Συναρτησιακός Προγραμματισμός 2008
Λύσεις Επαναληπτικής Εξέτασης

1. Γράψτε μία συνάρτηση που να παίρνει ένα πίνακα ακεραίων δύο διαστάσεων και επιστρέφει τον αριθμό της πρώτης σειράς της οποίας όλα τα στοιχεία είναι ίσα με το μηδέν. Αν δεν υπάρχει τέτοια σειρά, τότε η συνάρτηση πρέπει να επιστρέφει τον αριθμό όλων των σειρών του πίνακα. Για παράδειγμα:

   ```
   first_zero_row [[1,2],[0,0],[3,4]] = 1
   first_zero_row [[1,2],[3,4]] = 2
   ```

   §

   ```
   first_zero_row [] = 0
   first_zero_row (h:t)
     | [x | x<-h, x/=0] == []  =  0
     | otherwise  =  1+first_zero_row t
   ```

2. Φτιάξτε ένα *διαδραστικό* πρόγραμμα τρίλιζας για δύο παίχτες. Οι κινήσεις και των δύο παιχτών εισάγονται από το πληκτρολόγιο (το πρόγραμμά σας δε θα παίζει τρίλιζα). Το πρόγραμμα θα πρέπει σε κάθε κίνηση να τυπώνει το ταμπλώ του παιχνιδιού, να παίρνει την επόμενη κίνηση από τον παίχτη, να ελέγχει αν η κίνηση τελειώνει το παιχνίδι (και αν ναι να τυπώνει το αποτέλεσμα και να τερματίζει) και να επαναλαμβάνει.

   § ...

3. Η Haskell επιτρέπει λίστες με σταθερό βάθος. Για παράδειγμα, ο τύπος `[Int]` περιγράφει λίστες με βάθος 1, ο τύπος `[[Int]]` περιγράφει λίστες με βάθος 2, κτλ. Δεν υπάρχει τύπος του οποίου τα στοιχεία να είναι λίστες με διαφορετικό βάθος η μία από την άλλη. Επίσης δε μπορούμε να κατασκευάσουμε λίστες των οποίων το βάθος να ποικίλει, π.χ. `[1,[2],[[3]]]`.

   (α) Κατασκευάστε έναν πολυμορφικό τύπο `EList a` του οποίου τα αντικείμενα να είναι λίστες *ποικίλου* βάθους που περιέχουν στοιχεία του τύπου `a`. Φυσικά, οι λίστες σας δε θα ακολουθούν τη σύνταξη για λίστες της Haskell.

   §

   ```
   data EList a = Lf a | Lst [EList a]
   ```

(β) Κατασκευάστε μία λίστα τύπου `EList Int` που να αντιστοιχεί στο παραπάνω παράδειγμα, δηλ. στη λίστα `[1,[2],[[3]]]`.

§

```
Lst [Lf 1, Lst [Lf 2], Lst [Lst [Lf 3]]]
```

(γ) Κατασκευάστε μία λίστα με *άπειρο* βάθος και δώστε τον πιο γενικό πολυμορφικό τύπο που την περιγράφει.

§

```
inf_dep_list = Lst [inf_dep_list]

inf_dep_list :: EList a
```

4. Η μονάδα `Log` ορίζεται ως εξής:

```
data Log a = Log (a,String)
instance Monad Log where
  return r = Log (r,"")
  Log (r,s) >>= k = Log (r',s++s')
    where Log (r',s') = k r
```

Αποδείξτε ότι η `Log` είναι πράγματι μονάδα, δηλαδή ότι ικανοποιεί τις ιδιότητες:

```
return x >>= f   =   f x
mx >>= return   =   mx
mx >>= (\x-> f x >>= g)   =   (mx >>= f) >>= g
```

§ Πρώτη ιδιότητα:

```
   return x >>= f
=  Log (x,"") >>= f
=  (Log (x',""++c') where Log (x',c') = f x)
=  (Log (x',c') where Log (x',c') = f x)
=  f x
```

Δεύτερη ιδιότητα:

```
   Log (x,s) >>= return
=  (Log (x',s++s') where Log (x',s') = return x)
=  (Log (x',s++s') where Log (x',s') = Log (x,""))
=  Log (x,s++"")
=  Log (x,s)
```

Τρίτη ιδιότητα:

```
    Log (x,s) >>= (\x-> f x >>= g)
=   (Log (x',s++s') where Log (x',s') = ((\x-> f x >>= g) x))
=   (Log (x',s++s') where Log (x',s') = f x >>= g)
=   ...
=   (Log (x,s) >>= f) >>= g
```