Corso di Programmazione

Esame del 29 Luglio 2020

cognome e nome

Risolvi i seguenti esercizi giustificando sinteticamente le risposte.

1. Ricorsione di coda e correttezza dei programmi ricorsivi

La procedura `pattern-count` calcola il numero di occorrenze del pattern *p* nel testo *s*, dove *p* e *s* sono stringhe.

```
(define pattern-count   ; val : intero
  (lambda (p s)         ; p ≠ "", s : stringhe
    (count-rec p s (string-length p) (string-length s) 0)
    ))

(define count-rec       ; val : intero
  (lambda (p s k n r)   ; p ≠ "", s : stringhe;  k = len(p), n ≤ len(s), r : interi
    (cond ((< n k)
           r)
          ((string=? p (substring s (- n k) n))
           (count-rec p s k (- n 1) (+ r 1)))
          (else
           (count-rec p s k (- n 1) r))
          )))
```

Formalmente, per ogni coppia di stringhe $p \neq ""$ (p diversa dalla stringa vuota) e s :

$$\texttt{(pattern-count } p\ s\texttt{)} \quad \rightarrow \quad |\ \{\ i :\ 0 \le i \le len(s)\text{–}len(p)\ \wedge\ p = sub(\ s,\ i,\ i+len(p)\ )\ \}\ |$$

dove $len(x)$ è la lunghezza della stringa x e $sub(x,i,j)$ è la sottostringa di x dal carattere di posizione i a quello di posizione j–1. (Il risultato è espresso in termini di cardinalità dell'insieme di indici per i quali valgono le prorpietà specificate a destra dei due punti.)

1.1. Considera la procedura ricorsiva `count-rec`: come può essere formalizzato il valore restituito da `count-rec` per poter eventualmente dimostrare la correttezza sia di `count-rec` che di `pattern-count`?
Più precisamente, date due stringhe $p \neq ""$, s, nonché gli interi $k = len(p)$, $n \le len(s)$, r :

`(count-rec` $p\ s\ k\ n\ r$`)` $\rightarrow$ .....

1.2. La procedura `count-rec` applica la *ricorsione di coda* (tail recursion) e pertanto la relativa computazione può essere rielaborata in forma iterativa introducendo variabili di stato che corrispondano a ciascuno dei parametri della procedura. Scrivi un programma in Java basato su un ciclo *while* per realizzare una computazione sostanzialmente equivalente a quella di `count-rec` (indipendentemente dall'eventuale utilizzo di `pattern-count`).

2. Argomenti procedurali in Scheme

Il programma impostato nel riquadro è inteso a realizzare in un modo alternativo la procedura `pattern-count` per calcolare il numero di occorrenze del pattern *pattern* nel testo *text*, dove entrambi gli argomenti sono stringhe. In questa versione si applica due volte la procedura predefinita `map`: la prima per passare da una lista di posizioni nel testo alla lista di tutte le possibili sottostringhe del testo di lunghezza pari a quella del pattern, la seconda per identificare le corrispondenze con il pattern (destinate al successivo conteggio).

2.1. Completa la definizione della procedura `pattern-count`.

```
(define pattern-count      ; val : intero
  (lambda (pattern text)   ; pattern ≠ "", text : stringhe
    (let (
          (k (string-length pattern))
          (n (string-length text))
          )
      (let ((subtetxts

              (map ..........
                   (index-range 0 (- n k))))
                   )
        (list-sum

         (map (lambda (x) (if ..... 1 0))
              subtetxts
              ))
        ))
    ))

(define index-range
  (lambda (inf sup)
    (if (> inf sup)
        null
        (cons inf (index-range (+ inf 1) sup))
        )))

(define list-sum
  (lambda (nums)
    (if (null? nums)
        0
        (+ (car nums) (list-sum (cdr nums)))
        )))
```

2.2. Considera la valutazione dell'esempio:

```
(pattern-count "nihil" "ex nihilo nihil")
```

Qual è la lista passata come secondo argomento e la lista restituita come risultato per ciascuna applicazione di `map`?

- Applicazione 1 – lista argomento: .....
- Applicazione 1 – lista restituita: .....
- Applicazione 2 – lista argomento: .....
- Applicazione 2 – lista restituita: .....

3. Memoization

Considera la seguente procedura funzionale (metodo statico), basata su una ricorsione ad albero:

```
public static long rec( int n, int k ) {  // 0 ≤ k ≤ n

  if ( (n-k) * k <= 0 ) {
    return 1;
  } else {
    return rec( n-1, k ) + rec( n-1, n-k );
  }
}
```

3.1. Supponi che nel corso dell'esecuzione di un programma che utilizza `rec` venga valutata l'espressione:

```
rec( 10, 4 )
```

Questa valutazione si svilupperà attraverso successive invocazioni ricorsive di `rec(`n`,`k`)` per diversi valori degli argomenti n e k. Quali sono il valore più piccolo e il valore più grande che assumerà ciascuno degli argomenti nelle ricorsioni che discendono da `rec(10,4)` ?

- Valore più piccolo di n : ..... e valore più grande di n : ..... ;
- Valore più piccolo di k : ..... e valore più grande di k : ..... .

3.2. Applica una tecnica *top-down* (ricorsiva) di memoization per realizzare la procedura `rec` in modo più efficiente.

4. Correttezza dei programmi iterativi

In corrispondenza al programma iterativo che realizza la procedura `f` sono riportate precondizione (*Pre*), postcondizione (*Post*), invariante (*Inv*) e funzione di terminazione (*Term*). In particolare, si intende che l'invariante e la funzione di terminazione consentono di dimostrare la correttezza completa del programma in relazione a quanto specificato da precondizione e postcondizione.

4.1. Completa la definizione di `f` in modo che le proprietà formalizzate nell'invariante siano soddisfatte e che l'espressione associata a *Term* definisca effettivamente una funzione di terminazione.

```
public static int f( int n ) {   // Pre:  n > 0

  int x = 1;

  int y = ..... ;

  int z = ..... ;

  while ( x + y <= n ) {          // Inv:  ∃k . x=2^k ∧ 1 ≤ y ≤ x ∧ x+y ≤ n+1 ∧ z+1 = 2y
                                  // Term:  n+1 – x – y
    if ( x == y ) {

      x = ..... ;
      y = 1;

      z = ..... ;
    } else {
      y = y + 1;

      z = z + ..... ;
  }}
  return z;                       // Post:  ∃k . x=2^k ∧ 2x > n ∧ 1 ≤ y ≤ x ∧ x+y = n+1 ∧ z+1 = 2y
}
```

4.2. Dimostra che al termine dell'iterazione le proprietà specificate dalla postcondizione (*Post*) sono una conseguenza logica di quelle relative all'invariante (*Inv*).

5. Classi in Java

Il package `java.awt`, destinato alla realizzazione di interfacce grafiche in Java, rende disponibili diversi strumenti, fra i quali una classe `Color` per istanziare colori specifici. Ora si vuole definire una nuova classe `ColorRandomizer`, per rappresentare "collezioni" di colori accessibili in modo causale: fra i colori contenuti sarà possibile pescarne uno a caso (`pick`) oppure estrarne uno, sempre scelto casualmente, rimuovendolo inoltre dalla collezione (`extract`).
Più precisamente, il protocollo di `ColorRandomizer` può essere specificato come segue:

```
ColorRandomizer r = new ColorRandomizer();      // costruttore di una collezione di colori vuota

r.size()        // numero di colori che fanno parte della collezione

r.add(c)        // aggiunge il colore c alla collezione

r.pick()        // restituisce un colore scelto a caso fra quelli presenti, senza modificare la collezione

r.extract()     // restituisce e rimuove un colore casuale fra quelli inclusi nella collezione
```

Per gestire le operazioni casuali puoi utilizzare il metodo statico `Math.random()` che restituisce un valore di tipo `double` scelto in modo random nell'intervallo [0, 1[(zero incluso, uno escluso).

5.1. Definisci le variabili di istanza e il costruttore di una classe Java `ColorRandomizer` che realizzi il protocollo specificato sopra.

5.2. Completa la definizione della classe realizzando i metodi `size`, `add`, `pick` ed `extract` in ottemperanza al protocollo specificato sopra.