## Corso di Programmazione

**I Accertamento del 2 Febbraio 2009**

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 a valori booleani `prime-factors?` che, dati un numero intero positivo $n$ e una lista di numeri primi $p$, verifica se gli elementi di $p$ rappresentano la fattorizzazione di $n$ in numeri primi. In particolare, nella lista $p$ ciascun fattore primo deve comparire tante volte quanto è il relativo esponente nella fattorizzazione di $n$. Si assume comunque che tutti gli elementi di $p$ siano effettivamente numeri primi, ordinati in ordine non decrescente. Per esempio:

```
(prime-factors? 1 '())                →    #t
(prime-factors? 8 '(2 2 2))           →    #t
(prime-factors? 12 '(2 2 3))          →    #t
(prime-factors? 15 '(3 5))            →    #t
(prime-factors? 180 '(2 2 3 3 5))     →    #t
(prime-factors? 12 '(2 3))            →    #f
(prime-factors? 15 '(3 5 5))          →    #f
```

```
(define prime-factors?
  (lambda (n pfs)
    (cond ((null? pfs)
           (= n 1))
          ((= (remainder n (car pfs)) 0)
           (prime-factors? (/ n (car pfs)) (cdr pfs)))
          (else
           #f)
          )))
```

**2. Procedure in Scheme**

Con riferimento alla procedura `balanced-str` così definita:

```
(define balanced-str
  (lambda (lv)
    (if (= lv 0)
        ""
        (let ((q? (even? lv)) (pn (balanced-str (- lv 1))))
          (string-append (if q? "[" "(") pn pn (if q? "]" ")"))
          ))))
```

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

```
(balanced-str 0)    →    ""
(balanced-str 1)    →    "()"
(balanced-str 2)    →    "[()()]"
(balanced-str 3)    →    "([()()][()()])"
(balanced-str 4)    →    "[([()()][()()])([()()][()()])]"
```

**3. Procedure con argomenti procedurali**

Considera il seguente programma in Scheme:

```
(define nxt
  (lambda (sq op)
    (if (null? (cdr sq))
        null
        (cons (op (car sq) (cadr sq))
              (nxt (cdr sq) op))
        )))
```

```
(define tri
  (lambda (sq op)
    (if (null? (cdr sq))
        sq
        (cons sq
              (tri (nxt sq op) op))
        )))
```

Assumi che il primo argomento della procedura `tri` sia una sequenza binaria, cioè una lista di `0` e `1`, non vuota. Quale espressione definisce il secondo argomento in modo tale che il valore restituito dalla procedura sia il *triangolo di Steinhaus* costruito a partire dalla sequenza binaria data? Formalizza un'opportuna espressione per il secondo argomento completando l'applicazione di `tri` riportata come esempio nel riquadro.

```
(tri '(1 1 0 1)   (lambda (u v) (if (= u v) 0 1))   )

        →   '((1 1 0 1) (0 1 1) (1 0) 1)
```

## 4. Verifica formale della correttezza

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

In relazione alla procedura in Scheme definita sopra è possibile dimostrare che:

$$\forall\, k \in \mathbf{N}^+ .\ \texttt{(ufo}\ \ 3 \cdot 2^{k-1}\texttt{)} \ \rightarrow\ 2^k + 1$$

Imposta e sviluppa la dimostrazione per induzione di questa proprietà; in particolare:

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

  $\texttt{(ufo}\ \ 3 \cdot 2^0\texttt{)} \ \rightarrow\ 2^1 + 1$

- Formalizza l'ipotesi induttiva: fissato $n \in \mathbf{N}^+$

  $\texttt{(ufo}\ \ 3 \cdot 2^{n-1}\texttt{)} \ \rightarrow\ 2^n + 1$

- Formalizza la proprietà da dimostrare come passo induttivo: per $n$ fissato sopra

  $\texttt{(ufo}\ \ 3 \cdot 2^n\texttt{)} \ \rightarrow\ 2^{n+1} + 1$

- Dimostra il/i caso/i base:

  $\texttt{(ufo}\ \ 3\texttt{)} \ \rightarrow\ \texttt{(+ (* 2 (ufo (quotient}\ 3\ \texttt{2))) 1)}$

  $\rightarrow\ \texttt{(+ (* 2 (ufo}\ 1\texttt{)) 1)} \ \rightarrow\ \texttt{(+ (* 2}\ 1\texttt{) 1)} \ \rightarrow\ 3$

- Dimostra il passo induttivo:

  $\texttt{(ufo}\ \ 3 \cdot 2^n\texttt{)} \ \rightarrow\ \texttt{(- (* 2 (ufo (quotient}\ 3 \cdot 2^n\ \texttt{2))) 1)}$ ; $3 \cdot 2^n$ pari poiché $n \in \mathbf{N}^+$

  $\rightarrow\ \texttt{(- (* 2 (ufo}\ 3 \cdot 2^{n-1}\texttt{)) 1)} \ \rightarrow\ \texttt{(- (* 2}\ (2^n+1)\texttt{) 1)}$ ; per l'ipotesi induttiva

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

**5. Astrazione sui dati**

In relazione al *problema di Giuseppe Flavio*, supponi che lo stato del gioco sia accessibile *esclusivamente* attraverso le seguenti operazioni, che hai ha disposizione ma di cui non conosci l'implementazione:

```
(round-table <commensali>)  :  interi -> tavole
(last-player? <tavola>)     :  tavole -> booleani
(exiting-player <tavola>)   :  tavole -> nomi
(next-table <tavola>)       :  tavole -> tavole
```

Rispetto al protocollo (semplificato) considerato a lezione, la procedura `current-player` è sostituita dalla procedura `exiting-player` che, data una configurazione della *tavola rotonda* con almeno due *commensali*, restituisce il *nome* (etichetta numerica) del commensale che sta per essere servito e quindi sarà il prossimo ad *uscire*. Le altre procedure hanno la stessa funzione di quelle discusse a lezione: `round-table` restituisce lo stato iniziale del gioco per un dato numero di commensali; `last-player?` verifica se è rimasto in tavola solo l'ultimo commensale; infine, `next-table` applica le regole del gioco e restituisce la configurazione della tavola rotonda che si ottiene da quella data come argomento in seguito all'uscita di un commensale e al passaggio di mano della *moka*.

Utilizzando questo protocollo, definisci in Scheme una procedura a valori booleani

`(josephus-test` $n$ $k$`)`

che, dati il numero iniziale $n$ dei commensali e il nome $k$ di uno di essi, con $1 \leq k \leq n$, restituisce `#t` se il $k$-imo commensale resterà a tavola per ultimo quando tutti gli altri saranno usciti, `#f` altrimenti.

```
(define josephus-test      ; valore: boolean
  (lambda (players label)  ; players, label: interi
    (choice-test (round-table players) label)
    ))

(define choice-test
  (lambda (table label)
    (cond ((last-player? table)
           #t)
          ((equal? (exiting-player table) label)
           #f)
          (else
           (choice-test (next-table table) label))
          )))
```