6. Instanțiați clasa `Ord` pentru funcții Haskell care iau un argument numeric, astfel încât o funcție este "mai mică" decât alta dacă valoarea ei este mai mică decât a celeilalte funcții **în cel puțin unul** dintre numerele întregi între 1 și 10.
Soluție:
```
instance (Num a, Enum a, Ord b) => Ord (a -> b) where
  f <= g = or (zipWith (<=) (map f [1..10]) (map g [1..10]))
```
Notă: `Enum a` nu era cerut.

6. Evidențiați o posibilă instanță a clasei Haskell de mai jos:
```
class MyClass c where
  f :: c a -> a
```
Soluție:
```
instance MyClass [] where
  f = head
```

6. Scrieți o instanță posibilă a clasei de mai jos, conținând o implementare **neconstantă** a funcției `f`.

```
class MyClass c where
    f :: Num a => c a -> c a -> c a
```

Soluție:

```
instance MyClass Maybe where
    f (Just x) (Just y) = Just $ x + y
    f _ _ = Nothing
```

6. Supraîncărcați în Haskell operatorul de comparație pe liste, astfel încât o listă să fie mai mică sau egală cu alta, dacă toate elementele din prima listă sunt mai mici sau egale cu toate elementele din a doua. Spre exemplu, `[2, 3, 1] <= [5, 4, 3]`, dar nu avem că `[2, 4] <= [3, 5]`.

Soluție:

```
instance Ord a => Ord [a] where
    xs <= ys = maximum xs <= minimum ys
```

6. Scrieți o instanță posibilă a clasei de mai jos, conținând o implementare **neconstantă** a funcției `f`.

```
class MyClass c where
    f :: Ord a => c a -> c a -> c Bool
```

Soluție:

```
instance MyClass Maybe where
    f (Just x) (Just y) = Just $ x <= y
    f _ _ = Nothing
```

6. Supraîncărcați în Haskell operatorii `(+)` și `(*)` pentru valori booleene, pentru a surprinde operațiile de *sau*, respectiv *și* logic.
Soluție:
```
instance Num Bool where
  (+) = (||)
  (*) = (&&)
```

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste un alt tip `v`, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.
Instanțiați această clasă pentru tipul `data Triple a = T a a a`
Soluție:
```
class Ended t where frontEnd ::  t v -> v; backEnd ::  t v -> v
instance Ended Triple where

  frontEnd (T x _ _) = x
  backEnd (T _ _ x) = x
```

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste un alt tip `v`, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.
Instanțiați această clasă pentru tipul `data NestedL a = A a | L [NestedL a]`
Soluție:

```
class Ended t where frontEnd ::  t v -> v; backEnd ::  t v -> v
instance Ended NestedL where

  frontEnd (A a) = a; frontEnd (L l) = frontEnd $ head l
  backEnd (A a) = a; backEnd (L l) = backEnd $ last l
```

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste un alt tip `v`, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.
Instanțiați această clasă pentru tipul `data Pair a = MakePair a a`
Soluție:

```
class Ended t where frontEnd ::  t v -> v; backEnd ::  t v -> v
instance Ended Pair where

  frontEnd (MakePair x _) = x
  backEnd (MakePair _ x) = x
```

5. Scrieți definiția în Haskell a clasei `Ended` care, pentru un tip colecție `t` construit peste un alt tip `v`, definește o funcție `frontEnd` care extrage primul element din colecție și o funcție `backEnd` care extrage ultimul element din colecție.
Instanțiați această clasă pentru tipul listă Haskell.
Soluție:

```
class Ended t where frontEnd ::  t v -> v; backEnd ::  t v -> v
instance Ended  []  where

  frontEnd = head
  backEnd = last sau head .  reverse
```

5. Instanțiați clasa `Show` pentru funcții Haskell care iau un argument numeric, astfel încât afișarea unei funcții `f` va produce afișarea rezultatelor aplicării funcției pe numerele de la 1 la 10. E.g. afișarea lui `(+1)` va produce: `234567891011`.
Soluție:

```
-# LANGUAGE FlexibleInstances #- -- nu este cerut în rezolvarea din examen
instance (Enum a, Num a, Show b) => Show (a -> b) where
         show f = concatMap (show .  f) [1..10]
-- Enum nu este cerut în rezolvarea din examen
```

5. Instanțiați clasa `Show` pentru funcții Haskell care iau un argument numeric, astfel încât afișarea unei funcții `f` va produce afișarea rezultatelor aplicării funcției pe numerele de la 1 la 10. E.g. afișarea lui `(+1)` va produce: `234567891011`.
Soluție:

```
-# LANGUAGE FlexibleInstances #- -- nu este cerut în rezolvarea din examen
instance (Enum a, Num a, Show b) => Show (a -> b) where
         show f = concatMap (show .  f) [1..10]
-- Enum nu este cerut în rezolvarea din examen
```

6. Instanțiați în Haskell clasa `Eq` pentru tripluri, considerând că `(a1, a2, a3)` este egal cu `(b1, b2, b3)` dacă `a1 == b1` și `a2 == b2`.
Soluție:

```
instance (Eq a, Eq b) => Eq (a, b, c) where (a1, a2, _) == (b1, b2, _) = (a1 == b1)
&& (a2 == b2)
```

6. Instanțiați în Haskell clasa `Ord` pentru tripluri (știind că `Eq` este deja instanțiată), considerând că `(a1, a2, a3)` este mai mic decât `(b1, b2, b3)` dacă `a1 < b1`.
Soluție:
```
instance Ord a => Ord (a, b, c) where (a1, _, _) < (b1, _, _) = a1 < b1
```

5. Instanțiați în Haskell clasa `Ord` pentru perechi. Ordinea perechilor va fi dată de compararea celui de-al doilea element din pereche. E.g. `(1, 2) > (2, 0)` (pentru că $2 > 0$).
Soluție:
NOTĂ: Pentru ca implementarea să compileze am folosit aici `MyOrd` și `#<=` în loc de `Ord` și `<=`, definite astfel: `class Eq a => MyOrd a where (#<=) :: a -> a -> Bool`. Soluția cerută, (dar cu `Ord` și `<=` în loc de `MyOrd` și `#<=`), era:
```
instance (Eq a, Ord b) => MyOrd (a, b) where
  (_, y1) #<= (_, y2) = y1 <= y2
```

5. Instanțiați în Haskell clasa `Ord` pentru liste. Ordonarea listelor va fi dată de compararea primului element din fiecare listă. E.g. `[1, 2, 3] < [2, 3, 4]` pentru că $1 < 2$.
Soluție:
NOTĂ: Pentru ca implementarea să compileze am folosit aici `MyOrd` și `#<=` în loc de `Ord` și `<=`, definite astfel: `class Eq a => MyOrd a where (#<=) :: a -> a -> Bool`. Soluția cerută, (dar cu `Ord` și `<=` în loc de `MyOrd` și `#<=`), era:
```
instance Ord a => MyOrd [a] where
  (h1:_) #<= (h2:_) = h1 <= h2
```

6. Supraîncărcați în Haskell operatorul de egalitate pentru funcții unare cu parametru numeric, astfel încât două funcții să fie considerate egale dacă valorile lor coincid în cel puțin 10 puncte din intervalul $1, \ldots, 100$.

Soluție:

```
instance (Num a, Enum a, Eq b) => Eq (a -> b) where
    f == g = length (filter id [f x == g x | x <- [1..100]]) >= 10
```

7. Supraîncărcați în Haskell afișarea funcțiilor care au ca parametru un număr, afișându-se valoarea funcției în punctul 0.
Soluție:
```
instance (Show b, Num a) => Show (a -> b) where show f = show (f 0)
```

7. Supraîncărcați în Haskell ordonarea funcțiilor care au ca parametru un număr, ordonând funcțiile după valoarea lor pentru argumentul `0`. Se consideră că operatorul de egalitate între astfel de funcții a fost deja definit.

Soluție:
```
instance (Num a, Ord b) => Ord (a -> b) where f > g = f 0 > g 0
```

5. Supraîncărcați în Haskell afișarea funcțiilor care au ca parametru un număr, afișându-se valoarea funcției în punctul 1.
Soluție:
```
instance (Show b, Num a) => Show (a -> b) where show f = show (f 1)
```