Corso di Programmazione
II Accertamento del 15 Marzo 2005 / A

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 che si renda necessario.

1. Procedure in Scheme

Con riferimento alla procedura *h* così definita:

```
(define h
  (lambda (r x s)  ; r procedura, s lista
    (cond ((null? s)      0)
          ((r x (car s))  0)
          (else  (+ (h r x (cdr s)) 1))
    ) ))
```

calcola i risultati della valutazione di ciascuna delle seguenti espressioni Scheme:

(h <= 3 '(1 2 3 4 5)) $\longrightarrow$ 2 *(h char<=? #\c (list #\a #\b #\c #\d #\e))* $\longrightarrow$ 2

(h <= 8 '(1 2 3 4 5)) $\longrightarrow$ 5 *(h string<=? "bob" (list "alice" "chris"))* $\longrightarrow$ 1

Generalizza inoltre il risultato della valutazione dell'espressione seguente per $m \geq 0$ e $n > 0$:

(h <= m '(1 2 ... n)) $\longrightarrow$ 0 se $m = 0$; $m - 1$ se $1 \leq m \leq n$; n se $m > n$

2. Procedure in Scheme

Considera alberi binari di ricerca (BST) i cui nodi hanno valori interi e per i quali sono definite le operazioni: *(empty-tree)* per costruire un albero vuoto; *(empty? T)* per verificare se l'albero *T* è vuoto; *(root-node T)* che assume come valore il nodo radice di un albero *T* non vuoto; *(left T)*, *(right T)* per determinare il sottoalbero sinistro e quello destro relativamente a un albero *T* non vuoto; *(grow-tree N L R)* che assume come valore l'albero di radice *N* e con sottoalberi sinistro *L* e destro *R*.
Completa la definizione della procedura *add* che, dati un intero *x* e un albero binario di ricerca *bst*, assuma come valore un nuovo albero binario di ricerca con tutti i nodi di *bst* più un ulteriore nodo di valore *x*. Gli alberi binari di ricerca considerati qui hanno la seguente proprietà: preso un qualunque nodo di valore *n*, tutti i nodi del relativo sottoalbero sinistro hanno valore *maggiore* di *n* e tutti i nodi del sottoalbero destro hanno valore *minore* di *n*. Assumendo che l'argomento *bst* abbia questa proprietà, si richiede che anche il valore di *add* la soddisfi.

```
(define add
 (lambda (x bst)  ; x intero, bst non vuoto
   (cond ((empty? bst) (grow-tree x   (empty-tree) (empty-tree)          ))
         ((= x (root-node bst)) bst)
         ((< x   (root-node bst)                                 )
          (grow-tree   (root-node bst) (left bst) (add x (right bst))          ))
         (else
          (grow-tree   (root-node bst) (add x (left bst)) (right bst)          ))
         ) ))
```

3. Definizione di procedure in Scheme

Il problema proposto si ispira a una costruzione di Steinhaus basata sull'insieme binario **B** = $\{+1, -1\}$: data una sequenza di elementi di **B** di lunghezza n, si generano sequenze di lunghezza $n-1$, $n-2$, ..., 2, 1, dove la sequenza di lunghezza $k-1$ è ricavata dalla sequenza *S* di lunghezza k calcolando nell'ordine tutti i prodotti delle coppie di elementi consecutivi in *S*, come illustrato dall'esempio sotto.

```
-1  -1  -1  +1  +1  -1
  +1  +1  -1  +1  -1
    +1  -1  -1  -1
      -1  +1  +1
        -1  +1
          -1
```

Derfinisci formalmente un programma in Scheme che, data una sequenza di elementi di **B**, permetta di conoscere il valore ($+1$ o -1) dell'unico elemento contenuto nella sequenza finale di lunghezza 1.

```
(define steinhaus
  (lambda (b) ; |b| > 0
    (if (null? (cdr b))
        (car b)
        (steinhaus (next b))
        )
    ))


(define next
  (lambda (r)
    (if (null? (cdr r))
        null
        (cons (* (car r) (cadr r)) (next (cdr r)))
        )
    ))
```

4. Utilizzo di strutture dati

Questo esercizio fa riferimento alla soluzione del problema delle N regine, di cui si riporta qui di seguito il programma discusso durante il corso per calcolare il numero di soluzioni:

```
(define queens-arrangements
  (lambda (n)
    (queens-completions (empty-board n))
    ))
(define queens-completions
  (lambda (board)                    ; scacchiera completa?
    (if (= (assigned-rows board) (board-size board))
        1                            ; si: e' stata trovata una soluzione
        (next-row-trials 1 board)    ; no: considera la riga successiva
    ) ))
(define next-row-trials
  (lambda (c board)                  ; sulla riga successiva
    (let ((depth-completions         ; a partire da colonna c
           (if (safe-next? board c)  ; prima: tentativo in posizione c
               (queens-completions (add-next-queen board c))
               0)
          ))
      (if (< c (board-size board))   ; quindi: colonne successive
          (+ depth-completions (next-row-trials (+ c 1) board))
          depth-completions
     )) ))
```

Supponi che sia disponibile una procedura *queen-in?* tale che *(queen-in? (arrangement B) i j)* assuma come valore vero o falso a seconda che nella scacchiera *B* ci sia o meno una regina collocata esattamente nella *j*-ima colonna dell' *i*-ima riga, dove *arrangement* fa parte del consueto protocollo relativo ai dati astratti di tipo scacchiera.
Introduci due ulteriori parametri di *queens-arrangements* per rappresentare le coordinate *i*, *j* e apporta le modifiche che si rendono necessarie, salvaguardando la struttura del programma, affinché il valore di *(queens-arrangements n i j)* sia la lista delle soluzioni di dimensione *n* in cui una regina si trova nella posizione di coordinate *i*, *j*. A tal fine, numera le righe che intendi modificare nel codice formalizzato sopra e riporta i numeri e le corrispondenti modifiche nel riquadro.

```
(define queens-arrangements
  (lambda (n i j)  ; i, j in [1,n]
    (queens-completions (empty-board n) i j)
    ))
(define queens-completions
  (lambda (board i j)
    (if (= (assigned-rows board) (board-size board))
        (if (queen-in? (arrangement board) i j)
            (list (arrangement board))
            null
            )
        (next-row-trials 1 board i j)
        )
    ))
(define next-row-trials
  (lambda (c board i j)              ; sulla riga successiva
    (let ((depth-completions         ; a partire da colonna c
           (if (safe-next? board c)  ; prima: tentativo in posizione c
              (queens-completions (add-next-queen board c) i j)
              null)
          ))
      (if (< c (board-size board))   ; quindi: colonne successive
         (append depth-completions (next-row-trials (+ c 1) board i j))
         depth-completions
          ))
    ))
```

5. Definizione di procedure in Scheme

Definisci la procedura *queen-in?* introdotta nell'esercizio precedente. Come nell'esempio discusso durante il corso, assumi che *A = (arrangement B) =* (c_n ... c_2 c_1) rappresenti una disposizione di n regine nella scacchiera, dove c_i si riferisce alla posizione (colonna) occupata dalla regina collocata nella riga *i*-ima; si vuole allora che *(queen-in? A i j)* assuma come valore vero se e solo se una regina è collocata esattamente nella *j*-ima colonna dell' *i*-ima riga.

```
(define queen-in?
  (lambda (arr i j)  ; 1 <= i <= |arr|
    (if (< i (length arr))
        (queen-in? (cdr arr) i j)
        (= j (car arr))
        )
    ))
```

Corso di Programmazione
II Accertamento del 15 Marzo 2005 / B

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 che si renda necessario.

1. Procedure in Scheme

Con riferimento alla procedura *h* così definita:

```
(define h
  (lambda (r x s)  ; r procedura, s lista
    (cond ((null? s)      0)
          ((r x (car s))  0)
          (else  (+ (h r x (cdr s)) 1))
    ) ))
```

calcola i risultati della valutazione di ciascuna delle seguenti espressioni Scheme:

(h >= 2 '(5 4 3 2 1)) $\longrightarrow$ 3 *(h char>=? #\b (list #\e #\d #\c #\b #\a))* $\longrightarrow$ 3

(h >= 1 '(5 4 3 2 1)) $\longrightarrow$ 4 *(h string>=? "alice" (list "chris" "bob"))* $\longrightarrow$ 2

Generalizza inoltre il risultato della valutazione dell'espressione seguente per $m \geq 0$ e $n > 0$:

(h >= m '(n ... 2 1)) $\longrightarrow$ 0 se $m > n$; $n - m$ se $0 \leq m \leq n$

2. Procedure in Scheme

Considera alberi binari di ricerca (BST) i cui nodi hanno valori interi e per i quali sono definite le operazioni: *(empty-tree)* per costruire un albero vuoto; *(empty? T)* per verificare se l'albero *T* è vuoto; *(root-node T)* che assume come valore il nodo radicedi un albero *T* non vuoto; *(left T)*, *(right T)* per determinare il sottoalbero sinistro e quello destro relativamente a un albero *T* non vuoto; *(grow-tree N L R)* che assume come valore l'albero di radice *N* e con sottoalberi sinistro *L* e destro *R*.
Completa la definizione della procedura *add* che, dati un intero *x* e un albero binario di ricerca *bst*, assuma come valore un nuovo albero binario di ricerca con tutti i nodi di *bst* più un ulteriore nodo di valore *x*. Gli alberi binari di ricerca considerati qui hanno la seguente proprietà: preso un qualunque nodo di valore *n*, tutti i nodi del relativo sottoalbero sinistro hanno valore *minore* di *n* e tutti i nodi del sottoalbero destro hanno valore *maggiore* di *n*. Assumendo che l'argomento *bst* abbia questa proprietà, si richiede che anche il valore di *add* la soddisfi.

```
(define add
 (lambda (x bst)  ; x intero, bst non vuoto
   (cond ((empty? bst) (grow-tree x   (empty-tree) (empty-tree)          ))
         ((< x (root-node bst))
          (grow-tree   (root-node bst) (add x (left bst)) (right bst)          ))
         ((> x   (root-node bst)                                  )
          (grow-tree   (root-node bst) (left bst) (add x (right bst))          ))
         (else bst)
         ) ))
```

3. Definizione di procedure in Scheme

Il problema proposto si ispira a una costruzione di Steinhaus basata sull'insieme binario **B** = $\{0, 1\}$: data una riga di elementi di **B** di lunghezza n, si generano righe di lunghezza $n-1$, $n-2$, ..., 2, 1, dove la riga di lunghezza $k-1$ è ricavata dalla riga *R* di lunghezza k considerando nell'ordine tutte le coppie di elementi consecutivi in *R* e introducendo 0 se i due elementi sono diversi oppure 1 se sono uguali, come illustrato dall'esempio sotto.

```
0   0   0   1   1   0
  1   1   0   1   0
    1   0   0   0
      0   1   1
        0   1
          0
```

Derfinisci formalmente un programma in Scheme che, data una riga di elementi di **B**, permetta di conoscere il valore (0 o 1) dell'unico elemento contenuto nella riga unitaria finale.

```
(define steinhaus
  (lambda (b) ; |b| > 0
    (if (null? (cdr b))
        (car b)
        (steinhaus (next b))
        )
    ))


(define next
  (lambda (r)
    (if (null? (cdr r))
        null
        (cons (if (equal? (car r) (cadr r)) 1 0) (next (cdr r)))
        )
    ))
```

4. Utilizzo di strutture dati

Questo esercizio fa riferimento alla soluzione del problema delle N regine, di cui si riporta qui di seguito il programma discusso durante il corso per calcolare il numero di soluzioni:

```
(define queens-arrangements
  (lambda (n)
    (queens-completions (empty-board n))
    ))
(define queens-completions
  (lambda (board)                    ; scacchiera completa?
    (if (= (assigned-rows board) (board-size board))
        1                            ; si: e' stata trovata una soluzione
        (next-row-trials 1 board)    ; no: considera la riga successiva
    ) ))
(define next-row-trials
  (lambda (c board)                  ; sulla riga successiva
    (let ((depth-completions         ; a partire da colonna c
           (if (safe-next? board c)  ; prima: tentativo in posizione c
               (queens-completions (add-next-queen board c))
               0)
          ))
      (if (< c (board-size board))   ; quindi: colonne successive
          (+ depth-completions (next-row-trials (+ c 1) board))
          depth-completions
     )) ))
```

Supponi che sia disponibile una procedura *queen-in-main-diag?* tale che *(queen-in-main-diag? (arrangement B))* assuma come valore vero o falso a seconda che nella scacchiera *B* ci sia o meno una regina collocata in un quadrato della diagonale principale, dove *arrangement* fa parte del consueto protocollo relativo ai dati astratti di tipo scacchiera.
Apporta le modifiche che ritieni necessarie, salvaguardando la struttura del programma, affinché il valore restitutito da *(queens-arrangements n)* sia la lista delle soluzioni del problema di dimensione *n* in cui una regina si trova in un quadrato della diagonale principale. A tal fine, numera le righe che intendi modificare nel codice formalizzato sopra e riporta i numeri e le corrispondenti modifiche nell'apposito spazio.

```
(define queens-arrangements
  (lambda (n)
    (queens-completions (empty-board n))
    ))
(define queens-completions
  (lambda (board)
    (if (= (assigned-rows board) (board-size board))
        (if (queen-in-main-diag? (arrangement board))
            (list (arrangement board))
            null
            )
        (next-row-trials 1 board)
        )
    ))
(define next-row-trials
  (lambda (c board)                   ; sulla riga successiva
    (let ((depth-completions          ; a partire da colonna c
           (if (safe-next? board c)   ; prima: tentativo in posizione c
              (queens-completions (add-next-queen board c))
              null)
          ))
      (if (< c (board-size board))    ; quindi: colonne successive
         (append depth-completions (next-row-trials (+ c 1) board))
         depth-completions
          ))
    ))
```

5. Definizione di procedure in Scheme

Definisci la procedura *queen-in-main-diag?* introdotta nell'esercizio precedente. Come nell'esempio discusso durante il corso, assumi che *A = (arrangement B) =* $(c_n \ldots c_2\ c_1)$ rappresenti una disposizione di n regine nella scacchiera, dove c_i si riferisce alla posizione (colonna) occupata dalla regina collocata nella riga *i*-ima; *(queen-in-main-diag? A)* deve assumere il valore vero se e solo se una regina è collocata nella diagonale principale (cioè la diagonale costituita dal primo quadrato della prima riga, dal secondo quadrato della seconda riga, ..., dall'ultimo quadrato dell'ultima riga).

```
(define queen-in-main-diag?
  (lambda (arr)
    (cond ((null? arr)                 #f)
          ((= (car arr) (length arr))  #t)
          (else (queen-in-main-diag? (cdr arr)))
          )
    ))
```