Corso di Programmazione

I Accertamento del 24 Gennaio 2011

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

Definisci in Scheme una procedura `sorted-ins` che, data una parola *w*, cioè una stringa di lettere minuscole, e data una lista di parole *s* ordinata in ordine alfabetico, restituisce la lista che si ottiene da *s* introducendo *w* nella posizione corretta affinché l'ordine alfabetico venga rispettato. A tal fine puoi utilizzare le procedure predefinite per confrontare l'ordine alfabetico/lessicografico delle stringhe: `string<?`, `string<=?`, `string>?`, `string>=?`. Esempi:

```
(sorted-ins "giacinto" '("begonia" "gardenia" "giglio" "viola"))
                              →  '("begonia" "gardenia" "giacinto" "giglio" "viola")
(sorted-ins "viola" '("begonia" "gardenia" "giacinto" "violaciocca"))
                         →  '("begonia" "gardenia" "giacinto" "viola" "violaciocca")
```

```
(define sorted-ins
  (lambda (w s)
    (cond ((null? s) (list w))
          ((string<=? w (car s)) (cons w s))
          (else (cons (car s) (sorted-ins w (cdr s))))
          )))
```

2. Procedure in Scheme

Con riferimento alla procedura `q` così definita:

```
(define q
  (lambda (x u)
    (let ((k (- (string-length x) 1))
          (f (lambda (z)
               (integer->char (+ (char->integer z) (if (char=? z #\0) u 0) -1))
               )))
      (let ((v (string-ref x k)) (y (substring x 0 k)))
        (string-append
         (cond ((char>? v #\0) y) ((string=? y "1") "") (else (q y u)))
         (string (f v)))
        ))))
```

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

`(q "1" 4)`	→	`"0"`	`(q "11" 4)`	→	`"10"`
`(q "2" 4)`	→	`"1"`	`(q "30" 4)`	→	`"23"`
`(q "3" 4)`	→	`"2"`	`(q "200" 8)`	→	`"177"`
`(q "10" 4)`	→	`"3"`	`(q "1000" 10)`	→	`"999"`

3. Procedure con valori procedurali

Date due stringhe `plain` ed `encrypted`, la procedura `permutation-rule` restituisce una funzione *f* che rappresenta una regola per crittare un testo sostituendone o permutandone i caratteri in base alle posizioni in cui compaiono nelle stringhe. Più precisamente, `plain` rappresenta l'alfabeto del testo in chiaro e contiene caratteri tutti diversi fra loro; `encrypted` ha la stessa lunghezza di `plain` e ciascuno dei suoi caratteri rappresenta la codifica, nel testo crittato, del carattere che ha la stessa posizione in `plain`. La funzione restituita ha come dominio l'insieme dei caratteri di `plain` e come codominio l'insieme dei caratteri di `encrypted`, entrambi di tipo *char*. Per esempio, sulla base della definizione

```
(define pr (permutation-rule "ABCDEFGHILMNOPQRSTVX" "DEFGHILMNOPQRSTVXABC"))
```

ne risultano le seguenti valutazioni (codice di *Giulio Cesare* applicato all'alfabeto *Latino*):

`(pr #\C)` → `#\F`	`(pr #\A)` → `#\D`	`(pr #\E)` → `#\H`
`(pr #\S)` → `#\X`	`(pr #\V)` → `#\B`	`(pr #\R)` → `#\V`

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

```
(define permutation-rule
  (lambda (plain encrypted)

    (   lambda (x)        (permutation-rec    x   0          plain encrypted))
    ))


(define permutation-rec
  (lambda (x i plain encrypted)

    (if (char=?   x   (string-ref plain i)                                    )
        (string-ref encrypted i)

         (permutation-rec x (+ i 1) plain encrypted)
        )))
```

4. Verifica formale della correttezza

```
(define magic          ; valore intero
  (lambda (x)          ; x > 0  intero
    (cond ((= x 1) 1)
          ((even? x)
           (- (* 2 (magic (quotient x 2))) 1))
          (else
           (+ (* 2 (magic (quotient x 2))) 3))
          )))
```

In relazione alla procedura definita sopra è possibile dimostrare che per tutti i valori interi positivi di k:

$$\texttt{(magic } 2^k-1\texttt{)} \quad \rightarrow \quad 2^{k+1}-3$$

Dimostra per induzione questa proprietà; in particolare:

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

  $\texttt{(magic } 2^1-1\texttt{)} \quad \rightarrow \quad 2^2-3$

- Formalizza l'ipotesi induttiva: Fissato un intero $n > 0$,

  $\texttt{(magic } 2^n-1\texttt{)} \quad \rightarrow \quad 2^{n+1}-3$

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

  $\texttt{(magic } 2^{n+1}-1\texttt{)} \quad \rightarrow \quad 2^{n+2}-3$

- Dimostra il caso / i casi base:

  $\texttt{(magic } 1\texttt{)} \quad \rightarrow \quad 1$

- Dimostra il passo induttivo:

  $\texttt{(magic } 2^{n+1}-1\texttt{)} \quad \rightarrow \quad \texttt{(+ (* 2 (magic (quotient } 2^{n+1}-1 \texttt{ 2))) 3)} \quad \texttt{;;} \quad 2^{n+1}-1$ *dispari*

  $\rightarrow \quad \texttt{(+ (* 2 (magic } 2^n-1\texttt{)) 3)}$

  $\rightarrow \quad \texttt{(+ (* 2 } 2^{n+1}-3\texttt{) 3)} \quad \texttt{;;}$ *ipotesi induttiva*

  $\rightarrow \quad 2^{n+2}-6+3$

5. Programmi in Scheme

Definisci in Scheme una procedura `sort` che, data una lista di stringhe binarie, restituisce la lista composta dalle stesse stringhe ordinate per valore numerico rappresentato crescente. Si assume che ciascun elemento della lista passata come argomento sia una stringa corretta nella notazione binaria, specificamente: i soli caratteri utilizzati sono `0`, `1`, e la prima cifra è `1` con l'unica eccezione della rappresentazione del numero *zero*. Per esempio:

```
(sort '("101" "1011" "10" "111" "10000" "1110" "1" "100" "0" "110" "11"))
             →   '("0" "1" "10" "11" "100" "101" "110" "111" "1011" "1110" "10000")
```

```
(define sort
  (lambda (s)
    (if (null? s)
        null
        (add (car s) (sort (cdr s)))
        )))


(define add
  (lambda (n s)
    (cond ((null? s) (list n))
          ((leq? n (car s)) (cons n s))
          (else (cons (car s) (add n (cdr s))))
          )))


(define leq?
  (lambda (u v)
    (let ((m (string-length u)) (x (string-ref u 0))
          (n (string-length v)) (y (string-ref v 0)))
      (cond ((< m n) #t)
            ((> m n) #f)
            ((char=? x y) (if (= m 1) #t (leq? (substring u 1) (substring v 1))))
            (else (char=? y #\1))
            ))))
```