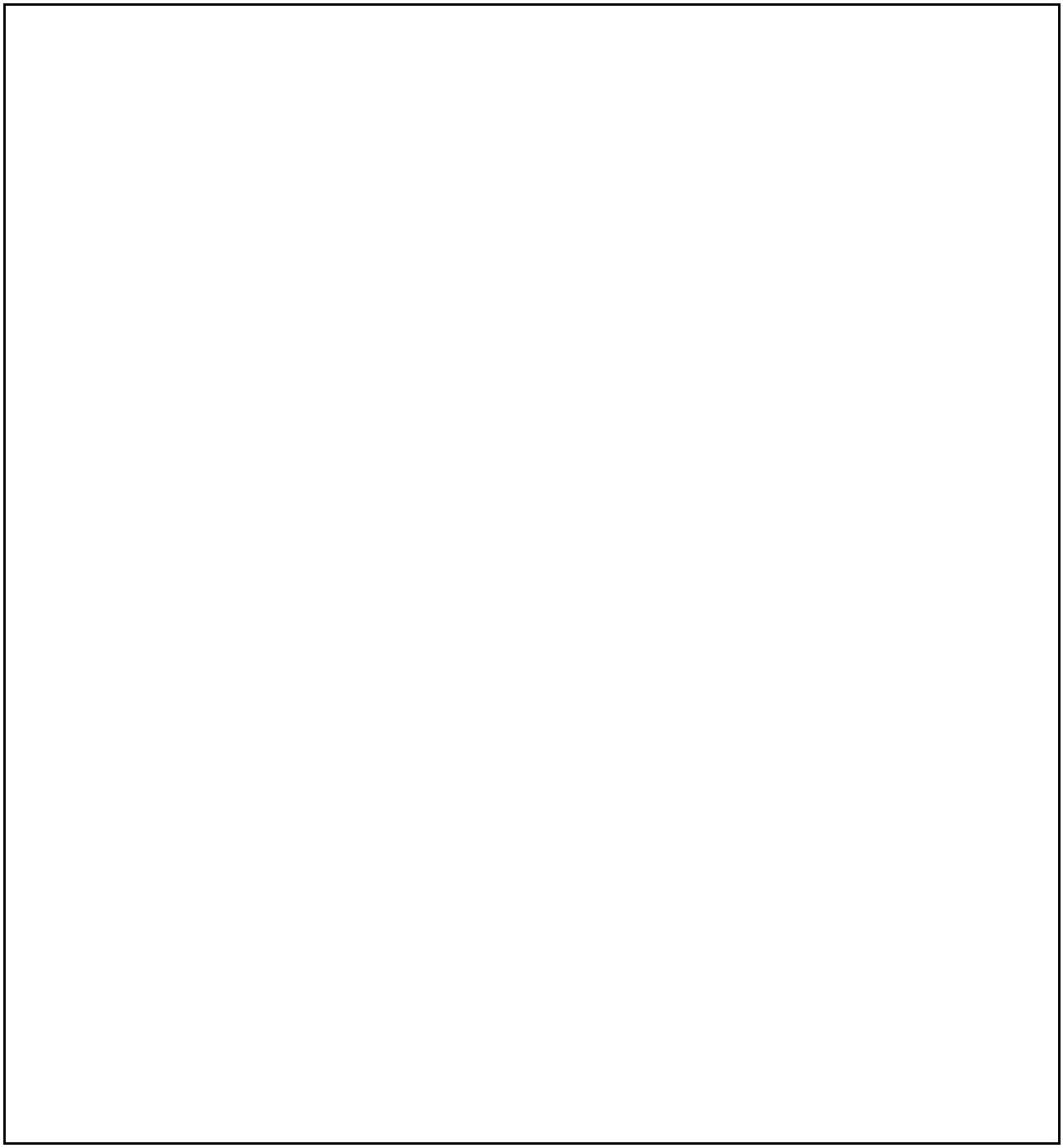
Considera il seguente programma ricorsivo in Scheme:

```
(define paths
  (lambda (i j k)
    (if (or (= i 0) (= j 0))
        1
        (+ (down (- i 1) j k) (right i (- j 1) k))
        )))

(define down
  (lambda (i j k)
    (cond ((and (= k 0) (> j 0)) 0)
          ((or (= i 0) (= j 0)) 1)
          (else
           (+ (down (- i 1) j k)
              (right i (- j 1) (- k 1))
              ))
          )))

(define right
  (lambda (i j k)
    (cond ((and (= k 0) (> i 0)) 0)
          ((or (= i 0) (= j 0)) 1)
          (else
           (+ (down (- i 1) j (- k 1))
              (right i (- j 1) k)
              ))
          )))
```

Trasformalo in un programma più efficiente in Java applicando opportunamente la tecnica di *memoization*.



3. Programmi in Scheme

Scrivi un programma in Scheme per risolvere il seguente problema: data una lista di parole (stringhe) si vuole costruire una lista "ripulita" dalle ripetizioni. Nella lista risultante le parole diverse devono comparire nello stesso ordine della lista di partenza, conservando solo la prima occorrenza in caso di ripetizione. Per esempio, a partire dalla lista:

```
("pippo" "pluto" "felix" "yogi" "yogi" "bubu" "braccobaldo" "felix"
                  "topolino" "eta-beta" "bubu" "bubu" "linux" "topolino" "pluto" "felix")
```

si vuole ottenere la lista:

```
("pippo" "pluto" "felix" "yogi" "bubu" "braccobaldo" "topolino" "eta-beta" "linux")
```

4. Asserzioni e invarianti

Il metodo statico `fourthPower` è progettato per calcolare la quarta potenza dell'intero passato come argomento *senza mai eseguire prodotti* (`*`). Completane il codice basandoti sulle proprietà *invarianti* riportate in corrispondenza ai due comandi iterativi.

```
public static int fourthPower( int n ) {        // Pre:   n ≥ 0
  int  x = 0,  y = 0,  z = 1,  t = n;
  while ( x < t ) {                              // Inv1:  0 ≤ x ≤ n,  y = x^2,  z = 2x+1,  t = n
    x = x + 1;  y = y + z;  z = z + 2;
  }

  t = ..... ;
  while ( x < t ) {                              // Inv2:  n ≤ x ≤ n^2,  y = x^2,  z = 2x+1,  t = n^2

    .....
  }

  return ..... ;   // Post:  y = n^4
}
```

5. Classi in Java

Considera il modello della scacchiera realizzato dalla classe `Board` per affrontare il rompicapo delle *n* regine. In relazione alla versione discussa a lezione, per le istanze di `Board` è definito il protocollo richiamato qui sotto:

```
Board( int n )  // costruttore

void addQueen( int i, int j )
void removeQueen( int i, int j )
int size()
int queensOn()
boolean underAttack( int i, int j )
String arrangement()
```

Ora si vuole migliorare la classe `Board` in due aspetti:

(i) Estendendo il protocollo con un nuovo metodo `boolean free( int i, int j )` che consenta di verificare se il quadrato di coordinate *i, j* è libero, nel senso che non è stata collocata una regina *esattamente* in quella posizione.

(ii) Assicurando che il risultato delle operazioni `addQueen` e `removeQueen` sia sempre consistente, in particolare che non sia possibile collocare più di una regina sullo stesso quadrato. Se gli argomenti di `addQueen` fanno riferimento a un quadrato occupato da una regina, oppure se gli argomenti di `removeQueen` fanno riferimento a un quadrato libero, l'operazione deve lasciare la configurazione della scacchiera inalterata, senza produrre alcun effetto.

Integra la definizione della classe `Board` in modo da soddisfare i requisiti (i) e (ii). Riporta solo le modifiche che si rendono necessarie, indicando chiaramente in quali punti del codice della classe vanno introdotte; non riscrivere, invece, le parti che restano inalterate rispetto alla versione discussa a lezione.