```
{-
    1. Să se găsească cuvintele care au lungimea cel puțin egală
cu 10 caractere în două moduri:
        1) folosind filter
        2) folosind list comprehensions
-}

findStringsLongerThanTenChars :: [String] -> [String]
findStringsLongerThanTenChars l = filter (\x -> length x >= 10) l

findStringsLongerThanTenChars2 :: [String] -> [String]
findStringsLongerThanTenChars2 l = [x | x <- l, length x >= 10]


{-
    2. Să se construiască o listă de perechi de tip (string,
lungime_string) în două moduri:
        1) folosind map
        2) folosind list comprehensions
-}

buildPairsStringLength :: [String] -> [(String, Int)]
buildPairsStringLength l = map (\x -> (x, length x)) l

buildPairsStringLength2 :: [String] -> [(String, Int)]
buildPairsStringLength2 l = [(x, length x) | x <- l]


{-
    3. Implementați, folosind obligatoriu list-comprehensions,
operații pe mulțimi:
    intersecție, diferență, produs cartezian. Utilizați ulterior
funcțiile definite anterior
    pentru a reprezenta reuniunea mulțimilor.
-}

setIntersection :: Eq a => [a] -> [a] -> [a]
setIntersection a b = [x | x <- a, x `elem` b]

setDiff :: Eq a => [a] -> [a] -> [a]
setDiff a b = [x | x <- a, x `notElem` b]

cartProduct :: [a] -> [b] -> [(a, b)]
cartProduct a b = [(x, y) | x <- a, y <- b]

setUnion :: Eq a => [a] -> [a] -> [a]
setUnion a b = a ++ setDiff b (setIntersection a b)


naturals = [0..]

naturals = iter 0
    where iter x = x : iter (x + 1)
```

```
> :t iterate
iterate :: (a -> a) -> a -> [a]

naturals = iterate (\x -> x + 1) 0  -- SAU
naturals = iterate (+ 1) 0


ones = repeat 1 -- [1, 1, 1, ..]
onesTwos = intersperse 2 ones -- [1, 2, 1, 2, ..]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs) -- sirul lui Fibonacci
powsOfTwo = iterate (* 2) 1 -- puterile lui 2
palindromes = filter isPalindrome [0..] -- palindroame
    where
    isPalindrome x = show x == reverse (show x) -- truc: reprezint
numarul ca String


f $ x = f x

length $ 3 : [1, 2] -- length (3 : [1, 2])

sum xs = foldl (+) 0 xs

sum = foldl (+) 0


(.) :: (b -> c) -> (a -> b) -> a -> c


> length (tail (zip [1,2,3,4] ("abc" ++ "d")))
> length $ tail $ zip [1,2,3,4] $ "abc" ++ "d"

> (length . tail . zip [1,2,3,4]) ("abc" ++ "d")
> length . tail . zip [1,2,3,4] $ "abc" ++ "d"


flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x

> :t map
map :: (a -> b) -> [a] -> [b]
> :t flip map
flip map :: [a] -> (a -> b) -> [b]


myIntersperse :: a -> [a] -> [a]
myIntersperse y = foldr (++) [] . map (: [y])
```

Sintaxa if:
Haskell
```
if a < 0 then
    if (a > 10)
        then a * a
        else 0
    else -1
```

Definirea unei funcții:
Haskell
```
-- cu `if .. else .. then`
sumList :: [Int] -> Int
sumList l = if null l then 0 else head l + sumList (tail l)

-- cu gărzi
sumList3 :: [Int] -> Int
sumList3 l
    | null l = 0
    | otherwise = head l + sumList3 (tail l)

-- cu `case .. of`
sumList4 :: [Int] -> Int
sumList4 l = case l of
    [] -> 0
    (x:xl) -> x + sumList4 xl

-- cu pattern matching (sintactic sugar pentru varianta cu `case` de mai sus)
sumList2 :: [Int] -> Int
sumList2 [] = 0
sumList2 (x:xl) = x + sumList2 xl
```

Funcționale:
```
-- map
map (\x -> x + 1) [1, 2, 3, 4]  -- [2, 3, 4, 5]
map (+ 1) [1, 2, 3, 4]  -- [2, 3, 4, 5]

-- filter
filter (\x -> mod x 2 == 0) [1, 2, 3, 3, 4, 5, 6]  -- [2, 4, 6]

-- foldl
reverse $ foldl (\acc x -> x : acc) [] [1, 2, 3, 4, 5]  -- [1, 2, 3, 4, 5]

-- foldr
foldr (\x acc -> x : acc) [] [1, 2, 3, 4, 5]  -- [1, 2, 3, 4, 5]
```
Observăm că, spre deosebire de Racket, în Haskell funcționalele foldl și foldr primesc funcții cu semnături diferite:

```
foldl :: (b -> a -> b) -> b -> [a] -> b (funcția ajutătoare primește acumulatorul și apoi elementul curent)
```

```
foldr :: (a -> b -> b) -> b -> [a] -> b (funcția ajutătoare
primește elementul curent și apoi acumulatorul)


Legări:
-- cu let
f a =
    let c = a
        b = a + 1
    in (c + b) -- let din Racket

g a =
    let c = a
        b = c + 1
    in (c + b) -- let* din Racket

h a =
    let c = b
        b = a + 1
    in (c + b) -- letrec din Racket, aici nu avem eroare datorită
evaluării lenește

-- cu where
f' a = (c + b)
    where
        c = a
        b = a + 1  -- let din Racket

g' a = (c + b)
    where
        c = a
        b = c + 1  -- let* din Racket

h' a = (c + b)
    where
        c = b
        b = a + 1  -- letrec din Racket, aici nu avem eroare
datorită evaluării lenește
```