Corso di Programmazione

I Accertamento del 1 Febbraio 2010

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. Definizione di procedure in Scheme

Derfinisci in Scheme una procedura `swap` che, data una lista *s* e un indice intero $k \geq 0$, restituisce la lista che si ottiene da *s* scambiando l'ordine degli elementi di posizione *k* e *k+1* (e solo di questi due elementi). L'indice della prima posizione della lista è *0*; inoltre, se *k* o *k+1* non sono indici validi, allora `swap` restituisce la lista *s* immutata. Esempi:

```
(swap '(A B C D E F G H) 0)     →   '(B A C D E F G H)
(swap '(A B C D E F G H) 3)     →   '(A B C E D F G H)
(swap '(A B C D E F G H) 6)     →   '(A B C D E F H G)
(swap '(A B C D E F G H) 7)     →   '(A B C D E F G H)
```

```
(define swap
  (lambda (s k)
    (cond ((< (length s) 2) s)
          ((= k 0) (cons (cadr s) (cons (car s) (cddr s))))
          (else (cons (car s) (swap (cdr s) (- k 1))))
          )))
```

2. Procedure in Scheme

Con riferimento alla procedura `q` così definita:

```
(define q
  (lambda (x)
    (let ((n (string-length x)))
      (if (= n 0)
          "+"
          (let ((y (string-ref x (- n 1))))
            (if (char=? y #\+)
                (string-append (q (substring x 0 (- n 1))) "-")
                (string-append (substring x 0 (- n 1)) (if (char=? y #\-) "." "+"))
                ))
          ))))
```

calcola il risultato della valutazione di ciascuna delle seguenti espressioni Scheme:

`(q "-")` → `"."` `(q "-.")` → `"-+"`

`(q ".")` → `"+"` `(q "+-+")` → `"+.-"`

`(q "+")` → `"+-"` `(q "+++")` → `"+---"`

`(q "--")` → `"-."` `(q "-+++")` → `".---"`

3. Procedure con valori procedurali

Data la lista `args` degli arogmenti e la lista `vals` dei corrispondenti valori, dove le due liste hanno la stessa lunghezza e gli elementi di `args` sono tutti diversi fra loro, l'oggetto restituito dalla procedura `tab-fun` rappresenta la funzione che applicata al *k*-imo elemento di `args` assume come valore il *k*-imo elemento di `vals`. Tale funzione è definita solo per argomenti che appartengono alla lista `args`. Per esempio, sulla base della definizione

```
(define f (tab-fun '(0 1 2 3 4 5) '(2 3 5 7 11 13)))
```

ne risultano le seguenti valutazioni:

`(f 0)` → `2` `(f 3)` → `7`

`(f 1)` → `3` `(f 5)` → `13`

Completa la definizione della procedura `tab-fun` riportata qui sotto, introducendo il codice appropriato negli spazi indicati a tratto punteggiato.

```
(define tab-fun        ; valore: procedura
  (lambda (args vals)  ; args, vals: liste di uguale lunghezza
    (  lambda (x)
      (if (=   x                    (car args))
          (car vals)
           ((tab-fun (cdr args) (cdr vals)) x)
          ))
     ))
```

4. Verifica formale della correttezza

```
(define range-sum  ; valore: intero
  (lambda (a b)    ; a, b: interi tali che 0 <= a <= b
    (if (= a b)
        a
        (let ((k (quotient (+ a b) 2)))
          (+ (range-sum a k) (range-sum (+ k 1) b))
          ))
    ))
```

In relazione alla procedura definita sopra è possibile dimostrare che per tutte le coppie di interi m, n con $0 \le m \le n$:

(range-sum m n) → $(m+n)\cdot(n-m+1) / 2$

Dimostra questa proprietà, procedendo per induzione sul valore della "distanza" $n-m$ fra i parametri; in particolare:

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

  Per tutte le coppie di interi m, $n \ge 0$ tali che $n-m = 0$: (range-sum m n) → $(m+n)\cdot(n-m+1) / 2$

- Formalizza l'ipotesi induttiva: Fissata una "distanza" intera $d > 0$,

  per tutte le coppie di interi m, $n \ge 0$ tali che $n-m < d$: (range-sum m n) → $(m+n)\cdot(n-m+1) / 2$

- Formalizza la proprietà da dimostrare come passo induttivo: Per d fissato sopra,

  per tutte le coppie di interi m, $n \ge 0$ tali che $n-m = d$: (range-sum m n) → $(m+n)\cdot(n-m+1) / 2$

- Dimostra il caso / i casi base:

  Poiché $n = m$, (range-sum m n) → $m = 2m\cdot 1 / 2 = (m+n)\cdot(n-m+1) / 2$

- Dimostra il passo induttivo: Sia $q = \lfloor d/2 \rfloor$:

  (range-sum m n) → (let ((k (quotient (+ m n) 2))) ...) *poiché* $n > m$

  → (let ((k $m+q$)) ...) *poiché* $n = m+d$

  → (+ (range-sum m $m+q$) (range-sum (+ $m+q$ 1) n))

  → (+ (range-sum m $m+q$) (range-sum $m+q+1$ n))

  → (+ $(2m+q)\cdot(q+1) / 2$ $(m+n+q+1)\cdot(n-m-q) / 2$) *per l'ipotesi induttiva*
  poiché $q,\ d-q-1 < d$

  → $((m+n-d+q)\cdot(q+1) / 2 + (m+n+q+1)\cdot(n-m-q) / 2$ *poiché* $m = n-d$

  $= ((m+n)\cdot(q+1) + (q-d)\cdot(q+1) + (m+n)\cdot(n-m-q) + (q+1)\cdot(n-m-q)) / 2$

  $= ((m+n)\cdot(n-m+1) + (n-m-d)\cdot(q+1)) / 2 = ((m+n)\cdot(n-m+1) + 0\cdot(q+1)) / 2$

  $= (m+n)\cdot(n-m+1) / 2$

5. Programmi in Scheme

Definisci in Scheme una procedura `word-list` che, dato un testo, restituisce la lista delle parole che lo compongono. Il testo e ciascuna parola sono rappresentati da stringhe. Più precisamente, si intende spezzare il testo in corrispondenza ai caratteri bianchi (spazi), che delimitano le parole senza farvi parte. Anche eventuali caratteri non alfabetici vengono raggruppati in un unica parola se non sono separati da spazi bianchi. Esempi:

```
(word-list "the cat cathces the mouse")   →   '("the" "cat" "cathces" "the" "mouse")
(word-list "  hello     world   ")         →   '("hello" "world")
(word-list "bye bye.")                     →   '("bye" "bye.")
(word-list "hi!")                          →   '("hi!")
```

```
(define word-list
  (lambda (txt)
    (process-words 0 txt)
    ))

(define process-words
  (lambda (k txt)
    (cond ((= k (string-length txt))
           (if (= k 0)
               null
               (cons txt null)
               ))
          ((char=? (string-ref txt k) #\space)
           (if (= k 0)
               (process-words 0 (substring txt 1))
               (cons (substring txt 0 k)
                     (process-words 0 (substring txt (+ k 1))))
               ))
          (else
           (process-words (+ k 1) txt)
           )
          )))
```