Corso di Programmazione

II Accertamento del 28 Marzo 2008 / 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. Valutazioni Scheme

Considera il seguente programma in Scheme:

```
(define val
  (lambda (rep)
    (val-tr rep 0)
    ))
```

```
(define val-tr
  (lambda (r n)
    (cond ((null? r) n)
          ((equal? (car r) '-) (val-tr (cdr r) (- (* 3 n) 1)))
          ((equal? (car r) '+) (val-tr (cdr r) (+ (* 3 n) 1)))
          ((equal? (car r) ':) (val-tr (cdr r)    (* 3 n)   ))
          )))
```

Completa le seguenti valutazioni di espressioni, riportando il risultato o l'argomento nell'apposito spazio:

```
(val '(+ -))                                  ⟶        2
(val '(- + :))                                ⟶       -6
(val '(+ - + :))                              ⟶       21
(val   '(- - : + :)                 )         ⟶      -105
```

2. Strutture dati

Un albero *AVL* è un albero binario *bilanciato* in base a questo criterio: qualsiasi nodo si consideri, l'altezza dei relativi sottoalberi sinistro e destro differisce al più di uno. Completa il seguente programma per verificare se un albero binario è *AVL*, assumendo che per gli alberi binari siano definite le operazioni `empty-tree?`, per conoscere se l'albero è vuoto, `left-subtree` e `right-subtree`, per determinare i sottoalberi sinistro e destro rispetto alla radice. La procedura `avl?` verifica se l'argomento è un albero *AVL*; la procedura `avl-balancement` restituisce una coppia di valori: la profondità dell'albero e la massima differenza di altezza fra due sottoalberi relativi a uno stesso nodo.

```
(define avl?
  (lambda (tree)
    (<=   (cadr (avl-balancement tree))                                  1)
    ))
```

```
(define avl-balancement
  (lambda (t)
    (if (empty-tree? t)
        (list 0 0)
        (let ((lb (avl-balancement (left-subtree  t)))
              (rb (avl-balancement (right-subtree t)))
              )
          (list (+   (max (car lb) (car rb))                             1)
                (max (cadr lb)  (cadr rb)   (abs (-   (car lb) (car rb)   )))
                ))
        )))
```

3. Definizione di procedure in Scheme

Definisci una procedura ***digit-list*** che, dati il numeratore *n* e il denominatore *d* di una frazione *n/d*, con $0 \le n < d$, e dato un numero naturale *k*, restituisce la lista delle prime *k* cifre dell'espansione decimale di *n/d* (successive a *"0."*).
Per esempio:

```
(digit-list 1 8 5)   ⟶   (1 2 5 0 0)
```

```
(define digit-list
  (lambda (n d k)   ; 0 <= n < d
    (if (= k 0)
        null
        (cons (quotient (* 10 n) d)
              (digit-list (remainder (* 10 n) d) d (- k 1)))
        )))
```

4. Definizione di procedure in Scheme

Definisci una procedura ***compute*** che, data una lista $(n_0, n_1, ..., n_k)$ di $k+1$ numeri interi e data una lista $(o_1, o_2, ..., o_k)$ di *k* operazioni aritmetiche (ciascuna definita per coppie di interi e a valori interi), restituisce il risultato *r* del seguente processo di calcolo: $r_1 = o_1(n_0,n_1)$; $r_2 = o_2(r_1,n_2)$; $r_3 = o_3(r_2,n_3)$; . . . $r = r_k = o_k(r_{k-1},n_k)$.
Per esempio:

```
(compute '(5 5 2 9) (list * + /))   ⟶   3
```

```
(define compute
  (lambda (numlist oplist)
    (if (null? oplist)
        (car numlist)
        (compute
         (cons ((car oplist) (car numlist) (cadr numlist)) (cddr numlist))
         (cdr oplist))
        )))
```

5. Astrazione sui dati

Considera la seconda realizzazione del dato astratto "tavola rotonda" (problema dei commensali), riportata qui sotto:

```
(define round-table
  (lambda (n)
    (cons (subrange 1 n) null)
    ))

(define last-player?
  (lambda (table)
    (null? (cdar table))
    ))

(define current-player caar)

(define next-table
  (lambda (table)
    (move-item
     (caar table) (cddar table) (cdr table))
    ))

(define subrange
  (lambda (inf sup)
    (if (> inf sup)
        null
        (cons inf (subrange (+ inf 1) sup))
        )
    ))
```

```
(define move-item
  (lambda (itm lft rgt)
    (cond
      ((null? lft)
       (cons
         (reverse-items rgt (cons itm null))
         null)
         )
      ((null? (cdr lft))
       (cons
         (cons
           (car lft)
           (reverse-items rgt (cons itm null)) )
         null)
         )
      (else (cons lft (cons itm rgt)))
    )))

(define reverse-items
  (lambda (src dst)
    (if (null? src)
        dst
        (reverse-items (cdr src)
                       (cons (car src) dst))
    )))
```

Una stima del numero *C(n)* di operazioni `cons` necessarie per completare il gioco a partire da una tavola con *n* commensali è *C(n) < 3n*, come si è visto in classe (non contando quelle dovute all'applicazione iniziale del costruttore `round-table`). Se si vuole calcolare il valore esatto di *C(n)*, si può modificare leggermente la rappresentazione della configurazione generica in modo da includere il numero *c* di `cons` effettuati per raggiungere la configurazione stessa. In particolare, al posto di una coppia di liste *(L . R)* si può utilizzare la struttura *((L . R) c)*. Come di consueto, *L* è la lista ordinata in verso orario dei primi commensali, a partire da chi ha la moka, *R* è la lista dei rimanenti commensali in ordine rovesciato (rispetto al verso orario); si suppone inoltre che *L* non possa essere vuota se *R* non lo è. All'inizio *c = 0* e il corpo del costruttore `round-table` diventa `(list (cons (subrange 1 n) null) 0)`.

Proponi le opportune integrazioni anche per le altre procedure, fra quelle riportate sopra, salvaguardando la compatibilità con il programma che utilizza il dato astratto. A tale proposito, numera le righe di codice che intendi modificare e riporta numeri e corrispondenti modifiche nello spazio sottostante.

```
(define round-table
  (lambda (n)
    (list (cons (subrange 1 n) null) 0)
    ))


(define last-player?
  (lambda (table)
    (null? (cdaar table))
    ))

(define current-player caaar)

(define next-table
  (lambda (table)
    (move-item
     (caaar table) (cddar (car table)) (cdar table) (cadr table))
    ))


(define move-item
  (lambda (itm lft rgt n)
    (cond
      ((null? lft)
       (list
        (cons (reverse-items rgt (cons itm null)) null)
        (+ n (length rgt) 2)
        ))
      ((null? (cdr lft))
       (list
        (cons (cons (car lft) (reverse-items rgt (cons itm null))) null)
        (+ n (length rgt) 3)
        ))
      (else
       (list
        (cons lft (cons itm rgt))
        (+ n 2)
        ))
    )))
```

Corso di Programmazione

II Accertamento del 28 Marzo 2008 / 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. Valutazioni Scheme

Considera il seguente programma in Scheme:

```
(define val
  (lambda (rep)
    (val-tr rep 0)
    ))
```

```
(define val-tr
  (lambda (r n)
    (cond ((null? r) n)
          ((equal? (car r) '-) (val-tr (cdr r) (- (* 3 n) 1)))
          ((equal? (car r) '+) (val-tr (cdr r) (+ (* 3 n) 1)))
          ((equal? (car r) ':) (val-tr (cdr r)    (* 3 n)   ))
          )))
```

Completa le seguenti valutazioni di espressioni, riportando il risultato o l'argomento nell'apposito spazio:

```
(val '(+ +))                                  ⟶   ____4____
(val '(- : +))                                ⟶   ____-8____
(val '(- : + -))                              ⟶   ___-25____
(val  __'(+ - - : +)_____ )                   ⟶       46
```

2. Strutture dati

Un albero *AVL* è un albero binario *bilanciato* in base a questo criterio: qualsiasi nodo si consideri, l'altezza dei relativi sottoalberi sinistro e destro differisce al più di uno. Completa il seguente programma per verificare se un albero binario è *AVL*, assumendo che per gli alberi binari siano definite le operazioni `empty-tree?`, per conoscere se l'albero è vuoto, `left-subtree` e `right-subtree`, per determinare i sottoalberi sinistro e destro rispetto alla radice. La procedura `avl?` verifica se l'argomento è un albero *AVL*; la procedura `avl-balancement` restituisce una coppia di valori: la massima differenza di altezza fra due sottoalberi relativi a uno stesso nodo e la profondità dell'albero.

```
(define avl?
  (lambda (tree)
    (<=  __(car (avl-balancement tree))_____________________________  1)
    ))
```

```
(define avl-balancement
  (lambda (t)
    (if (empty-tree? t)
        (list 0 0)
        (let ((lb (avl-balancement (left-subtree  t)))
              (rb (avl-balancement (right-subtree t)))
              )
          (list (max  _(car lb)_  (car rb) (abs (-  _(cadr lb) (cadr rb)_ )))
                (+  __(max (cadr lb) (cadr rb))_____________________  1)
                ))
        )))
```

3. Definizione di procedure in Scheme

Definisci una procedura ***digit-list*** che, dati il numeratore *n* e il denominatore *d* di una frazione *n/d*, con $0 \le n < d$, e dato un numero naturale *k*, restituisce la lista delle prime *k* cifre dell'espansione binaria di *n/d* (successive a *"0."*).
Per esempio:

```
(digit-list 3 8 5)   ⟶   (0 1 1 0 0)
```

```
(define digit-list
  (lambda (n d k)   ; 0 <= n < d
    (if (= k 0)
        null
        (cons (quotient (* 2 n) d)
              (digit-list (remainder (* 2 n) d) d (- k 1)))
        )))
```

4. Definizione di procedure in Scheme

Definisci una procedura ***evaluate*** che, data una lista $(o_1, o_2, ..., o_k)$ di *k* operazioni aritmetiche (ciascuna definita per coppie di interi e a valori interi) e data una lista $(n_0, n_1, ..., n_k)$ di $k+1$ numeri interi, restituisce il risultato *r* del seguente processo di calcolo: $r_1 = o_1(n_0,n_1)$; $r_2 = o_2(r_1,n_2)$; $r_3 = o_3(r_2,n_3)$; . . . $r = r_k = o_k(r_{k-1},n_k)$.
Per esempio:

```
(evaluate (list * / -) '(6 7 2 7))   ⟶   14
```

```
(define evaluate
  (lambda (oplist numlist)
    (if (null? oplist)
        (car numlist)
        (evaluate
         (cdr oplist)
         (cons ((car oplist) (car numlist) (cadr numlist)) (cddr numlist)))
        )))
```

5. Astrazione sui dati

Considera la seconda realizzazione del dato astratto "tavola rotonda" (problema dei commensali), riportata qui sotto:

```
(define round-table
  (lambda (n)
    (cons (subrange 1 n) null)
    ))

(define last-player?
  (lambda (table)
    (null? (cdar table))
    ))

(define current-player caar)

(define next-table
  (lambda (table)
    (move-item
     (caar table) (cddar table) (cdr table))
    ))

(define subrange
  (lambda (inf sup)
    (if (> inf sup)
        null
        (cons inf (subrange (+ inf 1) sup))
        )
    ))
```

```
(define move-item
  (lambda (itm lft rgt)
    (cond
      ((null? lft)
       (cons
         (reverse-items rgt (cons itm null))
         null)
         )
      ((null? (cdr lft))
       (cons
         (cons
           (car lft)
           (reverse-items rgt (cons itm null)) )
         null)
         )
      (else (cons lft (cons itm rgt)))
    )))

(define reverse-items
  (lambda (src dst)
    (if (null? src)
        dst
        (reverse-items (cdr src)
                       (cons (car src) dst))
    )))
```

Una stima del numero *C(n)* di operazioni `cons` necessarie per completare il gioco a partire da una tavola con *n* commensali è *C(n) < 3n*, come si è visto in classe (non contando quelle dovute all'applicazione iniziale del costruttore `round-table`). Se si vuole calcolare il valore esatto di *C(n)*, si può modificare leggermente la rappresentazione della configurazione generica in modo da includere il numero *c* di `cons` effettuati per raggiungere la configurazione stessa. In particolare, al posto di una coppia di liste *(L . R)* si può utilizzare la struttura *(c . (L . R))*. Come di consueto, *L* è la lista ordinata in verso orario dei primi commensali, a partire da chi ha la moka, *R* è la lista dei rimanenti commensali in ordine rovesciato (rispetto al verso orario); si suppone inoltre che *L* non possa essere vuota se *R* non lo è. All'inizio *c = 0* e il corpo del costruttore `round-table` diventa `(cons 0 (cons (subrange 1 n) null))`.

Proponi le opportune integrazioni anche per le altre procedure, fra quelle riportate sopra, salvaguardando la compatibilità con il programma che utilizza il dato astratto. A tale proposito, numera le righe di codice che intendi modificare e riporta numeri e corrispondenti modifiche nello spazio sottostante.

```
(define round-table
  (lambda (n)
    (cons 0 (cons (subrange 1 n) null))
    ))


(define last-player?
  (lambda (table)
    (null? (cdadr table))
    ))

(define current-player caadr)

(define next-table
  (lambda (table)
    (move-item
     (caadr table) (cddar (cdr table)) (cddr table) (car table))
    ))


(define move-item
  (lambda (itm lft rgt n)
    (cond
      ((null? lft)
       (cons
        (+ n (length rgt) 2)
        (cons (reverse-items rgt (cons itm null)) null)
        ))
      ((null? (cdr lft))
       (cons
        (+ n (length rgt) 3)
        (cons (cons (car lft) (reverse-items rgt (cons itm null))) null)
        ))
      (else
       (cons
        (+ n 2)
        (cons lft (cons itm rgt))
        ))
    )))
```