6. Instanțiați clasa `Eq` pentru funcții Haskell care iau un argument numeric, astfel încât două funcții sunt "egale" dacă valoarea lor este egală pentru fiecare număr întreg între 1 și 10.

7. Construiți în Haskell funcția `perms :: [Char] -> [[Char]]` care primește o listă de caractere și întoarce fluxul șirurilor formate din aceste caractere, începând cu șiruri de lungime 1. De exemplu: `take 45 $ perms "abc"` întoarce `["a","b","c","aa","ba","ca", ...,"bc","cc","aaa","baa","caa","aba","bba", ..., "acc","bcc","ccc","aaaa","baaa","caaa","abaa","bbaa","cbaa"]`

Construiesc șirurile de lungime 1, apoi celelalte șiruri sunt șiruri deja în flux la care adaug unul dintre caracterele din alfaetul dat.

7. Implementați în Racket fluxul în care primele 3 elemente sunt 1, 2 și 3, iar fiecare dintre următoarele elemente este produsul dintre cele trei elemente anterioare.
Soluție:

```
(define (stream-zip3 f s1 s2 s3) (stream-cons (f (stream-first s1) (stream-first s2) (stream-first
s3))
(stream-zip3 f (stream-rest s1) (stream-rest s2) (stream-rest s3)) ))
(define superProducts (stream-cons 1 (stream-cons 2 (stream-cons 3
(stream-zip3 * superProducts
(stream-rest superProducts) (stream-rest (stream-rest superProducts)) ) ))) (stream->list (stream-take
superProducts 7))
```

4. Cum decurge, pas cu pas, evaluarea expresiei:

```
head $ ([1] ++ [2]) ++ [3]
```

presupunând cunoscută implementarea operatorului de concatenare:

```
[] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

Soluție:

```
head $ ([1] ++ [2]) ++ [3]      -- (1)
head $ (1 : ([] ++ [2])) ++ [3] -- (2)
head $ 1 : (([] ++ [2]) ++ [3]) -- (3)
1                               -- (4)
```

7. Implementați în Haskell, fără a utiliza recursivitate explicită, funcția `setD` care realizează diferența a două mulțimi `a` și `b` (`a \ b`) date ca liste (fără duplicate). Care este tipul funcției?
Soluție:
```
setD a b = [x | x <- a, not $ elem x b]
```
sau
```
setD a b = filter (not .  (flip elem) b) a
setD ::  Eq t => [t] -> [t] -> [t]
```

7. Implementați în Haskell, fără a utiliza recursivitate explicită, funcția `setN` care realizează intersecția a două mulțimi `a` și `b` date ca liste (fără duplicate). Care este tipul funcției?
Soluție:
```
setN a b = [x | x <- a, elem x b]
```
sau
```
setN a b = filter ((flip elem) b) a
setN ::  Eq t => [t] -> [t] -> [t]
```

6. Scrieți o funcție Haskell care elimină dintr-o listă elementele care apar de mai multe ori. E.g. `[1, 2, 3, 2, 3, 5] -> [1, 5]`.
Soluție:
```
 nodups [] = []
nodups l = head l :  nodups (filter (not .  (== head l)) $ tail l)
```

6. Scrieți o funcție Haskell care păstrează dintr-o listă doar valorile care apar de mai multe ori. E.g. `[1, 2, 3, 2, 3] -> [2, 3]`.
Soluție:
```
 dups [] = []
dups (h:t)
  | elem h t = h :  dups (filter (/= h) t)
  | otherwise = dups t
```

7. Care este fluxul `s` pentru care este adevărat:
`(take 10 $ zipWith (+) s (tail s)) == (take 10 $ (tail .  tail) s)`
Soluție:
Fibonacci

4. Scrieți în Haskell o funcție care realizează produsul cartezian a două mulțimi oarecare (liste) `A` și `B`. Utilizați facilitățile oferite de limbaj. Care este tipul funcției create?
Soluție:
```
cart a b = [(x, y) | x <- a, y <- b]
cart ::  [t] -> [t1] -> [(t, t1)]
```

5. Construiți în Haskell fluxul puterilor lui 2.
Soluție:
```
fluxus = 1 :  map (* 2) fluxus
```

6. Construiți în Haskell fluxul puterilor lui 4.
Soluție:
```
fluxus = 1 :  map (* 4) fluxus
```

7. Scrieți în Haskell o funcție care realizează produsul cartezian a două mulțimi oarecare (liste) `A` și `B`. Utilizați facilitățile oferite de limbaj. Care este tipul funcției create?
Soluție:
```
cart a b = [(x, y) | x <- a, y <- b]
cart ::  [t] -> [t1] -> [(t, t1)]
```

5. Câte dintre cele trei adunări se vor realiza în evaluarea expresiei Haskell de mai jos? Justificați!
```
 let selector x y z = x in selector (1 + 2) (2 + 3) (3 + 4)
```

Soluție:
Una – (1 + 2), pentru că evaluarea este leneșă și parametrii `y` și `z` nu sunt evaluați.

6. Folosiți list comprehensions pentru aproduce fluxul listelor formate din primii 5 multipli ai fiecărui număr natural:
`[[1,2,3,4,5],[2,4,6,8,10],[3,6,9,12,15],[4,8,12,16,20] ...] .`
Soluție:
```
[take 5 [m | m <- [n..], mod m n == 0] | n <- [1..]]
```

6. Folosiți list comprehensions pentru aproduce fluxul listelor de divizori pentru numerele naturale: `[[1], [1, 2], [1, 3], [1, 2, 4], [1, 5], [1, 2, 3, 6] ...] .`
Soluție:
```
[[d | d <- [1..n], mod n d == 0] | n <- [1..]]
```

Scrieți o funcție sumPairs care primește două liste de numere și returnează o listă care conține suma elementelor de pe aceeași poziție din cele două liste.

```
sumPairs :: [Int] -> [Int] -> [Int]
sumPairs [] _ = []
sumPairs _ [] = []
sumPairs (x:xs) (y:ys) = (x + y) : sumPairs xs ys
```

Implementați o funcție applyTwice care primește o funcție f și un argument x și returnează rezultatul aplicării funcției f de două ori pe x.

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

Definiți o funcție isPalindrome care verifică dacă un șir de caractere este palindrom.

```
isPalindrome :: String -> Bool
isPalindrome s = s == reverse s
```

Implementați o funcție mergeSort care sortează o listă de numere folosind algoritmul Merge Sort.

```
mergeSort :: Ord a => [a] -> [a]
mergeSort [] = []
mergeSort [x] = [x]
mergeSort xs =
  merge (mergeSort firstHalf) (mergeSort secondHalf)
  where
    merge [] ys = ys
    merge xs [] = xs
    merge (x:xs) (y:ys)
      | x <= y = x : merge xs (y:ys)
      | otherwise = y : merge (x:xs) ys
    (firstHalf, secondHalf) = splitAt (length xs `div` 2) xs
```

Scrieți o funcție factorial care calculează factorialul unui număr dat.

```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Definiți o funcție flatten care primește o listă de liste și returnează o singură listă cu toate elementele inițiale.

```
flatten :: [[a]] -> [a]
flatten [] = []
flatten (x:xs) = x ++ flatten xs
```

Implementați o funcție fibonacci care calculează al n-lea număr Fibonacci.

```
fibonacci :: Int -> Int
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

Scrieți o funcție isPrime care verifică dacă un număr dat este prim.

```
isPrime :: Int -> Bool
isPrime n
  | n <= 1 = False
  | otherwise = all (\x -> n `mod` x /= 0) [2..isqrt n]
  where
    isqrt = floor . sqrt . fromIntegral
```

Definiți o funcție lastElement care returnează ultimul element dintr-o listă.

```
lastElement :: [a] -> a
lastElement [x] = x
lastElement (_:xs) = lastElement xs
```

Implementați o funcție map' care primește o funcție și o listă și aplică funcția pe fiecare element din listă.

```
map' :: (a -> b) -> [a] -> [b]
map' _ [] = []
map' f (x:xs) = f x : map' f xs
```

Scrieți o funcție length' care calculează lungimea unei liste.

```
length' :: [a] -> Int
length' [] = 0
length' (_:xs) = 1 + length' xs
```

Definiți o funcție maximum' care returnează elementul maxim dintr-o listă de numere.

```
maximum' :: Ord a => [a] -> a
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

Implementați o funcție takeWhile' care primește o funcție predicat și o listă și returnează prefixul listei format din elementele care satisfac predicatul.

```
takeWhile' :: (a -> Bool) -> [a] -> [a]
takeWhile' _ [] = []
takeWhile' p (x:xs)
  | p x = x : takeWhile' p xs
  | otherwise = []
```

Scrieți o funcție everyNth care primește un număr n și o listă și returnează o nouă listă conținând doar elementele de pe pozițiile multiplu de n.

```
everyNth :: Int -> [a] -> [a]
everyNth n xs = [x | (i, x) <- zip [1..] xs, i `mod` n == 0]
```

Definiți o funcție isSorted care verifică dacă o listă de numere este sortată în ordine crescătoare.

```
isSorted :: Ord a => [a] -> Bool
isSorted [] = True
isSorted [_] = True
isSorted (x:y:xs) = x <= y && isSorted (y:xs)
```

Implementați o funcție rotate care primește un număr n și o listă și returnează lista rotită la stânga de n ori.

```
rotate :: Int -> [a] -> [a]
rotate _ [] = []
rotate n xs = drop n xs ++ take n xs
```

Scrieți o funcție groupBy' care primește o funcție de echivalență și o listă și grupează elementele consecutive echivalente conform funcției de echivalență.

```
groupBy' :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy' _ [] = []
```

```
groupBy' f (x:xs) = (x : takeWhile (f x) xs) : groupBy' f (dropWhile (f x) xs)
```

Definiți o funcție interleave care primește două liste și le intercalează elementele alternativ.

```
interleave :: [a] -> [a] -> [a]
interleave [] ys = ys
interleave xs [] = xs
interleave (x:xs) (y:ys) = x : y : interleave xs ys
```

Implementați o funcție partition care primește o funcție predicat și o listă și returnează o pereche de liste, una conținând elementele care satisfac predicatul și cealaltă conținând elementele care nu îl satisfac.

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition _ [] = ([], [])
partition p (x:xs)
  | p x = (x : ys, zs)
  | otherwise = (ys, x : zs)
  where
    (ys, zs) = partition p xs
```

Scrieți o funcție transpose care primește o listă de liste și returnează matricea transpusă.

```
transpose :: [[a]] -> [[a]]
transpose [] = []
transpose ([]:_) = []
transpose rows = map head rows : transpose (map tail rows)
```

Definiți o funcție isSublist care verifică dacă o listă este sublista unei alte liste.

```
isSublist :: Eq a => [a] -> [a] -> Bool
isSublist _ [] = False
isSublist xs ys
  | take (length xs) ys == xs = True
  | otherwise = isSublist xs (tail ys)
```

Implementați o funcție insertionSort care sortează o listă de numere folosind algoritmul Insertion Sort.

```
insertionSort :: Ord a => [a] -> [a]
```

```
insertionSort xs = foldr insert [] xs
  where
    insert x [] = [x]
    insert x (y:ys)
      | x <= y = x : y : ys
      | otherwise = y : insert x ys
```

Scrieți o funcție isEven care verifică dacă un număr dat este par.

```
isEven :: Int -> Bool
isEven n = n `mod` 2 == 0
```

Definiți o funcție nub care elimină duplicații consecutive dintr-o listă.

```
nub :: Eq a => [a] -> [a]
nub [] = []
nub [x] = [x]
nub (x:y:xs)
  | x == y = nub (y:xs)
  | otherwise = x : nub (y:xs)
```

Implementați o funcție splitAt' care primește un index n și o listă și returnează o pereche de liste, una conținând primele n elemente și cealaltă conținând restul.

```
splitAt' :: Int -> [a] -> ([a], [a])
splitAt' n xs = (take n xs, drop n xs)
```