1. (3 puncte) Definim un tip de date pentru a reprezenta o rețea de mulțimi de puncte în plan:

```
type Punct = (Int, Int)
data Multime = X
             | Y
             | DX Int Multime
             | DY Int Multime
             | U Multime Multime
```

Planul este centrat în punctul $(0, 0)$. Prima coordonată a unui punct (coordonata x) reprezintă distanța pe orizontală de la origine iar a doua (coordonata y) reprezintă distanța pe verticală. Prin convenție, coordonatele x cresc spre dreapta, în timp ce coordonatele y cresc în sus.

Constructorul *X* reprezintă mulțimea punctelor de pe axa x, adică punctele care au coordonata y zero:

$$\cdots(-2,0)(-1,0)(0,0)(1,0)(2,0)\cdots$$

Constructorul *Y* reprezintă mulțimea punctelor de pe axa y, adică punctele care au coordonata x zero:

$$\cdots(0,-2)(0,-1)(0,0)(0,1)(0,2)\cdots$$

Constructorul DX dx p, unde dx este un întreg și p e o mulțime de puncte, reprezintă punctele p deplasate la dreapta cu dx. De exemplu, DX 2 Y are ca rezultat axa y deplasată cu două unități la dreapta:

$$\cdots(2,-2)(2,-1)(2,0)(2,1)(2,2)\cdots$$

Observație 1: DX dx X și *X* denotă aceeași mulțime de puncte, deoarece prin deplasarea axei x pe orizontală se obține tot axa x.

Observație 2: Un punct (x, y) aparține lui DX dx p dacă și numai dacă punctul $(x - dx, y)$ aparține lui p.

Constructorul DY dy p, unde dy este un întreg și p e o mulțime de puncte, reprezintă punctele p deplasate în sus cu dx. De exemplu, DY 3 X are ca rezultat axa *X* deplasată cu două unități în sus:

$$\cdots(-2,3)(-1,3)(0,3)(1,3)(2,3)\cdots$$

Observație 1: DY dy Y și *Y* denotă aceeași mulțime de puncte, deoarece prin deplasarea axei y pe verticală se obține tot axa y.

Observație 2: Un punct (x, y) aparține lui DY dy p dacă și numai dacă punctul $(x, y - dy)$ aparține lui p.

Constructorul U p q, unde p și q sunt mulțimi de puncte, reprezintă reuniunea punctelor din p și q. De exemplu, U (U X Y) (U (DY 3 X) (DX 2 Y)) reprezintă mulțimea de puncte de forma

$$\cdots(-2,0)(-1,0)(0,0)(1,0)(2,0)\cdots\cdots(0,-2)(0,-1)(0,0)(0,1)(0,2)\cdots$$
$$\cdots(-2,3)(-1,3)(0,3)(1,3)(2,3)\cdots\cdots(2,-2)(2,-1)(2,0)(2,1)(2,2)\cdots$$

Cerințe:

(a) (1½ puncte) Scrieți o funcție apartine :: Punct -> Multime -> Bool care determină dacă un punct aparține unei mulțimi de puncte date. De exemplu:

```
apartine (3,0) X == True
apartine (0,1) Y == True
apartine (3,3) (DY 3 X) == True
apartine (2,1) (DX 2 Y) == True
apartine (3,0) (U X Y) == True
apartine (0,1) (U X Y) == True
apartine (3,3) (U (DY 3 X) (DX 2 Y)) == True
apartine (2,1) (U (DY 3 X) (DX 2 Y)) == True
apartine (3,0) (U (U X Y) (U (DX 2 Y) (DY 3 X))) == True
apartine (0,1) (U (U X Y) (U (DX 2 Y) (DY 3 X))) == True
apartine (3,3) (U (U X Y) (U (DX 2 Y) (DY 3 X))) == True
apartine (2,1) (U (U X Y) (U (DX 2 Y) (DY 3 X))) == True
apartine (1,1) X == False
apartine (1,1) Y == False
apartine (1,1) (DY 3 X) == False
apartine (1,1) (DX 2 Y) == False
apartine (1,1) (U X Y) == False
apartine (1,1) (U X Y) == False
apartine (1,1) (U (DY 3 X) (DX 2 Y)) == False
apartine (1,1) (U (DY 3 X) (DX 2 Y)) == False
apartine (1,1) (U (U X Y) (U (DX 2 Y) (DY 3 X))) == False
```

(b) (1½ puncte) Scrieți o funcție nrAxe :: Points -> Int care numără de câte ori apare X sau Y în descrierea unei mulțimi de puncte. Fiecare axă trebuie numărată câte o dată pentru fiecare apariție a sa. De exemplu:

```
nrAxe X == 1
nrAxe Y == 1
nrAxe (U X Y) == 2
nrAxe (U (DY 3 X) (DX 2 Y)) == 2
nrAxe (U (U X Y) (U (DX 2 Y) (DY 3 X))) == 4
nrAxe (U (U X Y) X) == 3
```

2. (3 puncte) Se consideră următorul tip algebric de date:

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

data Op = NEG | PLUS | TIMES

data Atom = AConst Int | AOp Op

type Polish = [Atom]
```

Cerințe:

(a) (1½ puncte) Să se scrie o funcție fp :: Expr -> Polish care asociază unei expresii aritmetice date scrierea ei în forma poloneză: o listă de Atomi, obținută prin parcurgerea în preordine a arborelui asociat expresiei (operațiile precedând reprezentărilor operanzilor). Exemple:

- forma poloneză a expresiei $5 * 3$ este $* 5 3$
  fp (Const 5 :*: Const 3) = [AOp TIMES, AConst 5, AConst 3]
- forma poloneză a expresiei $-(7 * 3)$ este $- * 7 3$
  fp (Neg (Const 7 :*: Const 3)) = [AOp Neg, AOp TIMES, AConst 7, AConst 3]
- forma poloneză a expresiei $(5 + -3) * 17$ este $* + 5 - 3 17$
  fp ((Const 5 :+: Neg (Const 3)) :*: Const 17)
  = [AOp TIMES, AOp PLUS, AConst 5, AOp NEG, AConst 3, AConst 17]
- forma poloneză a expresiei $(15 + (7 * (2 + 1))) * 3$ este $* + 15 * 7 + 2 1 3$
  fp ((Const 15 :+: (Const 7 :*: (Const 2 :+: Const 1))) :*: Const 3)
  = [AOp TIMES, AOp PLUS, AConst 15, AOp TIMES, AConst 7,
  AOp PLUS, AConst 2, AConst 1, AConst 3]

(b) (1½ puncte) Definiți o funcție rfp :: Polish -> Maybe Expr astfel încât rfp . fp = Just . id

3. (2 puncte) Introducem un tip de date ce reprezinta o colectie de puncte (o tabela).

```
type Point =(Int, Int)

data Points = Rectangle Point Point
            Union Points Points
            Difference Points Points
```

Tabela incepe cu punctul (0,0) stanga jos. Constructorul Rectangle selecteaza toate punctele dintr-o forma rectangulara. De pilda, Rectangle (0,0) (2,1) da colturile din stanga jos si dreapta sus ale unui dreptunghi si include punctele (0,0) ; (1,0) ;(2,0) ; (0,1) ; (1,1) ; (2,1)

Union combina doua colectii de puncte iar Difference contine acele puncte care sunt in prima colectie dar nu sunt in a doua.

(a) (1 punct) Scrieti o functie perimeter care calculeaza perimetrul celui mai mic dreptunghi care cuprinde complet o colectie de puncte.

```
perimeter :: Points -> Int
```

(b) (1 punct) Scrieti o functie distance care calculeaza distanta dintre doua colectii de puncte ca reprezentand distanta intre coltul dreapta-sus al dreptunghiului minimal care cuprinde prima colectie si coltul stanga-jos al dreptunghiului minimal care cuprinde cea de-a doua colectie.

```
distance :: Points -> Points -> Int
```

4. (3 puncte) Introducem un tip de date ce reprezinta o expresie booleana formată din literali / variabile (Lit), negație (Not), conjuncție (And), disjuncție (Or) și implicație (:->:), in care conjuncțiile si disjuncțiile au un număr arbitrar de termeni.

```
data Exp = Lit String
         | Not Exp
         | And [Exp]
         | Or [Exp]
         | Exp :->: Exp
  deriving (Show)
```

Un atom este fie un literal (Lit), fie negația unui literal. O expresie este în formă normală disjunctivă dacă este o disjuncție de conjuncții de atomi.

Să se scrie o funcție care dată fiind o expresie are ca rezultat forma normală disjunctivă a acelei expresii.

```
dnf :: Exp -> Exp
dnf ((Lit "a" :->: Lit "b") :->: Lit "c")
  = Or [And [Lit "a",Not (Lit "b")],And [Lit "c"]]
dnf (Lit "a" :->: (Lit "b" :->: Lit "c"))
  = Or [And [Not (Lit "b")],And [Lit "c"],And [Not (Lit "a")]]
```

Indicație—puteți folosi următoarele identități:

- $a \to b \equiv \neg a \vee b$
- $\neg\neg a = a$
- $\neg \bigwedge(a_1, \ldots, a_n) = \bigvee(\neg a_1, \ldots, \neg a_n)$
- $\neg \bigvee(a_1, \ldots, a_n) = \bigwedge(\neg a_1, \ldots, \neg a_n)$
- $\bigwedge(a_1, \ldots, a_i, \bigwedge(b_1, \ldots, b_m), a_{i+1}, \ldots, a_n) = \bigwedge(a_1, \ldots, a_n, b_1, \ldots, b_m)$
- $\bigvee(a_1, \ldots, a_i, \bigvee(b_1, \ldots, b_m), a_{i+1}, \ldots, a_n) = \bigvee(a_1, \ldots, a_n, b_1, \ldots, b_m)$
- $\bigwedge(a_1, \ldots, a_i, \bigvee(b_1, \ldots, b_m), a_{i+1}, \ldots, a_n) = \bigvee(\bigwedge(b_1, a_1, \ldots, a_n), \ldots, \bigwedge(b_m, \ldots, a_n))$