Corso di Programmazione

II Accertamento del 15 Giugno 2010

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Astrazione sui dati in Scheme

Modifica la definizione del dato astratto *scacchiera* discussa a lezione, e richiamata qui di seguito, in modo tale che la procedura `arrangement` restituisca la lista delle coppie di coordinate corrispondenti alle posizioni delle regine. Una configurazione verrà quindi descritta da una lista anziché da una stringa; per esempio, le due soluzioni relative alla scacchiera 4x4 saranno rappresentate dalle liste `'((4 3) (3 1) (2 4) (1 2))` e `'((4 2) (3 4) (2 1) (1 3))`. Riporta soltanto le modifiche del codice, facendo riferimento alla numerazione delle righe interessate.

```
1   (define empty-board             ; scacchiera vuota n x n
2     (lambda (n)
3       (list  n  0  (lambda (i j) #f)  "")
4       ))

5   (define size car)               ; dimensione scacchiera

6   (define queens-on cadr)         ; numero regine collocate sulla scacchiera

7   (define under-attack?           ; la posizione <i,j> e' sotto scacco?
8     (lambda (board i j)
9       ((caddr board) i j)
10      ))

11  (define add-queen               ; valore:  scacchiera con una nuova regina
12    (lambda (board i j)           ;          in posizione <i,j>
13      (list
14       (car board)
15       (+ (cadr board) 1)
16       (lambda (u v)
17         (if (or (= u i) (= v j) (= (- u v) (- i j)) (= (+ u v) (+ i j)))
18             #t
19             ((caddr board) u v)
20             ))
21       (string-append (code i j) (cadddr board))
22       )))

23  (define arrangement cadddr)     ; lista indici di colonna delle regine
```

```
3       (list  n  0  (lambda (i j) #f)  null)


21       (cons (list i j) (cadddr board))
```

2. Memoization

Considera il seguente metodo statico formalizzato nel linguaggio Java:

```
public static long f( int i, int j ) {  // i, j ≥ 0
  if ( i+j < 2 ) {
    return  i+j;
  } else if ( j == 0 ) {
    return  f( 1, i-2 ) + f( 0, i-1 );
  } else if ( j == 1 ) {
    return  f( 0, i ) + f( i+1, 0 );
  } else {
    return  f( i+2, j-2 ) + f( i+1, j-1 );
}}
```

Trasforma il programma ricorsivo applicando opportunamente la tecnica di *memoization*.

```
public static long fm( int i, int j ) {

  long[][] h = new long[ i+j+1 ][];

  for ( int u=0; u<h.length; u=u+1 ) {
    h[u] = new long[ h.length-u ];
    for ( int v=0; v<h[u].length; v=v+1 ) {
      h[u][v] = UNKNOWN;
  }}
  return  mem( i, j, h );
}

public static long mem( int i, int j, long[][] h ) {

  if ( h[i][j] == UNKNOWN ) {

    if ( i+j < 2 ) {
      h[i][j] = i+j;
    } else if ( j == 0 ) {
      h[i][j] = mem( 1, i-2, h ) + mem( 0, i-1, h );
    } else if ( j == 1 ) {
      h[i][j] = mem( 0, i, h ) + mem( i+1, 0, h );
    } else {
      h[i][j] = mem( i+2, j-2, h ) + mem( i+1, j-1, h );
  }}
  return  h[i][j];
}

public static final int UNKNOWN = -1;
```

3. Correttezza dei programmi iterativi

Il seguente metodo statico in Java calcola il valore numerico di un intero rappresentato in *notazione ternaria bilanciata*. L'argomento è un array di cifre, espresse dai rispettivi valori –1, 0 e +1, dove la cifra più significativa è la componente di indice 0. Il risultato restituito è il valore del numero intero, cioè: $btr[0]\cdot 3^{n-1} + btr[1]\cdot 3^{n-2} + \ldots + btr[n-1]\cdot 3^{0}$.
Completa il programma introducento opportune asserzioni, specificamente: precondizioni, postcondizioni e invarianti del comando iterativo; proponi inoltre una funzione di terminazione relativa al ciclo. Dimostra quindi che l'invariante vale all'inizio del ciclo e si conserva ad ogni passo iterativo.

```
public static int value( int[] btr ) {

  // Pre:    ∀ k ∈ [1, btr.length] . btr[k] ∈ { –1, 0, +1 }

  int  v = 0,  i = 0,  n = btr.length;
  while ( i < n ) {

    // Inv:   v = Σ_{k∈[0,i–1]} btr[k] · 3^{i–k–1}
    //        i ≤ n

    // Term:  n – i

    v = 3 * v + btr[i];  i = i + 1;
  }
  return  v;

  // Post:   v = Σ_{k∈[0,n–1]} btr[k] · 3^{n–k–1}
  //
}
```

Dimostrazione che l'invariante vale all'inizio del ciclo:

$$0 = \sum_{k \in \varnothing} btr[k] \cdot 3^{-k-1}$$

$0 \le n = btr.length$

Dimostrazione che l'invariante si conserva ad ogni passo iterativo:

$$3v + btr[i] = 3 \cdot \sum_{k\in[0,i-1]} btr[k] \cdot 3^{i-k-1} + btr[i] \cdot 3^{i+1-i-1}$$ *perché l'invariante vale prima del passo iterativo*

$$= \sum_{k\in[0,i-1]} btr[k] \cdot 3^{i+1-k-1} + btr[i] \cdot 3^{i+1-i-1}$$

$$= \sum_{k\in[0,i+1-1]} btr[k] \cdot 3^{i+1-k-1}$$

$i+1 \le n$ *per la condizione del while*

4. Oggetti in Java

In relazione alla "conta" ispirata a un racconto di *Giuseppe Flavio*, supponi che la configurazione dei cavalieri attorno alla tavola sia rappresentata in Java da una classe `RoundTable`, per le cui istanze è definito il seguente protocollo:

`RoundTable( int n )`	costruttore: crea la configurazione iniziale della tavola con *n* cavalieri
`int numberOfKnights()`	restituisce il numero di cavalieri rimasti attorno al tavolo
`int leadingKnight()`	restituisce il numero che identifica il cavaliere che deve servire il sidro
`int leavingKnight()`	restituisce il numero che identifica il cavaliere che sta per lasciare la tavola
`void afterNextExit()`	modifica la configurazione della tavola, determinando l'uscita di un cavaliere e il passaggio di mano della brocca di sidro

Utilizzando il protocollo sopra specificato (senza implementarlo), definisci in Java un metodo

`public static int[] exitOrder( int n )`

che, dato il numero iniziale di cavalieri, restituisce l'array dei numeri che li identificano, ordinati secondo l'ordine in cui essi lasciano la tavola con la coppa riempita di sidro. Per esempio, `exitOrder(12)` restituisce l'array:

`{ 2 4 6 8 10 12 3 7 11 5 1 }`

```
public static int[] exitOrder( int n ) {

  RoundTable table = new RoundTable( n );

  int[] knights = new int[ n-1 ];
  int i = 0;

  while ( table.numberOfKnights() > 1 ) {

    knights[ i ] = table.leavingKnight();
    table.afterNextExit();
    i = i + 1;
  }
  return  knights;
}
```