Corso di Programmazione

II Accertamento del 27 Marzo 2007 / 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

Considera la seguente definizione di procedura in Scheme, dove `number?` verifica se l'argomento è numerico, `number->string` converte un numero nella corrispondente stringa e `symbol->string` converte un simbolo atomico nella corrispondente stringa (p.es. `'atom` in `"atom"`):

```
(define process
  (lambda (e s)
    (cond ((null? e)
           (car s))
          ((number? (car e))
           (process (cdr e) (cons (number->string (car e)) s)))
          (else
           (process (cdr e) (cons (string-append "(" (symbol->string (car e))
                                                 " " (cadr s) " " (car s) ")")
                                  (cddr s))) )
          )))
```

Riporta i risultati della valutazione di ciascuna delle seguenti espressioni:

```
(process '(15) '())                    →   "15"

(process '(12 5 *) '())                →   "(* 12 5)"

(process '(12 5 * 24 4 / /) '())       →   "(/ (* 12 5) (/ 24 4))"

(process '(18 15 3 / 2 3 * + -) '())   →   "(- 18 (+ (/ 15 3) (* 2 3)))"
```

2. Procedure in Scheme

Completa la seguente procedura che può essere applicata per generare liste di numeri primi. Più precisamente, valutando l'espressione `(sieve 2` *n* `(lambda (q) #t) null)` si vuole ottenere la lista dei numeri primi compresi fra *2* e *n*. L'algoritmo si basa sul fatto che un numero è primo se e solo se non è divisibile per alcuno dei precedenti numeri primi; tale condizione è verificata dal predicato `is-prime?` passato come parametro procedurale.

```
(define sieve

  (lambda (inf sup is-prime? primes)

    (cond ((> inf sup) (reverse primes))

          ((is-prime? inf)

           (sieve (+ inf 1) sup

                  (lambda (q) (if (= (remainder q inf) 0) #f (is-prime? q)))

                  (cons inf primes)) )

          (else (sieve (+ inf 1) sup is-prime? primes))

          )))
```

3. Definizione di procedure in Scheme

Definisci una procedura *av* in Scheme che, data una lista non vuota $(x_1\ x_2\ ...\ x_n)$ i cui n elementi x_i appartengono all'insieme $\{-1, 0, 1\}$, restituisca la lista $(y_1\ y_2\ ...\ y_{n-1})$ di $n-1$ elementi dello stesso insieme tale che $y_i = -1$ se $x_i + x_{i+1} < 0$, $y_i = 0$ se $x_i + x_{i+1} = 0$ e $y_i = 1$ se $x_i + x_{i+1} > 0$. Per esempio:

```
(av '(0 0 -1 -1 1 0 0 1 0))      →      (0 -1 -1 0 1 0 1 1)
```

```
(define av
  (lambda (sq)
    (cond ((null? sq) null)
          ((null? (cdr sq)) null)
          ((< (+ (car sq) (cadr sq)) 0) (cons -1 (av (cdr sq))))
          ((> (+ (car sq) (cadr sq)) 0) (cons  1 (av (cdr sq))))
          (else (cons 0 (av (cdr sq))))
          )))
```

4. Strutture dati

Un albero di Huffman è un albero binario non vuoto le cui foglie sono etichettate con simboli atomici e i cui restanti nodi non sono etichettati e hanno sempre esattamente due figli. Una possibile rappresentazione in Scheme è definita come segue: (i) un albero con un solo nodo è rappresentato dal simbolo—quotato in Scheme—che etichetta quel nodo; (ii) l'albero costruito collegando a una radice non etichettata due sottoalberi di Huffman L e R è rappresentato dalla coppia $(L\ .\ R)$. La lunghezza (= numero di bit) della codifica di un simbolo basata su un albero di Huffman è pari alla lunghezza (= numero di archi) del percorso che connette la radice dell'albero alla corrispondente foglia.

Scrivi un programma in Scheme che, dato un albero di Huffman H, calcola la coppia $(S\ .\ C)$, dove S è la lista di tutti i simboli che etichettano foglie di H e C è la lista delle lunghezze delle corrispondenti codifiche, nello stesso ordine.

```
(define code-sizes
  (lambda (ht)
    (code-depths ht 0)
    ))

(define code-depths
  (lambda (ht d)
    (if (symbol? ht)
        (cons (list ht) (list d))
        (let ((lft (code-depths (car ht) (+ d 1)))
              (rgt (code-depths (cdr ht) (+ d 1))))
          (cons (append (car lft) (car rgt)) (append (cdr lft) (cdr rgt))) )
    )))
```

5. Astrazione sui dati

Considera la seconda realizzazione del dato astratto “tavola rotonda” (problema dei commensali) discussa in classe e riportata qui di seguito:

```
(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))
    )))
```

Si desidera modificare leggermente la rappresentazione della configurazione generica, utilizzando al posto di una coppia di liste una lista *(c L R)* di tre elementi, rispettivamente: il commensale *c* con la moka, la lista ordinata *L* dei primi commensali che seguono *c* in senso orario, la lista *R* dei rimanenti commensali in ordine rovesciato (rispetto al senso orario). Si suppone inoltre che *L* non possa essere vuota se *R* non lo è. In particolare, il corpo del costruttore `round-table` diventa `(list 1 (subrange 2 n) null)`.

Apporta le opportune modifiche anche alle altre procedure, fra quelle riportate qui, salvaguardando la compatibilità con il programma che utilizza il dato astratto e garantendo che la soluzione aggiornata resti efficiente. 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 1 (subrange 2 n) null)
    ))


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


(define current-player car)


(define next-table
  (lambda (table)
    (move-item
     (car table) (cdadr table) (caddr table))
    ))

;; subrange come sopra
```

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


;; reverse-items come sopra
```

Corso di Programmazione

II Accertamento del 27 Marzo 2007 / 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

Considera la seguente definizione di procedura in Scheme, dove `number?` verifica se l'argomento è numerico, `number->string` converte un numero nella corrispondente stringa e `symbol->string` converte un simbolo atomico nella corrispondente stringa (p.es. `'atom` in `"atom"`):

```
(define process
  (lambda (e s)
    (cond ((null? e)
           (car s))
          ((number? (car e))
           (process (cdr e) (cons (number->string (car e)) s)))
          (else
           (process (cdr e) (cons (string-append "(" (cadr s) " "
                                                 (symbol->string (car e)) " " (car s) ")")
                                  (cddr s))) )
          )))
```

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

```
(process '(21) '())                    →   "21"

(process '(16 3 *) '())                →   "(16 * 3)"

(process '(16 3 * 12 5 * +) '())       →   "((16 * 3) + (12 * 5))"

(process '(21 15 3 - 12 4 - + *) '())  →   "(21 * ((15 - 3) + (12 - 4)))"
```

2. Procedure in Scheme

Completa la seguente procedura che si applica per generare liste di numeri primi. Più precisamente, valutando l'espressione `(sieve 2` *n* `(lambda (q) #f) null)` si vuole ottenere la lista dei numeri primi compresi fra *2* e *n*. L'algoritmo si basa sul fatto che un numero è primo se e solo se non è divisibile per alcuno dei precedenti numeri primi; la condizione di avere divisori (primi) è verificata dal predicato `has-divisors?` passato come parametro procedurale.

```
(define sieve

  (lambda (inf sup has-divisors? primes)

    (cond ((> inf sup) (reverse primes))

          ((has-divisors? inf) (sieve (+ inf 1) sup has-divisors? primes))

          (else

           (sieve (+ inf 1) sup

                  (lambda (q) (if (= (remainder q inf) 0) #t (has-divisors? q)))

                  (cons inf primes)) )

          )))
```

3. Definizione di procedure in Scheme

Definisci una procedura *df* in Scheme che, data una lista non vuota $(x_1\ x_2\ ...\ x_n)$ i cui n elementi x_i appartengono all'insieme $\{-1, 0, 1\}$, restituisca la lista $(y_1\ y_2\ ...\ y_{n-1})$ di $n-1$ elementi dello stesso insieme tale che $y_i = 1$ se $x_i < x_{i+1}$, $y_i = 0$ se $x_i = x_{i+1}$ e $y_i = -1$ se $x_i > x_{i+1}$. Per esempio:

```
(df '(0 0 -1 -1 1 0 0 1 0))      →      (0 -1 0 1 -1 0 1 -1)
```

```
(define df
  (lambda (sq)
    (cond ((null? sq) null)
          ((null? (cdr sq)) null)
          ((< (car sq) (cadr sq)) (cons  1 (df (cdr sq))))
          ((> (car sq) (cadr sq)) (cons -1 (df (cdr sq))))
          (else (cons 0 (df (cdr sq))))
          )))
```

4. Strutture dati

Un albero di Huffman è un albero binario non vuoto le cui foglie sono etichettate con simboli atomici e i cui restanti nodi non sono etichettati e hanno sempre esattamente due figli. Una possibile rappresentazione in Scheme è definita come segue: (i) un albero con un solo nodo è rappresentato dal simbolo—quotato in Scheme—che etichetta quel nodo; (ii) l'albero costruito collegando a una radice non etichettata due sottoalberi di Huffman L e R è rappresentato dalla coppia $(L\ .\ R)$. La lunghezza (= numero di bit) della codifica di un simbolo basata su un albero di Huffman è pari alla lunghezza (= numero di archi) del percorso che connette la radice dell'albero alla corrispondente foglia.

Scrivi un programma in Scheme che, dato un albero di Huffman H, calcola la lista delle coppie $(c\ .\ s)$, per tutti i simboli s che etichettano foglie di H, dove c è la lunghezza della codifica di s.

```
(define code-sizes
  (lambda (ht)
    (code-depths ht 0)
    ))

(define code-depths
  (lambda (ht d)
    (if (symbol? ht)
        (list (cons d ht))
        (append (code-depths (car ht) (+ d 1))
                (code-depths (cdr ht) (+ d 1)))
    )))
```

5. Astrazione sui dati

Considera la seconda realizzazione del dato astratto “tavola rotonda” (problema dei commensali) discussa in classe e riportata qui di seguito:

```
(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))
    )))
```

Si desidera modificare la rappresentazione della configurazione generica separando tre elementi: il commensale *c* con la moka, la lista ordinata *L* dei primi commensali che seguono *c* in senso orario, la lista *R* dei rimanenti commensali in ordine rovesciato (rispetto al senso orario). Questa terna viene codificata utilizzando, al posto di una coppia di liste, la lista *(R c L)* degli elementi identificati sopra. Si suppone inoltre che *L* possa essere vuota solo se anche *R* lo è. In particolare, il corpo del costruttore `round-table` diventa `(list null 1 (subrange 2 n))`.

Apporta le opportune modifiche anche alle altre procedure, fra quelle riportate qui, salvaguardando la compatibilità con il programma che utilizza il dato astratto e garantendo che la soluzione aggiornata resti efficiente. 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 null 1 (subrange 2 n))
    ))


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


(define current-player cadr)


(define next-table
  (lambda (table)
    (move-item
     (cadr table) (cdaddr table) (car table))
    ))


;; subrange come sopra
```

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


;; reverse-items come sopra
```