# Corso di Programmazione

**I Accertamento del 27 Gennaio 2012**

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

Un numero *x* compreso nell'intervallo *[0, 1[* (cioè tale che $0 \leq x < 1$) può essere codificato, eventualmente in forma approssimata, tramite una sequenza finita di cifre binarie, precisamente quelle che comparirebbero a destra del punto che separa la parte intera dalla parte frazionaria nella rappresentazione in base 2. Definisci una procedura `fract-part` che, data una stringa `f` di `0`/`1`, restituisce il valore numerico *x* di `f` interpretata come parte frazionaria in base 2. Esempi:

```
(fract-part "0") → 0       (fract-part "01") → 0.25     (fract-part "101") → 0.625
(fract-part "1") → 0.5     (fract-part "11") → 0.75     (fract-part "0101") → 0.3125
```

```
(define fract-part
  (lambda (bits)
    (if (string=? bits "")
        0
        (/ (+ (if (char=? (string-ref bits 0) #\1) 1 0)
              (fract-part (substring bits 1)))
           2)
        )))
```

### 2. Programmi in Scheme

Facendo riferimento alla procedura `paths` definita nella pagina seguente, determina il risultato della valutazione di ciascuna delle espressioni riportate qui sotto:

```
(paths 2 2 0)   →   0          (paths 0 5 0)   →   1
(paths 2 2 1)   →   2          (paths 3 4 2)   →   7
(paths 2 2 3)   →   6          (paths 4 4 2)   →   8
```

```
(define paths
  (lambda (i j k)
    (if (or (= i 0) (= j 0))
        1
        (+ (down (- i 1) j k) (right i (- j 1) k))
        )))

(define down
  (lambda (i j k)
    (cond ((and (= k 0) (> j 0)) 0)
          ((or (= i 0) (= j 0)) 1)
          (else
           (+ (down (- i 1) j k)
              (right i (- j 1) (- k 1))
              ))
          )))
```

```
(define right
  (lambda (i j k)
    (cond ((and (= k 0) (> i 0)) 0)
          ((or (= i 0) (= j 0)) 1)
          (else
           (+ (down (- i 1) j (- k 1))
              (right i (- j 1) k)
              ))
          )))
```

**3. Dati procedurali**

Una sequenza di *k+1* numeri e *k* operazioni aritmetiche rappresenta un'espressione RPN (Reverse Polish Notation) se ogni sottosequenza propria iniziale (comprendente, cioè, i primi *j* elementi, con $1 \le j \le 2k$) contiene più numeri che operazioni. Per valutare un'espressione RPN si utilizza uno stack di numeri, inizialmente vuoto; procedendo nell'ordine della sequenza, i numeri vengono "caricati" in cima allo stack e le operazioni aritmetiche vengono applicate ai due elementi inseriti per ultimi nello stack, sostituendoli in questa struttura con il risultato dell'operazione. Nel caso di operazioni non commutative, più precisamente, il numero in cima allo stack svolge il ruolo di operando destro. Alla fine lo stack contiene il risultato. Per esempio, l'espressione RPN `4 3 12 + * 40 - 5 +` viene valutata come segue:

| *espressione* | *stack (cima a sinistra)* | *espressione* | *stack (cima a sinistra)* |
|---|---|---|---|
| `4 3 12 + * 40 - 5 +` | `< >` | `4 3 12 + * 40 - 5 +` | `< 60 >` |
| `4 3 12 + * 40 - 5 +` | `< 4 >` | `4 3 12 + * 40 - 5 +` | `< 40 60 >` |
| `4 3 12 + * 40 - 5 +` | `< 3 4 >` | `4 3 12 + * 40 - 5 +` | `< 20 >` |
| `4 3 12 + * 40 - 5 +` | `< 12 3 4 >` | `4 3 12 + * 40 - 5 +` | `< 5 20 >` |
| `4 3 12 + * 40 - 5 +` | `< 15 4 >` | `4 3 12 + * 40 - 5 +` | `< **25** >` |

Un'espressione RPN può essere rappresentata in Scheme da una lista i cui elementi (eterogenei) sono *numeri* e *procedure*. Per consentirne la valutazione, è stata progettata la procedura `rpn-eval`. Esempi di applicazione:

`(rpn-eval (list 5))` → `5`

`(rpn-eval (list 12 3 -))` → `9`

`(rpn-eval (list 120 3 12 + 4 * /))` → `2`

`(rpn-eval (list 4 3 12 + * 40 - 5 +))` → `25`

Completa il programma riportato qui sotto, che realizza `rpn-eval`, introducendo il codice Scheme appropriato negli spazi indicati a tratto punteggiato. A tal fine, per verificare se un dato è numerico o procedurale puoi utilizzare i predicati predefiniti `number?` e/o `procedure?`.

```
(define rpn-eval
  (lambda (exp)
    (rpn-eval-tr exp   null                                                    )
    ))

(define rpn-eval-tr
  (lambda (exp stk)
    (cond ((null? exp) (car stk))

          (  (number? (car exp))
           (rpn-eval-tr (cdr exp) (cons (car exp) stk)))
          (else

           (  rpn-eval-tr
              (cdr exp)
              (cons  ((car exp) (cadr stk) (car stk))  (cddr stk))          ))
          )))
```

**4. Verifica formale della correttezza**

```
(define g
  (lambda (p q k)              ; p, q, k > 0 interi
    (let ((r (* (- p 1) q)))
      (if (= k 1)
          r
          (+ (* p (g p q (- k 1))) r)
          ))))
```

In relazione alla procedura definita sopra è possibile dimostrare che per tutte le terne di valori interi positivi $x, y, z$:

$$(g\ x\ y\ z) \quad \rightarrow \quad y(x^z-1)$$

Dimostra per induzione questa proprietà; in particolare:

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

  Per tutte le coppie di interi $x, y > 0$ : $(g\ x\ y\ 1) \rightarrow y(x^1-1)$

- Formalizza l'ipotesi induttiva: Preso un intero $u > 0$

  Per tutte le coppie di interi $x, y > 0$ : $(g\ x\ y\ u) \rightarrow y(x^u-1)$

- Formalizza la proprietà da dimostrare come passo induttivo: Per $u$ scelto sopra

  Per tutte le coppie di interi $x, y > 0$ : $(g\ x\ y\ u+1) \rightarrow y(x^{u+1}-1)$

- Dimostra il caso / i casi base:

  $(g\ x\ y\ 1) \rightarrow r \rightarrow y(x-1) = y(x^1-1)$

- Dimostra il passo induttivo:

  $(g\ x\ y\ u+1)$

  $\rightarrow$ $(+\ (*\ x\ (g\ x\ y\ (-\ u+1\ 1)))\ r)$ poiché $u+1 > 1$, dove $r$ è associato a $y(x-1)$

  $\rightarrow$ $(+\ (*\ x\ (g\ x\ y\ u))\ r)$

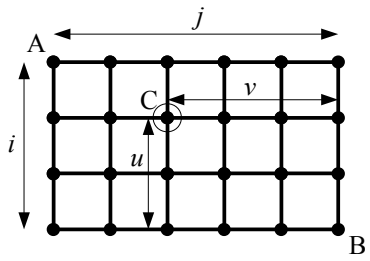
  $\rightarrow$ $(+\ (*\ x\ \ y(x^u-1)\ )\ r)$ per l'ipotesi induttiva

  $\rightarrow$ $(+\ \ xy(x^u-1)\ \ y(x-1)\ )$ poiché $r$ è associato a $y(x-1)$

  $\rightarrow$ $y(x^{u+1}-x+x-1) = y(x^{u+1}-1)$

**5. Programmi in Scheme**

Considera una variante del problema dei percorsi di Manhattan in cui si vuole conoscere quanti itinerari di lunghezza minima consentono di spostarsi da A a B *evitando* C. Formuliamo il probelma più precisamente. Per raggiungere B da A è necessario effettuare *i* passi in basso e *j* passi a destra, intercalati in qualsiasi modo; inoltre, B si trova *u* passi in basso e *v* passi a destra di C. Quanti percorsi diversi di lunghezza *i*+*j* collegano A a B senza passare per C?




Per esempio, nella situazione illustrata in figura, dove *i=3*, *j=5*, *u=2* e *v=3*, ci sono *26* percorsi diversi che soddisfano alle condizioni poste.

Definisci una procedura `xpaths` che risolve il problema proposto. Esempi:

```
(xpaths 2 2 0 0)   →   0          (xpaths 2 2 1 1)   →   2
(xpaths 2 2 0 1)   →   3          (xpaths 2 2 2 2)   →   0
(xpaths 2 2 0 2)   →   5          (xpaths 3 5 2 3)   →   26
```

```
(define xpaths
  (lambda (i j u v)
    (cond ((and (= i u) (= j v))
           0)
          ((and (= i 0) (= j 0))
           1)
          ((= i 0)
           (xpaths i (- j 1) u v))
          ((= j 0)
           (xpaths (- i 1) j u v))
          (else
           (+ (xpaths (- i 1) j u v)
              (xpaths i (- j 1) u v)
           )))
    ))
```

*oppure:*

```
(define xpaths
  (lambda (i j u v)
    (let ((all (manhattan i j)))
      (if (or (< i u) (< j v))
          all
          (- all (* (manhattan (- i u) (- j v)) (manhattan u v)))
          ))
    ))
```

**6. Astrazione funzionale** (ai fini della valutazione, questo esercizio ha un peso minore degli altri)

Riesci a intuire quale problema risolve (o che funzione calcola) la procedura `paths` definita nell'esercizio **2**?

Numero di percorsi di Manhattan in un reticolo $i \times j$ per cui la direzione di spostamento cambia al più $k$ volte.