Συναρτησιακός Προγραμματισμός 2008
Λύσεις στο Δεύτερο Φύλλο Ασκήσεων

1. Στις Σημ. 4, είδαμε τη δημιουργία της κλάσης `Condition` που μας επιτρέπει να χρησιμοποιούμε αριθμούς, λίστες και ζεύγη ως αληθοτιμές στη συνάρτηση `cond` που αποτελεί γενίκευση της `if then else`.

   Θα θέλαμε να επεκτείνουμε την κλάση αυτή, έτσι ώστε να μπορούμε να χειριστούμε μία λίστα αληθοτιμών στην `cond` είτε ως σύζευξη είτε ως διάζευξη των στοιχείων της. Για παράδειγμα, η λίστα `[2::Int,False]` θα πρέπει να λειτουργεί σαν `True` στην περίπτωση που τη βλέπουμε σα διάζευξη (το `2::Int` λειτουργεί σαν `True`), ενώ θα πρέπει να λειτουργεί σα `False` στην περίπτωση που τη βλέπουμε σα σύζευξη.

   Το πρόβλημα είναι ότι οι λίστες από μόνες τους δε μπορούν να μας πουν αν τις βλέπουμε σα σύζευξη ή σα διάζευξη. Επιπλέον, οι λίστες έχουν ήδη ένα χειρισμό στην `Condition`: μία λίστα συμπεριφέρεται σαν `True` αν και μόνον αν δεν είναι κενή.

   Χρησιμοποιώντας αλγεβρικούς τύπους, λύστε αυτό το πρόβλημα και επεκτείνετε την κλάση `Condition` ώστε να διατηρήσει τη λειτουργικότητα που έχει τώρα, αλλά να υποστηρίζει και σύζευξη / διάζευξη στοιχείων μίας λίστας.

   Θα βρείτε τον κώδικα ορισμού της κλάσης `Condition` όπως είναι τώρα στο αρχείο
   `http://cgi.di.uoa.gr/~kassios/courses/fp/exercises/Condition.hs`

   § Προσθέτουμε στο αρχείο `Condition.hs` τα εξής:

```
data CondList a = And [a] | Or [a]

instance ConvBool a => ConvBool (CondList a) where
  toBool (And l) = foldl (&&) True (map toBool l)
  toBool (Or l) = foldl (||) False (map toBool l)
instance ConvBool a => Condition (CondList a)
```

2. Εμπλουτίστε τον αλγεβρικό τύπο `Expr` των Σημ. 5, Ενότ. 8.2 με μία δομή `let` που εκχωρεί τιμές σε μεταβλητές. Μετά κατασκευάστε πλήρη διερμηνέα `eval` για τις εκφράσεις αυτές. Ο διερμηνέας θα πρέπει να επιστρέφει τιμή τύπου `Maybe Int`, καθώς θα μπορούσαν να συμβούν λάθη κατά την αποτίμηση μίας έκφρασης, όπως διαίρεση με το μηδέν ή αποτίμηση μίας μεταβλητής στην οποία δεν έχει εκχωρηθεί τιμή.

§

```
data Expr =  Constant Int
           | Variable String
           | UnaryExp UnOp Expr
           | BinaryExp BinOp Expr Expr
           | Let String Expr Expr
data UnOp = UPlus | UMinus
data BinOp = Plus | Minus | Mult | Div

type Env = [(String,Int)]

eval :: Env -> Expr -> Maybe Int
eval _ (Constant n) = Just n
eval env (Variable v) = find v env
  where find _ [] = Nothing
        find v1 ((v2,val):vs)
          = if v1 == v2 then Just val
            else find v1 vs
eval env (UnaryExp uo exp) = uod uo (eval env exp)
eval env (BinaryExp bo exp1 exp2)
   = bod bo (eval env exp1) (eval env exp2)
eval env (Let v exp1 exp2)
   = case eval env exp1 of
       Nothing -> Nothing
       Just val -> eval ((v,val):env) exp2

uod :: UnOp -> Maybe Int -> Maybe Int
uod _ Nothing = Nothing
uod UPlus (Just x) = Just x
uod UMinus (Just x) = Just (-x)

bod :: BinOp -> Maybe Int -> Maybe Int -> Maybe Int
bod _ Nothing _ = Nothing
```

```
bod _ _ Nothing = Nothing
bod Plus (Just x) (Just y) = Just (x+y)
bod Minus (Just x) (Just y) = Just (x-y)
bod Mult (Just x) (Just y) = Just (x*y)
bod Div (Just x) (Just y)
   = if y == 0 then Nothing else Just (x`div`y)
```

3. Ένα *πολυσύνολο (multiset)* είναι μία συλλογή δεδομένων στην οποία η σειρά δεν έχει σημασία, αλλά ο πληθυσμός έχει σημασία. Δηλαδή, ένα πολυσύνολο που περιέχει τα `1,2` είναι ίσο με ένα πολυσύνολο που περιέχει τα `2,1`, αλλά όχι ίσο με αυτό που περιέχει τα `1,2,1`. Υλοποιήστε έναν πολυμορφικό ΑΤΔ `MultiSet` για πολυσύνολα. Υποστηρίξτε την κατασκευή τουλάχιστον όλων των πεπερασμένων πολυσυνόλων και την εξαγωγή όλων των χρήσιμων πληροφοριών από ένα πολυσύνολο. Υποστηρίξτε πράξεις `union` (ένωση) και `difference` (διαφορά) φροντίζοντας να είναι άμεσες γενικεύσεις των αντίστοιχων πράξεων για σύνολα. Μπορείτε να φτιάξετε και ό,τι άλλη συνάρτηση νομίζετε ότι μπορεί να διευκολύνει τη χρήση πολυσυνόλων. Τέλος, δηλώστε τον τύπο σας ως στιγμιότυπο της κλάσης `Eq`.

   § Η υλοποίηση βασίζεται σε μία λίστα ζευγών. Το πρώτο στοιχείο ενός ζεύγους είναι ένα στοιχείο που βρίσκεται στο πολυσύνολο. Το δεύτερο στοιχείο είναι ο πληθυσμός του στοιχείου αυτού στο πολυσύνολο και πρέπει να είναι θετικός ακέραιος (όχι `0`). Μία ακόμα αναλλόιωτη είναι ότι ένα στοιχείο που βρίσκεται μέσα στο πολυσύνολο βρίσκεται μόνο μία φορά μέσα στη λίστα.

   Υλοποιούμε τις συναρτήσεις `emptyms` (κενό πολυσύνολο), `fromList` (διαβάζει ένα πολυσύνολο από μία λίστα), `union` (ένωση), `difference` (διαφορά), `card` (η γενίκευση του `el` των συνόλων: επιστρέφει τον πληθυσμό ενός στοιχείου στο πολυσύνολο) και `isempty` (έλεγχος αν το στοιχείο είναι άδειο).

   Επίσης, έχουμε χρησιμοποιήσει μία μη εξαγόμενη συνάρτηση `insert` που εισάγει ένα στοιχείο σε μία λίστα πολυσυνόλου διατηρώντας τις αναλλοίωτες των πολυσυνόλων.

   Ο έλεγχος ισότητας δεν είναι ο καλύτερος δυνατός (μπορεί να γίνει και με μία μόνο πράξη στα δύο πολυσύνολα), αλλά είναι η πιο εύκολη υλοποίηση με τις συναρτήσεις που έχουμε.

```
module MultiSet
 (MultiSet,emptyms,fromList,union,difference,card,isempty)
 where

data Eq a => MultiSet a = MS [(a,Int)]
```

```
emptyms :: Eq a => MultiSet a
emptyms = MS []

fromList :: Eq a => [a] -> MultiSet a
fromList = fromListaux emptyms
  where
  fromListaux m [] = m
  fromListaux m (x:xs) = fromListaux (insert (x,1) m) xs

card x (MS []) = 0
card x (MS ((y,cy):xs))
  = if x==y then cy else card x (MS xs)

insert (x,cx) (MS []) = if cx > 0 then MS[(x,cx)] else MS[]
insert (x,cx) (MS ((y,cy):ys))
  = if x == y then
      (if sum > 0 then MS ((y,sum):ys) else (MS ys))
    else MS ((y,cy):rest)
  where sum = cx + cy
        MS rest = insert (x,cx) (MS ys)

union m (MS []) = m
union m (MS ((x,cx):xs)) = (insert (x,cx) m) `union` (MS xs)

difference m (MS []) = m
difference m (MS ((x,cx):xs))
  = (insert (x,-cx) m) `difference` (MS xs)

isempty (MS []) = True
isempty (MS _) = False

instance Eq a => Eq (MultiSet a) where
  m1 == m2
     = isempty (difference m1 m2) && isempty (difference m2 m1)
```

4. Χρησιμοποιήστε το τμήμα `MultiSet` που φτιάξατε για να ορίσετε μία συνάρτηση `perm` που να παίρνει δύο λίστες και να επιστρέφει `True` αν και μόνον αν η μία είναι αναμετάθεση της άλλης.

§

```
import MultiSet
perm l1 l2 = fromList l1 == fromList l2
```

5. Υλοποιήστε το τμήμα `ResettableVariable` με υπογραφή εξόδου:

   - `ResVar a` (Πολυμορφικός ΑΤΔ)
   - `newRV :: ResVar`
   - `value :: ResVar a -> a`
   - `set :: ResVar a -> a -> ResVar`
   - `reset :: ResVar a -> ResVar`

   και με συμβόλαιο (για κάθε `r::ResVar a` και `v::a`)

```
   value(set r v) = v
&& value(reset(set r v)) = value r
&& reset . reset = reset
```

   Αποδείξτε ότι το συμβόλαιο ικανοποιείται.
   § Μία υλοποίηση έχει ως εξής:

```
module ResettableVariable(ResVar,newRV,value,set,reset) where

data ResVar a = R (a,a)

undef = undef
newRV = R(undef,undef)
value (R(n,_)) = n
set (R(n,o)) x = R(x,n)
reset (R(n,o)) = R(o,o)
```

   Αποδείξεις:

```
value(set (R(n,o)) v) = value (R(v,n)) = v

  value(reset(set (R(n,o)) v)) = value(reset(R(v,n)))
= value(R(n,n)) = n = value(R(n,o))

reset(reset(R(n,o))) = reset(R(o,o)) = R(o,o) = reset(R(n,o))
```