```haskell
-- Laboratorul 7

-- Se dau următoarele tipuri de date reprezentând expresii si arbori de expresii:

data Expr = Const Int -- integer constant
            | Expr :+: Expr -- addition
            | Expr :*: Expr -- multiplication
    deriving Eq

data Operation = Add | Mult deriving (Eq, Show)

data Tree = Lf Int -- leaf
            | Node Operation Tree Tree -- branch
    deriving (Eq, Show)

-- 1.1. Să se instantieze clasa Show pentru tipul de date Expr, astfel încât să
-- se afiseze mai simplu expresiile.

exp1 = ((Const 2 :*: Const 3) :+: (Const 0 :*: Const 5))
exp2 = (Const 2 :*: (Const 3 :+: Const 4))
exp3 = (Const 4 :+: (Const 3 :*: Const 3))
exp4 = (((Const 1 :*: Const 2) :*: (Const 3 :+: Const 1)) :*: Const 2)

instance Show Expr where
    show (Const x) =  show x
    show (a :+: b) =  "(" ++ show a ++ "+" ++ show b ++ ")"
    show (a :*: b) =   "(" ++ show a ++ "*" ++ show b ++ ")"

-- 1.2. Să se scrie o functie evalExp :: Expr -> Int care evaluează o expresie
-- determinând valoarea acesteia.
-- test11 = evalExp exp1 == 6
-- test12 = evalExp exp2 == 14
-- test13 = evalExp exp3 == 13
-- test14 = evalExp exp4 == 16

evalExp :: Expr -> Int
evalExp (Const a) = a
evalExp (a :+: b) = evalExp a + evalExp b
evalExp (a :*: b) = evalExp a * evalExp b

-- 1.3. Să se scrie o functie evalArb :: Tree -> Int care evaluează o expresie
-- modelată sub formă de arbore, determinând valoarea acesteia.
arb1 = Node Add (Node Mult (Lf 2) (Lf 3)) (Node Mult (Lf 0)(Lf 5))
arb2 = Node Mult (Lf 2) (Node Add (Lf 3)(Lf 4))
arb3 = Node Add (Lf 4) (Node Mult (Lf 3)(Lf 3))
```

```haskell
arb4 = Node Mult (Node Mult (Node Mult (Lf 1) (Lf 2)) (Node Add (Lf 3)(Lf 1)))
(Lf 2)

-- test21 = evalArb arb1 == 6
-- test22 = evalArb arb2 == 14
-- test23 = evalArb arb3 == 13
-- test24 = evalArb arb4 == 16

evalArb :: Tree -> Int
evalArb (Lf f) = f
evalArb (Node Add a b) = evalArb a + evalArb b
evalArb (Node Mult a b) = evalArb a * evalArb b

-- 1.4. Să se scrie o functie expToArb :: Expr -> Tree care transformă o expresie
-- în arborele corespunzător.

expToArb :: Expr -> Tree
expToArb (Const f) = Lf f
expToArb (a :+: b) = Node Add (expToArb a) (expToArb b)
expToArb (a :*: b) = Node Mult (expToArb a) (expToArb b)

-- In acest exercitiu vom exersa manipularea listelor si tipurilor de date prin
-- implementarea catorva colectii de tip tabela asociativa cheie-valoare.
-- Aceste colectii vor trebui sa aiba urmatoarele facilitati
-- • crearea unei colectii vide
-- • crearea unei colectii cu un element
-- • adaugarea/actualizarea unui element intr-o colectie
-- • cautarea unui element intr-o colectie
-- • stergerea (marcarea ca sters a) unui element dintr-o colectie
-- • obtinerea listei cheilor
-- • obtinerea listei valorilor
-- • obtinerea listei elementelor

class Collection c where
    empty :: c key value
    singleton :: key -> value -> c key value
    insert :: Ord key => key -> value -> c key value -> c key value
    clookup :: Ord key => key -> c key value -> Maybe value
    delete :: Ord key => key -> c key value -> c key value
    keys :: c key value -> [key]
    values :: c key value -> [value]
    toList :: c key value -> [(key, value)]
    fromList :: Ord key => [(key,value)] -> c key value

-- 2.1. Adaugati definitii implicite (in functie de functiile celelalte) pentru:
```

```haskell
-- a. keys
    keys c = map fst (toList c)

-- b. values
    values c = map snd (toList c)

-- c. fromList
    fromList [] = empty
    fromList ((key, value) : xs) = insert key value (fromList xs)

-- 2.2. Fie tipul listelor de perechi de forma cheie-valoare:

newtype PairList k v
    = PairList {getPairList :: [(k, v)]}

-- Faceti PairList instanta a clasei Collection.

instance Collection PairList where

    empty :: PairList key value
    empty = PairList []

    singleton :: key -> value -> PairList key value
    singleton key value = PairList[(key, value)]

    insert :: Ord key => key -> value -> PairList key value -> PairList key value
    insert key value (PairList list) = if key `elem` keys (PairList list) then
PairList list
                                       else PairList ((key, value) : list )

    delete :: Ord key => key -> PairList key value -> PairList key value
    delete key (PairList list) = PairList [(good_key, good_value) | (good_key,
good_value) <- list, good_key /= key]

    keys :: PairList key value -> [key]
    keys (PairList list) = map fst list

    values :: PairList key value -> [value]
    values (PairList list) = map snd list

    fromList :: Ord key => [(key, value)] -> PairList key value
    fromList [] = empty
    fromList ((key, value) : xs) = insert key value (fromList xs)
```

```haskell
    toList :: PairList key value -> [(key, value)]
    toList = getPairList

    clookup :: Ord key => key -> PairList key value -> Maybe value
    clookup key (PairList list) = lookup key list


-- 2.3. Fie tipul arborilor binari de cautare (ne-echilibrati):

data SearchTree key value
    = Empty
    | BNode
        (SearchTree key value) -- elemente cu cheia mai mica
        key -- cheia elementului
        (Maybe value) -- valoarea elementului
        (SearchTree key value) -- elemente cu cheia mai mare

-- Observati ca tipul valorilor este Maybe value. Acest lucru se face pentru a
-- reduce timpul operatiei de stergere prin simpla marcare a unui nod ca fiind
-- sters. Un nod sters va avea valoarea Nothing.

-- Faceti SearchTree instanta a clasei Collection.

instance Collection SearchTree where

    empty :: SearchTree key value
    empty = Empty

    singleton :: key -> value -> SearchTree key value
    singleton key value = BNode Empty key (Just value) Empty

    insert :: Ord key => key -> value -> SearchTree key value -> SearchTree key
value
    insert key value Empty = singleton key value
    insert key value (BNode littleTree current_key current_value biggerTree)
        | key > current_key = BNode littleTree current_key current_value (insert
key value biggerTree)
        | key < current_key = BNode (insert key value littleTree) current_key
current_value biggerTree
        | otherwise = BNode littleTree current_key current_value biggerTree

    clookup :: Ord key => key -> SearchTree key value -> Maybe value
    clookup key Empty = Nothing
    clookup key (BNode littleTree current_key current_value biggerTree)
        | key == current_key = current_value
```

```
        | key > current_key = clookup key littleTree
        | otherwise = clookup key biggerTree

    toList :: SearchTree key value -> [(key, value)]
    toList Empty = []
    toList (BNode littleTree key Nothing biggerTree) = (toList littleTree) ++
(toList biggerTree)
    toList (BNode littleTree key (Just value) biggerTree) = (toList littleTree)
++ [(key, value)] ++ (toList biggerTree)

    delete :: Ord key => key -> SearchTree key value -> SearchTree key value
    delete key (BNode littleTree current_key current_value biggerTree)
        | key == current_key = BNode littleTree current_key Nothing biggerTree
        | key > current_key = delete key biggerTree
        | otherwise = delete key littleTree

    keys :: SearchTree key value -> [key]
    keys (BNode littleTree key (Just value) biggerTree) = [fst l | l <- toList
(BNode littleTree key (Just value) biggerTree)]

    values :: SearchTree key value -> [value]
    values (BNode littleTree key (Just value) biggerTree) = [snd l | l <- toList
(BNode littleTree key (Just value) biggerTree)]

    fromList :: Ord key => [(key, value)] -> SearchTree key value
    fromList [] = empty
    fromList ((current_key, current_value) : xs) = insert current_key
current_value (fromList xs)
```