Corso di Programmazione

Esame del 27 Gennaio 2012

cognome e nome

Risolvi i seguenti esercizi, riporta le soluzioni in modo chiaro negli appositi spazi e giustifica sinteticamente le risposte. Dovrai poi consegnare queste schede con le soluzioni, avendo cura di scrivere il tuo nome nell'intestazione e su ciascun eventuale foglio aggiuntivo.

1. Programmazione in Java

Considera il modello della scacchiera realizzato dalla classe `Board` per affrontare il rompicapo delle *N regine*. Per le istanze di `Board` è definito il protocollo richiamato sotto. Utilizzando esclusivamente il protocollo specificato, definisci in Java un metodo statico che, data un'istanza di `Board`, restituisce in forma testuale (`String`) la griglia delle posizioni minacciate o meno dalle regine disposte sulla scacchiera. Ogni cella della scacchiera è rappresentata nella griglia da tre caratteri: una coppia di parentesi quadre aperta e chiusa che racchiudono una '`x`', se la casella è minacciata da qualche regina, oppure uno spazio bianco, se invece non lo è. A ciascuna riga della scacchiera corrisponde una riga di testo (la sequenza di controllo `"\n"` denota un "a capo" all'interno di una stringa).

Il testo riportato qui a fianco illustra la griglia relativa alla disposizione di quattro regine nelle caselle *b2*, *b5*, *c8* e *f1* di una scacchiera *8* x *8*. In alto si vede la stinga restituita dal metodo `arrangement()`. Convenzionalmente, le righe sono numerate dal basso verso l'alto; le lettere che identificano le colonne sono ordinate da sinistra verso destra. Protocollo di `Board`:

```
[  c8  b5  b2  f1  ]

[x][x][x][x][x][x][x][x]
[ ][x][x][x][ ][x][x][ ]
[x][x][x][ ][x][x][ ][ ]
[x][x][x][x][x][x][x][x]
[x][x][x][x][ ][x][x][ ]
[x][x][x][x][ ][x][ ][x]
[x][x][x][x][x][x][x][x]
[x][x][x][x][x][x][x][x]
```

```
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()
```

2. Dati procedurali

Una sequenza di *k+1* numeri e *k* operazioni aritmetiche rappresenta un'espressione RPN (Reverse Polish Notation) se ogni sottosequenza propria iniziale (comprendente, cioè, i primi *j* elementi, con $1 \le j \le 2k$) contiene più numeri che operazioni. Per valutare un'espressione RPN si utilizza uno stack di numeri, inizialmente vuoto; procedendo nell'ordine della sequenza, i numeri vengono "caricati" in cima allo stack e le operazioni aritmetiche vengono applicate ai due elementi inseriti per ultimi nello stack, sostituendoli in questa struttura con il risultato dell'operazione. Nel caso di operazioni non commutative, più precisamente, il numero in cima allo stack svolge il ruolo di operando destro. Alla fine lo stack contiene il risultato. Per esempio, l'espressione RPN `4 3 12 + * 40 - 5 +` viene valutata come segue:

espressione	*stack (cima a sinistra)*	*espressione*	*stack (cima a sinistra)*
`4 3 12 + * 40 - 5 +`	`< >`	`4 3 12 + * 40 - 5 +`	`< 60 >`
`4 3 12 + * 40 - 5 +`	`< 4 >`	`4 3 12 + * 40 - 5 +`	`< 40 60 >`
`4 3 12 + * 40 - 5 +`	`< 3 4 >`	`4 3 12 + * 40 - 5 +`	`< 20 >`
`4 3 12 + * 40 - 5 +`	`< 12 3 4 >`	`4 3 12 + * 40 - 5 +`	`< 5 20 >`
`4 3 12 + * 40 - 5 +`	`< 15 4 >`	`4 3 12 + * 40 - 5 +`	`< **25** >`

Un'espressione RPN può essere rappresentata in Scheme da una lista i cui elementi (eterogenei) sono *numeri* e *procedure*. Per consentirne la valutazione, è stata progettata la procedura `rpn-eval`. Esempi di applicazione:

```
(rpn-eval (list 5)) → 5                    (rpn-eval (list 120 3 12 + 4 * /)) → 2

(rpn-eval (list 12 3 -)) → 9               (rpn-eval (list 4 3 12 + * 40 - 5 +)) → 25
```

Completa il programma riportato qui sotto, che realizza `rpn-eval`, introducendo il codice Scheme appropriato negli spazi indicati a tratto punteggiato. A tal fine, per verificare se un dato è numerico o procedurale puoi utilizzare i predicati predefiniti `number?` e/o `procedure?`.

```
(define rpn-eval
  (lambda (exp)
    (rpn-eval-tr exp ..... )
    ))

(define rpn-eval-tr
  (lambda (exp stk)
    (cond ((null? exp) (car stk))

          ( .....
           (rpn-eval-tr (cdr exp) (cons (car exp) stk)))
          (else

           ( .....

             .....

             ..... ))

          )))
```

3. Programmi 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)
              ))
          )))
```

Facendo riferimento alla procedura `paths` definita nella pagina precedente, determina il risultato della valutazione di ciascuna delle espressioni riportate qui sotto:

`(paths 2 2 0)`	→	.....	`(paths 0 5 0)`	→	.....
`(paths 2 2 1)`	→	.....	`(paths 3 4 2)`	→	.....
`(paths 2 2 3)`	→	.....	`(paths 4 4 2)`	→	.....

4. Memoization

Applica la tecnica di *memoization* per realizzare una versione più efficiente del programma dell'esercizio **3**.

5. Verifica formale della correttezza

```
(define g
  (lambda (p q k)              ; p, q, k > 0 interi
    (let ((r (* (- p 1) q)))
      (if (= k 1)
          r
          (+ (* p (g p q (- k 1))) r)
          ))))
```

In relazione alla procedura definita sopra è possibile dimostrare che per tutte le terne di valori interi positivi x, y, z:

$$(g\ x\ y\ z) \quad \rightarrow \quad y(x^z - 1)$$

Dimostra per induzione questa proprietà; in particolare:

- Formalizza la proprietà che esprime il caso / i casi base:

- Formalizza l'ipotesi induttiva:

- Formalizza la proprietà da dimostrare come passo induttivo:

- Dimostra il caso / i casi base:

- Dimostra il passo induttivo: