LAB 1

```haskell
import Data.List

main = putStrLn "Hello, world!"
myInt =
55555555555555555555555555555555555555555555555555555555555555
double :: Integer -> Integer
double x = x+x

triple :: Integer -> Integer
triple x = 3*x

maxim :: Integer -> Integer -> Integer
maxim a b = if (a > b) then a else b

maxim3 :: Integer -> Integer -> Integer -> Integer
maxim3 a b c = let
                   u = maxim a b
               in
                   maxim u c

maxim4 :: Integer -> Integer -> Integer -> Integer -> Integer
maxim4 a b c d = let
                   u = maxim a b
                   v = maxim c d
               in
                   maxim u v

-- Exercitii
-- a)
suma_patrate :: Integer -> Integer -> Integer
suma_patrate a b = a*a + b*b

-- b)
```

```haskell
par_impar :: Integer -> [Char]
par_impar a = if (mod a 2 == 0) then "par" else "impar"


-- c)
factorial :: Integer -> Integer
factorial a =
           if (a == 0)
               then 1
               else a * factorial (a-1)



-- d)
dublu :: Integer -> Integer -> Bool
dublu a b = if (a > 2*b) then True else False
-- sau dublu a b = a > 2 * b
```

LAB 2

```haskell
poly :: Double -> Double -> Double -> Double -> Double
poly a b c x = a*x*x + b*x + c

eeny :: Integer -> String
eeny x
   | even x = "eeny"
   | otherwise = "meeny"

fizzbuzz :: Integer -> String
fizzbuzz x
       | mod x 15 == 0 = "FizzBuzz"
       | mod x 3 == 0 = "Fizz"
       | mod x 5 == 0 = "Buzz"
       | otherwise = ""

fizzbuzz2 :: Integer -> String
fizzbuzz2 x =   if (mod x 15 == 0)
                  then "FizzBuzz"
              else
                  if (mod x 3 == 0)
                  then "Fizz"
                  else
                      if (mod x 5 == 0)
                          then "Buzz"
                      else ""

tribonacci :: Integer -> Integer
tribonacci n
   | n < 3     = 1
   | n == 3    = 2
   | otherwise = tribonacci (n-1) + tribonacci (n-2) +
tribonacci(n-3)
```

```haskell
binomial :: Integer -> Integer -> Integer
binomial n k
   | k == 0 = 1
   | n == 0 = 0
   | otherwise = binomial (n-1) k + binomial (n-1) (k-1)


verifL :: [Int] -> Bool
verifL lst = even (length lst)

takefinal :: [a] -> Int -> [a]
takefinal lst n
   | length lst < n = lst
   | otherwise = drop (length lst - n) lst

remove :: [a] -> Int -> [a]
remove lst n = take (n-1) lst ++ drop n lst


myreplicate :: Int -> v -> [v]
myreplicate 1 v = [v]
myreplicate n v = [v] ++ myreplicate (n-1) v

sumImp :: [Int] -> Int
sumImp [] = 0
sumImp (h:t)
   | odd h = h + t'
   | otherwise = t'
   where t' = sumImp t
totalLen :: [String] -> Int
totalLen [] = 0
totalLen (h:t)
   | head h == 'A' = length h + t'
   | otherwise = t'
   where t' = totalLen t
```

LAB 3

```haskell
import Data.Char

palindrom :: String -> Bool
palindrom s = s == reverse s

voc :: String -> Int
voc "" = 0
voc (h : t)
   | elem h "aeiouAEIOU" = 1 + t'
   | otherwise = t'
   where t' = voc t

-- voc s = sum[1 | c <- s, e `elem` "aeiouAEIOU"]

nrVocale :: [String] -> Int
nrVocale [] = 0
nrVocale (h : t)
   | palindrom h = voc h + t'
   | otherwise = t'
   where t' = nrVocale t
-- nrVocale l = sum[voc s | s <- l, s==reverse s]

f :: Int -> [Int] -> [Int]
f a [] = []
f a (h : t)
   | even h = [h, a] ++ t'
   | otherwise = h : t'
   where t' = f a t

divizori :: Int -> [Int]
divizori n = [i |i <- [1 .. n], n `mod` i == 0]

listadiv :: [Int] -> [[Int]]
listadiv l = [divizori n | n <- l]
```

```haskell
inIntervalComp :: Int -> Int -> [Int] -> [Int]
inIntervalComp a b l = [x| x <- l, x >= a && x <= b]

inIntervalRec :: Int -> Int -> [Int] -> [Int]
inIntervalRec a b [] = []
inIntervalRec a b (h : t)
   | h >= a && h <= b = h : t'
   | otherwise = t'
   where t' = inIntervalRec a b t

pozitiveRec :: [Int] -> Int
pozitiveRec [] = 0
pozitiveRec (h : t)
   | h > 0 = 1 + t'
   | otherwise = t'
   where t' = pozitiveRec t

pozitiveComp :: [Int] -> Int
pozitiveComp l = sum[1 | x <- l, x > 0]

pozitiiImpareComp :: [Int] -> [Int]
pozitiiImpareComp l = [ poz | (elem, poz) <- zip l [0..], odd
elem]

pAuxRec :: [Int] -> Int -> [Int]
pAuxRec [] _ = []
pAuxRec (h:t) index
   | odd h = index : t'
   | otherwise = t'
   where t' = pAuxRec t (index+1)

pozitiiImpareRec :: [Int] -> [Int]
pozitiiImpareRec l = pAuxRec l 0
```

```haskell
-- importurile se pun la inceputul fisierului
-- mersi

multiDigitsRec :: String -> Int
multiDigitsRec "" = 1
multiDigitsRec (h : t)
   | isDigit h = digitToInt h * t'
   | otherwise = t'
   where t' = multiDigitsRec t

multiDigitsComp :: String -> Int
multiDigitsComp s = product[ digitToInt c | c <- s, isDigit c]
```

LAB 4

```haskell
factori :: Int -> [Int]
factori x = [d | d <- [1..x], x `mod` d == 0]

prim :: Int -> Bool
prim x = factori x == [1, x]

prim2 :: Int -> Bool
prim2 x = length(factori x) == 2

numerePrime :: Int -> [Int]
numerePrime n = [x | x <- [2..n], prim x]

myzip3 :: [a] -> [b] -> [c] -> [(a, b, c)]
myzip3 a b [c] = []
myzip3 a [] c = []
myzip3 [] b c = []
myzip3 (a1 : a2) (b1 : b2) (c1 : c2) = [(a1, b1, c1)] ++ myzip3 a2
b2 c2
-- myzip3 l1 l2 l3 = [(x, y, z) | i <- [0.. (min (min (length l1)
(length l2)) (length l3)- 1,)], let x = l1 !! i, let y = l2 !! i,
let z = l3 !! i]


firstEl :: [(a, b)] -> [a]
firstEl = map fst
-- firstEl = map (\(x, y) -> x)


sumList :: [[Int]] -> [Int]
sumList = map sum

prel2 :: [Int] -> [Int]
prel2 = map (\x -> if even x then div x 2 else 2*x)
```

```haskell
listaSiruri :: Char -> [String] -> [String]
listaSiruri ch = filter (\str -> elem ch str)

-- 9
impare :: [Int] -> [Int]
impare x = filter odd x
patrate :: [Int] -> [Int]
patrate = map(\x -> x*x)
patrateImpare :: [Int] -> [Int]
patrateImpare l = patrate (impare l)

-- 10
patratePozImpare lst = map (\(x1,x2) -> x1 * x1)(filter(\(x1,x2)
-> odd x2) (zip lst [1..]))

-- 11
elimConsoane :: String -> String
elimConsoane str = filter f str where f x = elem x "aeiouAEIOU"
numaiVocale :: [String] -> [String]
numaiVocale = map (\str -> elimConsoane str)

-- 12
mymap :: (a -> b) -> [a] -> [b]
mymap f [] = []
mymap f (h : t) = f h : mymap f t

-- am testat sa vad daca merge bine
sumList2 :: [[Int]] -> [Int]
sumList2 = mymap sum

myfilter :: (a -> Bool) -> [a] -> [a]
myfilter f [] = []
myfilter f (h : t) = if f h then [h] ++ myfilter f t
                     else myfilter f t
```

LAB 5

```haskell
-- 1
sumOdd :: [Int] -> Int
sumOdd l = foldl (+) 0 (map (^ 2) (filter odd l))

-- 2
allTrue :: [Bool] -> Bool
allTrue = foldr (&&) True

-- 3
allVerifies :: (Int -> Bool) -> [Int] -> Bool
allVerifies f l
 | length (filter f l) == length l = True
 | otherwise = False

allVerifiesFold :: (Int -> Bool) -> [Int] -> Bool
allVerifiesFold f l = foldr (&&) True (foldr (\x xs -> f x : xs)
[] l)

-- 4
anyVerifies :: (Int -> Bool) -> [Int] -> Bool
anyVerifies f l
 | length (filter f l) > 0 = True
 | otherwise = False

anyVerifiesFold :: (Int -> Bool) -> [Int] -> Bool
anyVerifiesFold f l = foldr (+) 0 (foldr (\x xs -> if f x then 1 :
xs else 0 : xs) [] l) > 0

-- 5
mapFoldr :: (a -> b) -> [a] -> [b]
mapFoldr _ [] = []
mapFoldr f l = foldr (\x xs -> f x : xs) [] l
```

```haskell
filterFoldr :: (a -> Bool) -> [a] -> [a]
filterFoldr _ [] = []
filterFoldr f l = foldr (\x xs -> if f x then x : xs else xs) [] l

-- 6
listToInt :: [Integer] -> Integer
listToInt l = foldl (\a b -> a * 10 + b) 0 l

-- 7
-- a
rmChar :: Char -> String -> String
rmChar ch = filter (/= ch)

-- b
rmCharsRec :: String -> String -> String
rmCharsRec [] l = l
rmCharsRec (h : t) s2 = rmCharsRec t (rmChar h s2)

-- c
rmCharsFold :: String -> String -> String
rmCharsFold s1 = foldr (\x xs -> if x `elem` s1 then xs else x :
xs) []
```

LAB 6

```haskell
-- 1
data Fruct = Mar String Bool | Portocala String Int

-- a)
ePortocalaDeSicilia :: Fruct -> Bool
ePortocalaDeSicilia (Mar tip viermi) = False
ePortocalaDeSicilia (Portocala tip felii) = tip `elem` ["Tarocco",
"Moro", "Sanguinello"]

-- b)
listaFructe =
 [ Mar "Ionatan" False,
   Portocala "Sanguinello" 10,
   Portocala "Valencia" 22,
   Mar "Golden Delicious" True,
   Portocala "Sanguinello" 15,
   Portocala "Moro" 12,
   Portocala "Tarocco" 3,
   Portocala "Moro" 12,
   Portocala "Valencia" 2,
   Mar "Golden Delicious" False,
   Mar "Golden" False,
   Mar "Golden" True
 ]

getFellii :: Fruct -> Int
getFellii (Mar _ _) = 0
getFellii (Portocala _ f) = f

nrFeliiSicilia :: [Fruct] -> Int
nrFeliiSicilia [] = 0
nrFeliiSicilia (x : xs) =
 if ePortocalaDeSicilia x
   then getFellii x + nrFeliiSicilia xs
```

```haskell
   else nrFeliiSicilia xs

-- c)
hasViermi :: Fruct -> Bool
hasViermi (Portocala _ _) = False
hasViermi (Mar _ v) = v

nrMereViermi :: [Fruct] -> Int
nrMereViermi [] = 0
nrMereViermi (x : xs) = if hasViermi x then 1 + nrMereViermi xs
else nrMereViermi xs

-- 2
type NumeA = String
type Rasa = String
data Animal = Pisica NumeA | Caine NumeA Rasa
 deriving (Show)

-- a)
vorbeste :: Animal -> String
vorbeste (Pisica _) = "Meoow!"
vorbeste (Caine _ _) = "Woof!"

-- b)
rasa :: Animal -> Maybe String
rasa (Pisica _) = Nothing
rasa (Caine _ rasa) = Just rasa

-- 3
data Linie = L [Int]
 deriving (Show)

data Matrice = M [Linie]
 deriving (Show)
-- a)
```

```haskell
matriceaMea = M [L [1, 2, 3], L [4, 5], L [2, 3, 6, 8], L [8, 5, 3]]

-- getSumaPeLinie :: Linie -> Int
-- getSumaPeLinie (L []) = 0
-- getSumaPeLinie (L list) = sum list

getSumaLinii :: Matrice -> [Int]
getSumaLinii (M []) = []
getSumaLinii (M (L x : xs)) = sum x : getSumaLinii (M xs)

verifica :: Matrice -> Int -> Bool
verifica (M m) n = and (foldr (\x y -> if x == n then True : y else False : y) [] (getSumaLinii (M m)))

-- b
getLiniiDeLungimeN :: Matrice -> Int -> Matrice
getLiniiDeLungimeN (M m) n = M (filter (\(L l) -> length l == n) m)

getCateNegativePeLinii :: Matrice -> [Int]
getCateNegativePeLinii (M m) = map (\(L l) -> length (filter (<= 0) l)) m

doarPozN :: Matrice -> Int -> Bool
doarPozN (M m) n = sum (getCateNegativePeLinii (getLiniiDeLungimeN (M m) n)) == 0

-- c)
corect :: Matrice -> Bool
corect (M []) = True
corect (M (L h1 : [])) = True
corect (M (L h1 : L h2 : t)) = length h1 == length h2 && corect (M (L h2 : t))
```

LAB 7

```haskell
import Control.Arrow (ArrowChoice (right))

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

```
evalExp (a :*: b) = evalExp a * evalExp b

-- 1.3
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))

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

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

-- 1.4
expToArb :: Expr -> Tree
expToArb (Const x) = Lf x
expToArb (a :+: b) = Node Add (expToArb a) (expToArb b)
expToArb (a :*: b) = Node Mult (expToArb a) (expToArb b)

-- 2.1
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
```

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

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

 -- c
 fromList = foldr (uncurry insert) empty

-- 2.2
newtype PairList k v = PairList {getPairList :: [(k, v)]}

instance Collection PairList where
 empty = PairList []
 singleton key value = PairList [(key, value)]
 insert key value (PairList l) = insert key value (delete key
(PairList l))
 toList = getPairList
 clookup key (PairList l) = Main.clookup key (PairList l)
 delete key (PairList l) = PairList $ filter (\(k, v) -> k /= key)
l

-- 2.3
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

instance Collection SearchTree where
 empty = Empty
 singleton k value = BNode Empty k (Just value) Empty
```

```
 insert k v Empty = singleton k v
 insert k v (BNode leftTree key val rightTree)
   | key < k = BNode leftTree key val (insert k v rightTree)
   | key > k = BNode (insert k v leftTree) key val rightTree
   | otherwise = BNode leftTree key val rightTree
 toList Empty = []
 toList (BNode leftTree key Nothing rightTree) = toList leftTree
++ toList rightTree
 toList (BNode leftTree key val rightTree) = (key, case val of {
Just a -> a }) : toList leftTree ++ toList rightTree
 clookup k Empty = Nothing
 clookup k (BNode leftTree key val rightTree)
   | k == key = val
   | k < key = clookup k leftTree
   | otherwise = clookup k rightTree
 delete k (BNode leftTree key val rightTree)
   | k == key = BNode leftTree key Nothing rightTree
   | k < key = delete k leftTree
   | otherwise = delete k rightTree
```

LAB 8

```haskell
-- 1
data Punct = Pt [Int]

data Arb = Vid | F Int | N Arb Arb
 deriving (Show)

class ToFromArb a where
 toArb :: a -> Arb
 fromArb :: Arb -> a

-- a
instance Show Punct where
 show (Pt []) = "()"
 show (Pt l) =
   "("
     ++ show (head l)
     ++ concat
       [ b : show a
         | (a, b) <- zip (tail l) [',', ',' ..]
       ]
     ++ ")"

-- b
getPunctToList :: Punct -> [Int]
getPunctToList (Pt []) = []
getPunctToList (Pt (x : xs)) = x : getPunctToList (Pt xs)

instance ToFromArb Punct where
 fromArb Vid = Pt []
 fromArb (F a) = Pt [a]
 fromArb (N st dr) = Pt (getPunctToList (fromArb st) ++
getPunctToList (fromArb dr))
```

```haskell
 toArb (Pt []) = Vid
 toArb (Pt [a]) = F a
 toArb (Pt (x : xs)) = N (F x) (toArb (Pt xs))

-- 2
data Geo a = Square a | Rectangle a a | Circle a
 deriving (Show)

class GeoOps g where
 perimeter :: (Floating a) => g a -> a
 area :: (Floating a) => g a -> a

-- a
instance GeoOps Geo where
 perimeter (Square l) = 4 * l
 perimeter (Rectangle lMic lMare) = 2 * lMic + 2 * lMare
 perimeter (Circle r) = 2 * pi * r

 area (Square l) = l ^ 2
 area (Rectangle lMic lMare) = lMic * lMare
 area (Circle r) = pi * (r ^ 2)

-- b
instance (Floating l, Eq l) => Eq (Geo l) where
 a == b = perimeter a == perimeter b
```

LAB 10

```haskell
newtype Identity a = Identity a

data Pair a = Pair a a

data Constant a b = Constant b

data Two a b = Two a b

data Three a b c = Three a b c

data Three' a b = Three' a b b

data Four a b c d = Four a b c d

data Four'' a b = Four'' a a a b

data Quant a b = Finance | Desk a | Bloor b

instance Functor Identity where
 fmap f (Identity a) = Identity (f a)

instance Functor Pair where
 fmap f (Pair a b) = Pair (f a) (f a)

instance Functor (Constant a) where
 fmap f (Constant b) = Constant (f b)

instance Functor (Two a) where
 fmap f (Two a b) = Two a (f b)

instance Functor (Three a b) where
 fmap f (Three a b c) = Three a b (f c)
```

```haskell
instance Functor (Three' a) where
 fmap f (Three' a b c) = Three' a (f b) (f b)

instance Functor (Four a b c) where
 fmap f (Four a b c d) = Four a b c (f d)

instance Functor (Four'' a) where
 fmap f (Four'' a b c d) = Four'' a a a (f d)

instance Functor (Quant a) where
 fmap f Finance = Finance
 fmap f (Desk a) = Desk a
 fmap f (Bloor b) = Bloor (f b)

data LiftItOut f a = LiftItOut (f a)

data Parappa f g a = DaWrappa (f a) (g a)

data IgnoreOne f g a b = IgnoringSomething (f a) (g b)

data Notorious g o a t = Notorious (g o) (g a) (g t)

data GoatLord a = NoGoat | OneGoat a | MoreGoats (GoatLord a)
(GoatLord a) (GoatLord a)

data TalkToMe a = Halt | Print String a | Read (String -> a)

instance Functor f => Functor (LiftItOut f) where
 fmap f (LiftItOut fa) = LiftItOut (fmap f fa)

instance (Functor f, Functor g) => Functor (Parappa f g) where
 fmap f (DaWrappa fa ga) = DaWrappa (fmap f fa) (fmap f ga)

instance Functor g => Functor (IgnoreOne f g a) where
```

```
 fmap f (IgnoringSomething fa gb) = IgnoringSomething fa (fmap f
gb)

instance Functor g => Functor (Notorious g o a) where
 fmap f (Notorious go ga gt) = Notorious go ga (fmap f gt)

instance Functor GoatLord where
 fmap f NoGoat = NoGoat
 fmap f (OneGoat a) = OneGoat (f a)
 fmap f (MoreGoats a b c) = MoreGoats (fmap f a) (fmap f b) (fmap
f c)

instance Functor TalkToMe where
 fmap f Halt = Halt
 fmap f (Print s a) = Print s (f a)
 fmap f (Read g) = Read (f . g)
```

LAB 11

```haskell
-- 1
data List a
 = Nil
 | Cons a (List a)
 deriving (Eq, Show)

instance Functor List where
 fmap f Nil = Nil
 fmap f (Cons a l) = Cons (f a) (fmap f l)

myConcat :: List a -> List a -> List a
myConcat a Nil = a
myConcat Nil a = a
myConcat (Cons a l) b = Cons a (myConcat l b)

instance Applicative List where
 pure a = Cons a Nil
 f <*> Nil = Nil
 Nil <*> f = Nil
 Cons f fs <*> (Cons a l) = Cons (f a) (myConcat (fmap f l) (fs
<*> Cons a l))

-- 2
data Cow = Cow
 { name :: String,
   age :: Int,
   weight :: Int
 }
 deriving (Eq, Show)

noEmpty :: String -> Maybe String
noEmpty s
 | not (null s) = Just s
 | otherwise = Nothing
```

```haskell
noNegative :: Int -> Maybe Int
noNegative n
 | n < 0 = Nothing
 | otherwise = Just n

cowFromString :: String -> Int -> Int -> Maybe Cow
cowFromString n a w
 | noEmpty n == Just n && noNegative a == Just a && noNegative w
== Just w = Just Cow {name = n, age = a, weight = w}
 | otherwise = Nothing

cowFromString2 :: String -> Int -> Int -> Maybe Cow
cowFromString2 n a w = Cow <$> noEmpty n <*> noNegative a <*>
noNegative w

-- 3
newtype Name = Name String deriving (Eq, Show)

newtype Address = Address String deriving (Eq, Show)

data Person = Person Name Address
 deriving (Eq, Show)

-- a
validateLength :: Int -> String -> Maybe String
validateLength n sir
 | n > length sir = Just sir
 | otherwise = Nothing

-- b
mkName :: String -> Maybe Name
mkName nume
 | validateLength 26 nume == Just nume = Just (Name nume)
 | otherwise = Nothing
```

```haskell
mkAddress :: String -> Maybe Address
mkAddress adr
 | validateLength 101 adr == Just adr = Just (Address adr)
 | otherwise = Nothing

-- c
mkPerson :: String -> String -> Maybe Person
mkPerson n a
 | mkName n == Just (Name n) && mkAddress a == Just (Address a) =
Just (Person (Name n) (Address a))
 | otherwise = Nothing

-- d
mkPerson2 :: String -> String -> Maybe Person
mkPerson2 n a = Person <$> Just (Name n) <*> Just (Address a)

mkAddress2 :: String -> Maybe Address
mkAddress2 adr = Address <$> validateLength 101 adr

mkName2 :: String -> Maybe Name
mkName2 n = Name <$> validateLength 26 n
```

LAB 12

```haskell
import Data.Monoid

elem1 :: (Foldable t, Eq a) => a -> t a -> Bool
elem1 x = foldr (\h t -> h == x || t) False

elem1' x xs = getAny $ foldMap (\h -> Any (h == x)) xs

null1 :: (Foldable t) => t a -> Bool
null1 = foldr (\x xs -> False) True

null1' xs = not (getAny $ foldMap (\x -> Any True) xs)

length1 :: (Foldable t) => t a -> Int
length1 = foldr (const (1 +)) 0

length1' xs = getSum $ foldMap (const 1) xs

toList1 :: (Foldable t) => t a -> [a]
toList1 = foldMap (: [])

fold1 :: (Foldable t, Monoid m) => t m -> m
fold1 = foldMap id

data Constant a b = Constant b

instance Foldable (Constant a) where
 foldMap f (Constant b) = f b

data Two a b = Two a b

instance Foldable (Two a) where
 foldMap f (Two a b) = f b

data Three a b c = Three a b c
```

```
instance Foldable (Three a b) where
 foldMap f (Three a b c) = f c

data Three' a b = Three' a b b

instance Foldable (Three' a) where
 foldMap f (Three' a b c) = f b <> f c

data Four' a b = Four' a b b b

instance Foldable (Four' a) where
 foldMap f (Four' a b c d) = f b <> f c <> f d

data GoatLord a = NoGoat | OneGoat a | MoreGoats (GoatLord a)
(GoatLord a) (GoatLord a)

instance Foldable GoatLord where
 foldMap f NoGoat = mempty
 foldMap f (OneGoat a) = f a
 foldMap f (MoreGoats a b c) = foldMap f a <> foldMap f b <>
foldMap f c
```

LAB 13

```
addM :: Maybe Int -> Maybe Int -> Maybe Int
addM mx my = let Just r = mx; Just b = my in Just (r + b)

addM' :: Maybe Int -> Maybe Int -> Maybe Int
addM' mx my = do
 x <- mx
 y <- my
 return (x + y)

cartesianProduct xs ys = do
 x <- xs
 y <- ys
 return (x, y)

prod f xs ys = do
 x <- xs
 y <- ys
 return f x y

myGetLine :: IO String
myGetLine = do
 x <- getChar
 if x == '\n'
   then return []
   else do
     xs <- myGetLine
     return (x : xs)

prelNo = sqrt

ioNumber = readLn >>= (\noin -> (putStrLn $ "Intrare\n" ++ show
noin) >> let noout = prelNo noin in (putStrLn $ "Iesire") >> print
noout)
```

```haskell
--- Monada Writer

newtype WriterS a = Writer {runWriter :: (a, String)}

instance Monad WriterS where
 return va = Writer (va, "")
 ma >>= k =
   let (va, log1) = runWriter ma
       (vb, log2) = runWriter (k va)
    in Writer (vb, log1 ++ log2)

instance Applicative WriterS where
 pure = return
 mf <*> ma = do
   f <- mf
   a <- ma
   return (f a)

instance Functor WriterS where
 fmap f ma = pure f <*> ma

tell :: String -> WriterS ()
tell log = Writer ((), log)

logIncrement :: Int -> WriterS Int
logIncrement x = do
 tell ("increment:" ++ show x ++ "\n")
 return (x + 1)

logIncrement2 :: Int -> WriterS Int
logIncrement2 x = do
 y <- logIncrement x
 logIncrement y

logIncrementN :: Int -> Int -> WriterS Int
```

```haskell
logIncrementN x n = do
 if n > 0
   then do
     a <- logIncrement x
     logIncrementN a (n - 1)
   else return x

data Person = Person {name :: String, age :: Int}

showPersonN :: Person -> String
showPersonN (Person name age) = "NAME:" ++ show name

showPersonA :: Person -> String
showPersonA (Person name age) = "AGE:" ++ show age

showPerson :: Person -> String
showPerson (Person name age) = showPersonN (Person name age) ++ ",
" ++ showPersonA (Person name age)

newtype Reader env a = Reader {runReader :: env -> a}

instance Monad (Reader env) where
 return :: a -> Reader env a
 return x = Reader (\_ -> x)

 (>>=) :: Reader env a -> (a -> Reader env b) -> Reader env b
 ma >>= k = Reader f
   where
     f env =
       let a = runReader ma env
        in runReader (k a) env

instance Applicative (Reader env) where
 pure :: a -> Reader env a
 pure = return
```

```haskell
 (<*>) :: Reader env (a -> b) -> Reader env a -> Reader env b
 mf <*> ma = do
   f <- mf
   a <- ma
   return (f a)

instance Functor (Reader env) where
 fmap :: (a -> b) -> Reader env a -> Reader env b
 fmap f ma = pure f <*> ma

ask :: Reader env env
ask = Reader id

mshowPersonN :: Reader Person String
mshowPersonN = do
 env <- ask
 return ("NAME: " ++ name env)

mshowPersonA :: Reader Person String
mshowPersonA = do
 env <- ask
 return ("AGE: " ++ show (age env))

mshowPerson :: Reader Person String
mshowPerson = do
 env <- ask
 return ("(NAME: " ++ name env ++ ",AGE: " ++ show (age env) ++
")")
```

MODEL EXAMEN

```haskell
data Point = Pt [Int]
 deriving Show

data Arb = Empty | Node Int Arb Arb
 deriving Show

class ToFromArb a where
 toArb :: a -> Arb
 fromArb :: Arb -> a

instance ToFromArb Point where
 toArb (Pt []) = Empty
 toArb (Pt (x:xs)) = Node x (toArb (Pt (filter (< x) xs)))  (toArb
(Pt (filter (>= x) xs)))
 fromArb Empty = Pt []
 fromArb (Node x st dr) = let Pt l1 = fromArb st
                              Pt l2 = fromArb dr
                          in Pt (l1 ++ [x] ++ l2)

-- Subiectul 2
-- cu selectie
getFromIntervalSel a b list = [x | x<-list, x>=a, x<=b]
-- monade
getFromInterval a b list = do
   x<-list
   if a <= x && x <= b then return x else []

-- Subiectul 3
newtype ReaderWriter env a = RW {getRW :: env-> (a,String)}

instance Monad (ReaderWriter env) where
 return va = RW (\_ -> (va,""))
 ma >>= k = RW f
     where f env = let (va, str1) = getRW ma env
```

```
                        (vb, str2)  = getRW (k va) env
                    in (vb, str1 ++ str2)

instance Applicative (ReaderWriter env) where
 pure = return
 mf <*> ma = do
   f <- mf
   va <- ma
   return (f va)

instance Functor (ReaderWriter env) where
 fmap f ma = pure f <*> ma
```