Corso di Informatica@Matematica
Prova scritta del 22 giugno 2016

Tempo a disposizione: ore 2.
Esercizi (1, 2, 3) e (4, 5) su due fogli differenti. Es. ?? e ?? necessari per la sufficienza.
In entrambi gli esercizi è possibile definire funzioni ausiliarie.

1. (6 punti) Il codice ASCII su 7 bit della cifra n corrisponde alla codifica binaria (assoluta) di $48+n$. Si supponga di estendere tale codifica con uno "1" più significativo e di disporre da sinistra (più significativo) a destra le codifiche di 7 e 4 (ciascuna di 8 bit) ottenute nel modo descritto. Si interpreti ora tale sequenza di sedici bit come un numero in complemento a due e lo si cambi di segno. Si dia la rappresentazione in complemento a due e in ottale (cioè in base 8) del risultato così ottenuto.

   Sol: La codifica ASCII (sette bit) di 7 è `011 0111`; quella di 4 è `011 0100`. Estendendo entrambe a sinistra con 1 e giustapponendole si ottiene `1011 0111 1011 0100`. Cambiamo il segno, lavorando in complemento a due: `0100 1000 0100 1100`. Raggruppiamo a tre a tre da destra e convertiamo ogni tripletta in ottale (separatamente): `0 100 100 001 001 100`, da cui `044114`, che è quanto richiesto.

2. Si consideri il seguente frammento in Python:

```
a = 0
def f(S):
    for x in S:
        a = a + x
    return a
```

   (i) Si tratta di un programma sintatticamente corretto? (ii) Cosa avviene chiamando la funzione `f` come `f([2,3,4])`?

   Sol: (i) Sì. (ii) Restituisce l'errore `local variable 'a'referenced before assignment` perché la variabile `a` usata all'interno della funzione è locale a `f`, diversa cioè dalla `a` definita alla linea 1 del codice proposto. Si tratta dunque di un riferimento ad `a` (a destra dell'assegnamento) prima che essa abbia ricevuto un valore.

3. Si scriva una funzione `inverti(diz)` che, preso come argomento un dizionario nel quale chiavi ed elementi sono interi, restituisce un nuovo dizionario nel quale il ruolo delle chiavi e dei valori è invertito. Siccome un valore può essere presente più volte in `diz` (con chiavi diverse), il nuovo dizionario dovrà avere associato ad ogni (vecchio) valore la somma delle (vecchie) chiavi. Esempio: `inverti({0:0, 1:100, 2:200, 3:100, 4:100})` restituirà il dizionario `{0:0, 100:8, 200:2}`.

   Sol: Una soluzione compatta è

```
def inverti(D):
  R={}
  for x in D:
    R[D[x]] = R.get(D[x],0) + x
  return R
```

4. (6 punti) Cosa stampa il seguente frammento di codice Python (si possono usare artifici grafici per descrivere l'output):

```
A = [1,2,1]
B = [0,A,0]

def g(x, y):
    x[1]=y
    y[x[2]]=8

C=B
g(A,C)
B = A.append(A[1])
print(A,B,C)
```

   Sol: `[1, [0, 8, 0], 1, [0, 8, 0]] None [0, 8, 0]`

5. (7 punti) Si supponga di avere a disposizione le seguenti funzioni Python, che permettono di costruire e manipolare alberi binari (come sono realizzate non è significativo per l'esercizio):

```
def tree():             # Restituisce l'albero vuoto
def bin(a,l,r):         # Costruisce (restituisce) l'albero con radice etichettata a,
                        # figlio sinistro l e figlio destro r
def is_empty(a):        # Restituisce True sse a e' l'albero vuoto
def label(a):           # Restituisce l'etichetta della radice dell'albero a
def first_child(a):     # Restituisce il figlio sinistro dell'albero a
def second_child(a):    # Restituisce il figlio destro dell'albero a
```

Un albero binario è *uniformemente etichettato* se tutti i nodi hanno la stessa etichetta. Scrivere una funzione `conta_uniformi(a)` che, dato in input un albero binario `a` restituisce il numero di sottoalberi uniformemente etichettati di `a`. Per esempio, il risultato dell'esecuzione della funzione sull'albero

```
    5
   / \
  4   5
 / \   \
4   4   5
```

è `5` (le tre foglie; l'albero di radice `4` con i due figli foglia; l'albero di radice `5` con un solo figlio foglia).

Sol: La soluzione modulare utilizza una funzione ausiliaria `e_uniforme` che ritorna `True` se tutti i nodi dell'albero `a` sono etichettati con `r`, e procede con una visita postfissa che incrementa il risultato se l'albero corrente è uniforme:

```
def e_uniforme(a, r):
    if is_empty(a):
        return True
    elif label(a) != r:
        return False
    else:
        return e_uniforme(first_child(a), r) and e_uniforme(second_child(a), r)

def conta_uniformi(a):
    if is_empty(a):
        return 0
    cf = conta_uniformi(first_child(a))
    cs = conta_uniformi(second_child(a))
    if e_uniforme(a, label(a)):
        return cf + cs + 1
    else:
        return cf + cs
```

La soluzione precedente non è quella più efficiente, perché effettua un numero di visite dell'albero pari al numero di nodi di `a`. La soluzione seguente visita ogni nodo una sola volta e utilizza una funzione ausiliaria che ritorna una coppia dove il primo elemento è `True` se tutti i nodi dell'albero `a` sono etichettati con `r`, ed il secondo elemento è il numero di sottoalberi uniformemente etichettati di `a`.

```
def aux(a, r):
    if is_empty(a):
        return (True, 0)
    (uf, cf) = aux(first_child(a), label(a))
    (us, cs) = aux(second_child(a), label(a))
    if uf and us:
        return (label(a) == r, cf + cs + 1)
    else:
        return (False, cf + cs)

def conta_uniformi(a):
    if is_empty(a):
        return 0
    else:
        return aux(a, label(a))[1]
```