```haskell
-- Geraldo de Medeiros Galvão Neto
-- EE1

-- 1.

vendas :: Int -> Int
vendas 0 = 0
vendas 1 = 3
vendas 2 = 4
vendas 3 = 0
vendas 4 = 2

-- (a)

zeroVendas :: Int -> Int
zeroVendas semana = length [1 | sem <- [0..semana], (vendas sem) == 0]

-- (b)

zeroVendas' :: Int -> Int
zeroVendas' semana = length (concatMap (\x -> if x == 0 then[x] else []) (map
vendas [0..semana]))

-- (c)

zeroVendas'' :: Int -> Int
zeroVendas'' semana = length (foldr (\x acc -> if x == 0 then x:acc else acc) []
(map vendas [0..semana]))

-- 2.

-- (a)

type Cidade = String

data ClasseAerea = Economica | Executiva
data ClasseTrem = Primeira | Segunda

data Bilhete = Trem Cidade Cidade ClasseTrem | Onibus Cidade Cidade | Aereo Cidade
Cidade ClasseAerea

-- (b)

bils = [(Trem "a" "b" Primeira), Aereo "b" "c" Executiva]

bilheteString :: Bilhete -> (String, String)
bilheteString (Trem p d Primeira) = (p,d)
bilheteString (Trem p d Segunda) = (p,d)
bilheteString (Aereo p d Economica) = (p,d)
bilheteString (Aereo p d Executiva) = (p,d)
bilheteString (Onibus p d) = (p,d)
```

```haskell
valida :: [Bilhete] -> Bool
valida [] = True
valida [x] = True
valida (x:xs@(y:_))
| d1 == p2 = valida xs
| otherwise = False
where
(p1,d1) = bilheteString x
(p2,d2) = bilheteString y

-- 3.

data Nat = Zero | Succ Nat deriving (Eq, Show)

-- (a)

toInt :: Nat -> Int
toInt Zero = 0
toInt (Succ nat)= 1 + (toInt nat)

-- (b)

toNat :: Int -> Nat
toNat 0 = Zero
toNat n = Succ (toNat (n - 1))

-- (c)

soma :: Nat -> Nat -> Nat
soma (Succ nat1) (Succ nat2) = Succ (soma nat1 (Succ nat2))
soma Zero (Succ nat2) = Succ nat2
soma (Succ nat1) Zero = Succ nat1
soma Zero Zero = Zero

-- (d)

mult :: Nat -> Nat -> Nat
mult n1 n2 = toNat ((toInt n1) * (toInt n2))
```