Pot sa fie absolut orice.

```
data Bool        = False | True
data Season      = Winter | Spring | Summer | Fall
data Shape       = Circle Float | Rectangle Float Float
data List      a = Nil | Cons a (List a)
data Nat         = Zero | Succ Nat
data Exp         = Lit Int | Add Exp Exp | Mul Exp Exp
data Tree      a = Empty | Leaf a | Branch (Tree a)(Tree a)
data Maybe     a = Nothing | Just a
data Pair    a b = Pair a b
data Either a b = Left a | Right b
```

Anotimpuri

```
data Season = Spring | Summer | Autumn | Winter

next :: Season -> Season
next Spring = Summer
next Summer = Autumn
next Autumn = Winter
next Winter = Spring

eqSeason :: Season -> Season -> Bool
eqSeason Spring Spring = True
eqSeason Summer Summer = True
eqSeason Autumn Autumn = True
eqSeason Winter Winter = True
eqSeason _      _      = False

showSeason :: Season -> String
Spring = " Spring "
Summer = " Summer "
Autumn = " Autumn "
Winter = " Winter "
```

```
toInt :: Season -> Int
toInt Winter = 0
toInt Spring = 1
toInt Summer = 2
toInt Autumn = 3

fromInt :: Int -> Season
fromInt 0 = Winter
fromInt 1 = Spring
fromInt 2 = Summer
fromInt 3 = Autumn

next :: Season -> Season
next x = fromInt ((toInt x + 1) `mod` 4)
```

Cercuri si dreptunghiuri

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius
           | Rectangle Width Height

area :: Shape -> Float
area (Circle r)      = pi * r^2
area (Rectangle w h) = w * h

eqShape :: Shape -> Shape -> Bool
eqShape (Circle r)      (Circle r2)       = (r == r2)
eqShape (Rectangle w h) (Rectangle w2 h2) = (w == w2) &&
                                            (h == h2)
eqShape _               _                 = False

isCircle :: Shape -> Bool
isCircle ( Circle r ) = True
isCircle _            = False

isRectangle :: Shape -> Bool
isRectangle ( Rectangle w h ) = True
isRectangle _                 = False

radius :: Shape -> Float
radius (Circle r) = r

width :: Shape -> Float
width (Rectangle w h) = w

height :: Shape -> Float
height (Rectangle w h) = h
```

```
area :: Shape -> Float
area (Circle r)      = pi * r^2
area (Rectangle w h) = w * h

area :: Shape -> Float
area s =
        if isCircle s then
                let
                        r = radius s
                in
                        pi * r^2
        else if isRectangle s then
                let
                        w = width s
                        h = height s
                in
                        w * h
        else error " impossible "
```

Expresii - curs

Expresii

```
data Exp = Lit Int
         | Add Exp Exp
         | Mul Exp Exp


evalExp :: Exp -> Int
evalExp (Lit n)   = n
evalExp (Add e f) = evalExp e + evalExp f
evalExp (Mul e f) = evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n)   = show n
showExp (Add e f) = par (showExp e ++ "+" ++ showExp f)
showExp (Mul e f) = par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s = "(" ++ s ++ ")"

e0, e1 :: Exp

e0 = Add (Lit 2) (Mul (Lit 3) (Lit 3))
e1 = Mul (Add (Lit 2) (Lit 3)) (Lit 3)

*Main> showExp e0
" (2+(3*3) ) "

* Main> evalExp e0
11

*Main> showExp e1
" ((2+3) *3) "

* Main> evalExp e1
15
```

Expresii - forma infixata

```
data Exp = Lit Int
          | Exp `Add` Exp
          | Exp `Mul` Exp

evalExp :: Exp -> Int
evalExp (Lit n)     = n
evalExp (e ‘Add‘ f) = evalExp e + evalExp f
evalExp (e ‘Mul‘ f) = evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n)     = show n
showExp (e `Add` f) = par (showExp e ++ "+" ++ showExp f)
showExp (e `Mul` f) = par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s = "(" ++ s ++ ")"

e0, e1 :: Exp

e0 = Lit 2 ‘Add‘ (Lit 3 ‘Mul‘ Lit 3)
e1 = (Lit 2 ‘Add‘ Lit 3) ‘Mul‘ Lit 3

*Main> showExp e0
" (2+(3*3) ) "

* Main> evalExp e0
11

*Main> showExp e1
" ((2+3) *3) "

* Main> evalExp e1
15
```

Expresii - cu operatori

```
data Exp = Lit Int
          | Exp :+: Exp
          | Exp :*: Exp

evalExp :: Exp -> Int
evalExp (Lit n)   = n
evalExp (e :+: f) = evalExp e + evalExp f
evalExp (e :*: f) = evalExp e * evalExp f

showExp :: Exp -> String
showExp (Lit n)   = show n
showExp (e :+: f) = par (showExp e ++ "+" ++ showExp f)
showExp (e :*: f) = par (showExp e ++ "*" ++ showExp f)

par :: String -> String
par s = "(" ++ s ++ ")"

e0, e1 :: Exp

e0 = Lit 2 :+: (Lit 3 :*: Lit 3)
e1 = (Lit 2 :+: Lit 3) :*: Lit 3

*Main> showExp e0
" (2+(3*3) ) "

* Main> evalExp e0
11

*Main> showExp e1
" ((2+3) *3) "

* Main> evalExp e1
15
```

Exercitii
Mere si portocale

```
-- The datatype 'Fruit'
data Fruit = Apple(String, Bool)
           | Orange(String, Int)

-- Some example Fruit
apple, apple', orange :: Fruit
apple  = Apple("Granny Smith", False) -- a Granny Smith
                                         apple with no worm
apple' = Apple("Braeburn", True)      -- a Braeburn apple
                                                with a worm
orange = Orange("Sanguinello", 10)    -- a Sanguinello
                                          with 10 segments

fruits :: [Fruit]
fruits = [Orange("Seville", 12),
          Apple("Granny Smith", False),
          Apple("Braeburn", True),
          Orange("Sanguinello", 10)]

-- This allows us to print out Fruit in the same way we
print out a list, an Int or a Bool.

instance Show Fruit where
  show (Apple(variety, hasWorm))   = "Apple("  ++ variety
                           ++ ", " ++ show hasWorm  ++ ")"
  show (Orange(variety, segments)) = "Orange(" ++ variety
                           ++ ", " ++ show segments ++ ")"


isBloodOrange :: Fruit -> Bool
isBloodOrange (Orange (b, c)) = if b `elem` ["Tarocco",
                                 "Moro", "Sanguinello"]
                                 then True
                                 else False
isBloodOrange _                = False
```

```
getNumberOfSlices :: Fruit -> Int
getNumberOfSlices (Apple (b, c))  = 0
getNumberOfSlices (Orange (b, c)) = c

bloodOrangeSegments :: [Fruit] -> Int
bloodOrangeSegments xs = foldl (+) 0 (getNumberOfSlices
                         `map` (filter isBloodOrange xs))

-- [Orange("Seville", 12),
--  Orange("Moro", 11),
--  Apple("Granny Smith", False),
--  Apple("Braeburn", True),
--  Orange("Sanguinello", 10)]

isAppleWithWorm :: Fruit -> Bool
isAppleWithWorm (Apple (_, True)) = True
isAppleWithWorm _                 = False

worms :: [Fruit] -> Int
worms xs = length (filter isAppleWithWorm xs)
```

Expresii

The following data type represents arithmetic expressions over a single variable:

```
data Expr = X                      -- variable
          | Const Int              -- integer constant
          | Expr :+: Expr          -- addition
          | Expr :-: Expr          -- substraction
          | Expr :*: Expr          -- multiplication
          | Expr :/: Expr          -- integer division
          | IfZero Expr Expr Expr -- conditional
                                                 expression
```

`IfZero p q r` represents the expression that would be written in Haskell as if `p == 0 then q else r`.

1. Write a function `eval :: Expr -> Int -> Int`, which given an expression and the value of the variable `X` returns the value of the expression. For example:

```
eval (X :+: (X :*: Const 2)) 3                   =  9
eval (X :/: Const 3) 7                           =  2
eval (IfZero (X :-: Const 3) (X:/:X) (Const 7)) 3  =  1
eval (IfZero (X :-: Const 3) (X:/:X) (Const 7)) 4  =  7
eval (Const 15 :-: (Const 7 :/: (X :-: Const 1))) 0 = 22
```

but both of the following should produce a divide-by-zero exception:

```
eval (Const 15 :-: (Const 7 :/: (X :-: Const 1))) 1
eval (X :/: (X :-: X)) 2
```

2. Write a function `protect :: Expr -> Expr` that protects against divide-by-zero exceptions by "guarding" all uses of division with a test for a zero-valued denominator. In this case the result should be `maxBound` (the maximum value of type `Int`, which is platform dependent). Do not attempt to simplify the result by omitting tests that appear to be unnecessary. For example:

```
protect (X :+: (X :*: Const 2)) = (X :+: (X :*: Const 2))

eval (protect (X :+: (X :*: Const 2))) 3 = 9

protect (X :/: Const 3)
           = IfZero (Const 3) (Const maxBound) (X :/:
                                                Const 3)

eval (protect (X :/: Const 3)) 7   = 2

eval (protect (X :/: (X :-: X))) 2 = maxBound
```

Afisarea

```
showExpr :: Expr -> String
showExpr X         = "X"
showExpr (Const n) = show n
showExpr (p :+: q) = "(" ++ showExpr p ++ "+" ++ showExpr
                     q ++ ")"
showExpr (p :-: q) = "(" ++ showExpr p ++ "-" ++ showExpr
                     q ++ ")"
showExpr (p :*: q) = "(" ++ showExpr p ++ "*" ++ showExpr
                     q ++ ")"
showExpr (p :/: q) = "(" ++ showExpr p ++ "/" ++ showExpr
                     q ++ ")"
showExpr (IfZero p q r) = "(if " ++ showExpr p ++ " = 0
                          then "++ showExpr q ++ "
                          else "++ showExpr r ++
                     ")"
```

Evaluarea

```
eval :: Expr -> Int -> Int
eval X v              = v
eval (Const n) _      = n
eval (p :+: q) v      = (eval p v)   +   (eval q v)
eval (p :-: q) v      = (eval p v)   -   (eval q v)
eval (p :*: q) v      = (eval p v)   *   (eval q v)
eval (p :/: q) v      = (eval p v) `div` (eval q v)
eval (IfZero p q r) v = if (eval p v) == 0 then eval q v
                                           else eval r v
```

Protectie pentru impartirea la 0

```
protect :: Expr -> Expr
protect X         = X
protect (Const n) = (Const n)
protect (p :+: q) = (protect p) :+: (protect q)
protect (p :-: q) = (protect p) :-: (protect q)
protect (p :*: q) = (protect p) :*: (protect q)
protect (p :/: q) = IfZero (protect q) (Const maxBound)
                      ((protect p) :/: (protect q))
protect (IfZero p q r) = IfZero (protect p) (protect q)
                           (protect r)
```

Arbori

Consider binary trees with `Int`-labelled nodes and leaves, defined as follows:

```
data Tree = Empty
          | Leaf Int
          | Node Tree Int Tree
```

and the following example of a tree:

```
t = Node (Node (Node (Leaf 1)
                2
                Empty)
        3
        (Leaf 4))
5
(Node Empty
        6
        (Node (Leaf 7)
                8
                (Leaf 9)))
```

Each label in a tree can be given an "address": this is the path from the root to the label, consisting of a list of directions:

```
data Direction = L | R
type Path = [Direction]
```

The empty path refers to the label at the root — in `t` above, the label `5`. A path beginning with L refers to a label in the left, or first, subtree, and a path beginning with R refers to the right, or second, subtree. Subsequent L/R directions in the list then refer to left/right subtrees of that subtree. So, for example, `[R,R,L]` is the address of the label 7 in `t`.

Verifica daca exista calea in arbore

```
present :: Path -> Tree -> Bool
present []    (Leaf n)     = True
present []    (Node _ n _) = True
present (L:p) (Node t _ _) = present p t
present (R:p) (Node _ _ t) = present p t
present _     _            = False
```

Returneaza valoarea unui nod din arbore

```
label :: Path -> Tree -> Int
label []    (Leaf n)     = n
label []    (Node _ n _) = n
label (L:p) (Node t _ _) = label p t
label (R:p) (Node _ _ t) = label p t
label _     _            = error "path absent"
```

Convertire abore in drumuri

```
toFTree' :: Tree -> FTree
toFTree' (Leaf n)       []    = n
toFTree' (Node t1 n t2) []    = n
toFTree' (Node t1 n t2) (L:p) = toFTree' t1 p
toFTree' (Node t1 n t2) (R:p) = toFTree' t2 p
toFTree' _              _     = error "path absent"
```

Construirea arborelui in oglinda

```
mirrorTree :: Tree -> Tree
mirrorTree Empty          = Empty
mirrorTree (Leaf n)       = Leaf n
mirrorTree (Node t1 n t2) = Node (mirrorTree t2) n
                                 (mirrorTree t1)
```

Evaluarea adancimii celei mai indepartate frunze din arbore

```
leafdepth :: Tree -> Int
leafdepth Empty        = 0
leafdepth (Leaf n)     = 1
leafdepth (Node t t') | d == 0 && d' == 0 = 0
                      | otherwise         = 1 + max d d'
                where
                d = leafdepth t
                d' = leafdepth t'
```

Cea mai indepartata frunza din arbore

```
deepest2 :: Tree -> [Int]
deepest2 Empty        = []
deepest2 (Leaf x)     = [x]
deepest2 (Node t t') | d > d'    = deepest2 t
                     | d < d'    = deepest2 t'
                     | otherwise = deepest2 t++deepest2 t'
                where
                d = leafdepth t
                d'= leafdepth t'
```

Propoziti

```
data Prop = X
          | F
          | T
          | Not Prop
          | Prop :|: Prop
```

Aproximarea expresiei

```
showProp :: Prop -> String
showProp X         = "X"
showProp F         = "F"
showProp T         = "T"
showProp (Not p)   = "(~" ++ showProp p ++ ")"
showProp (p :|: q) = "(" ++ showProp p ++ "|" ++ showProp
                           q ++ ")"
```

Evaluarea expresiei

```
eval :: Prop -> Bool -> Bool
eval X v         = v
eval F _         = False
eval T _         = True
eval (Not p) v   = not (eval p v)
eval (p :|: q) v = (eval p v) || (eval q v)
```

Simplificarea expresiei

```
simplify :: Prop -> Prop
simplify X         = X
simplify F         = F
simplify T         = T
simplify (Not p)   = negate (simplify p)
                        where
                                negate T = F
                                negate F = T
                                negate (Not p) = p
                                negate p = Not p
simplify (p :|: q) = disjoin (simplify p) (simplify q)
                        where
                                disjoin T p = T
                                disjoin F p = p
                                disjoin p T = T
                                disjoin p F = p
                                disjoin p q | p == q = p
                                            | otherwise =
                                                 p :|: q
```

Expresii

```
data Expr = Var String
          | Expr :+: Expr
          | Expr :*: Expr
```

Este Norm daca expresia este suma

```
isNorm :: Expr -> Bool
isNorm (a:+:b) = isNorm a && isNorm b
isNorm a       = isTerm a
```

Este Term daca expresia este constanta sau produs

```
isTerm :: Expr -> Bool
isTerm (Var x)   = True
isTerm (a :+: b) = False
isTerm (a :*: b) = isTerm a && isTerm b
```

Este normala daca expresia are proprietatea de distributivitate

```
norm :: Expr -> Expr
norm (Var v)   = Var v
norm (a :+: b) = norm a :+: norm b
norm (a :*: b) = norm a *** norm b
                        where
                                (a :+: b) *** c = (a ***
                                      c) :+: (b *** c)
                                a *** (b :+: c) = ( a ***
                                      b) :+: (a *** c)
                                a *** b         = a :*: b
```

Alte expresii

```
data Expr = X
          | Const Int
          | Neg Expr
          | Expr :+: Expr
          | Expr :*: Expr
```

Transforma expresia intr-o aproximare matematica

```
showExpr :: Expr -> String
showExpr X         = "X"
showExpr (Const n) = show n
showExpr (Neg p)   = "(-" ++ showExpr p ++ ")"
showExpr (p :+: q) = "(" ++ showExpr p ++ "+" ++ showExpr
                           q ++ ")"
showExpr (p :*: q) = "(" ++ showExpr p ++ "*" ++ showExpr
                           q ++ ")"
```

Evalueaza expresia dupa un X dat

```
evalExpr :: Expr -> Int -> Int
evalExpr X v         = v
evalExpr (Const n) _ = n
evalExpr (Neg p) v   = - (evalExpr p v)
evalExpr (p :+: q) v =   (evalExpr p v) + (evalExpr q v)
evalExpr (p :*: q) v =   (evalExpr p v) * (evalExpr q v)
```

Transforma expresia in Scriere Poloneza Inversa

```
rpn :: Expr -> [String]
rpn X         = ["X"]
rpn (Const n) = [show n]
rpn (Neg p)   = rpn p ++ ["-"]
rpn (p :+: q) = rpn p ++ rpn q ++ ["+"]
rpn (p :*: q) = rpn p ++ rpn q ++ ["*"]
```

Evalueaza o expresie in Scriere Poloneza Inversa

```
evalrpn :: [String] -> Int -> Int
evalrpn s n = the (foldl step [] s)
                where
                        step (x:y:ys) "+" = (y + x):ys
                        step (x:y:ys) "*" = (y * x):ys
                        step (x:ys)   "-" = (-x):ys
                        step ys "X"       = n:ys
                        step ys m | all (\c -> isDigit c
|| c == '-') m = (read m :: Int):ys
                                  | otherwise = error
                                        "ill-formed RPN"

                        the :: [a] -> a
                        the [x] = x
                        the xs  = error "ill-formed RPN"
```

Vectori

```
data Term = Vec Scalar Scalar
          | Add Term Term
          | Mul Scalar Term
```

Evaluarea expresiei

```
eva :: Term -> Vector
eva (Vec x y) = (x, y)
eva (Add t u) = add (eva t) (eva u)
eva (Mul x t) = mul x (eva t)
```

Printare expresia ca pereche vector

```
sho :: Term -> String
sho (Vec x y) = show (x, y)
sho (Add t u) = "(" ++ sho t ++ "+" ++ sho u ++ ")"
sho (Mul x t) = "(" ++ show x ++ "*" ++ sho t ++ ")"
```

Puncte

```
type Point = (Int, Int)
data Points = Rectangle Point Point
             | Union Points Points
             | Difference Points Points
```

Verificare daca un punct este intre 2 puncte

```
inPoints :: Point -> Points -> Bool
inPoints (x, y) (Rectangle (left,top) (right,bottom)) =
       left <= x && x <= right && top <= y && y <= bottom
inPoints p (Union ps qs) = inPoints p ps || inPoints p qs
inPoints p (Difference ps qs) = inPoints p ps && not
                                           (inPoints p qs)
```

Desenare puncte

```
showPoints :: Point -> Points -> [String]
showPoints (a, b) ps = [ makeline y | y <- [0..b] ]
                                where
                                        makeline y = [ if
     inPoints (x, y) ps then '*' else ' ' | x <- [0..a] ]
```