```
module Testes where

--1
quantos :: (Eq a) => a -> [a] -> Int
quantos _ [ ] = 0
quantos x (y:ys) | x == y = 1 + (quantos x ys)
          | otherwise = (quantos x ys)

--2
maiores :: (Ord a) => [a] -> Int
maiores [x] = 0
maiores (x:y:xs) | x < y = 1 + maiores (x:xs)
          |otherwise=maiores (x:xs)

--3
positivos :: [Int] -> Int
positivos [ ] = 0
positivos (x:xs) | x > 0 = 1 + positivos xs
         | otherwise = positivos xs

--4
contaMai :: [Char] -> Int
contaMai [] = 0
contaMai (x:xs) | isUpper x = 1 + contaMai xs
         | otherwise = contaMai xs

isUpper :: Char -> Bool
isUpper x | (x >= 'A') && (x<='Z') = True
      |otherwise = False

--5
maiores2 :: (Ord a) => [a] -> [a]
maiores2 [x] = [ ]
maiores2 (x:y:xs) | x < y = (y:(maiores2 (x:xs)))
         |otherwise= maiores2 (x:xs)

--6
positivos2 :: [Int] -> [Int]
positivos2 [] = []
positivos2 (x:xs) | x > 0 = (x:(positivos2 xs))
         |otherwise =positivos2 xs

--7
impares :: [Int] -> [Int]
impares [] = []
impares (x:xs) | odd x = (x:(impares xs))
         | otherwise = impares xs

--8
maior :: [Int] -> Int
maior [x] = x
maior (x:y:xs) | x > y = maior (x:xs)
               | otherwise = maior (y:xs)

--9
soConsoantes :: String -> String
```

```
soConsoantes [ ] = [ ]
soConsoantes (x:xs) | (vogal x) = soConsoantes xs
            | otherwise = x:(soConsoantes xs)

vogal :: Char -> Bool
vogal x  | (x == 'a' ) = True
      | (x == 'e' ) = True
      | (x == 'i' ) = True
      | (x == 'o' ) = True
      | (x == 'u' ) = True
      |otherwise=False

--10
ordenada :: [Int] -> Bool
ordenada [ ] = True
ordenada [x]= True
ordenada (x:y:xs) | x < y = ordenada (y:xs)
         | otherwise =False

--11
pares :: [Int] -> Int
pares [ ] = 0
pares (x:xs) | even x = 1 + pares xs
      |otherwise =pares xs

--12
contaAlg :: [Char] -> Int
contaAlg [] = 0
contaAlg (x:xs) | isDigit x = 1 + contaAlg xs
         |otherwise = contaAlg xs

isDigit :: Char -> Bool
isDigit x  = (x >= '0') && (x<='9')

--13
type Rectangulo = (Int,Int)

    -- a
quadrados :: [Rectangulo] -> Int
quadrados [ ] = 0
quadrados ((a,b):cs) | (a == b) = 1 + quadrados cs
            | otherwise = quadrados cs

    --b
areaTotal :: [Rectangulo] -> Int
areaTotal [] = 0
areaTotal ((a,b):cs) = (a*b) + areaTotal cs

--14
descomprime :: [(a,Int)] -> [a]
descomprime [ ] = [ ]
descomprime ((a,b):cs) | b>0 = a:(descomprime ((a,b-1):cs))
                           |otherwise = descomprime cs

--15
remove :: [a] -> Int -> [a]
```

```
remove [] _ = []
remove (a:b:c) d | d/=0 = a: ( remove(b:c) (d-1))
                 | otherwise = (b:c)

--16
copia  ::  [a]  ->  [Int]  ->  [a]
copia [] [] = []
copia _ [] = []
copia l(a:bs) = head(drop a l) : copia l bs



--17
replicate2 :: Int -> a -> [a]
replicate2 0 x = []
replicate2 x y = y:replicate (x-1) y

--18
gama :: Int -> Int -> [Int]
gama x y | y >= x = x:(gama (x+1) y)
              | otherwise = []


--19
intercala :: a -> [a] -> [a]
intercala _ [] =[]
intercala _ [x] = [x]
intercala k (x:xs) = x:k:(intercala k xs)

--20
iguais :: [Int] -> [Int] -> Bool
iguais [] [] = True
iguais (x:xs) (y:ys) | (length (x:xs) == length (y:ys) ) &&
(x==y) = (iguais xs ys)
                     |otherwise = False

--21
init2 :: [a] -> [a]
init2 [] = []
init2 [x] = []
init2 (x:xs) = x:(init xs)

--22
splitAts :: Int -> [a] -> ([a],[a])
splitAts _ [] = ([],[])
splitAts 0 l = ([],l)
splitAts x (y:ys) = (y:a , b)
                where (a,b) = splitAts (x-1) ys

--23
type Matriz = [Linha]
type Linha = [Float]

   --a
maxMat :: Matriz -> Float
maxMat [] = 0
```

```
maxMat [x] = maximum x
maxMat (x:y:xs) | (maximum x) < (maximum y) = maxMat (y:xs)
                    | otherwise = maxMat (x:xs)

--(b)
quantos1 :: Float -> Matriz -> Int
quantos1 a [] = 0
quantos1 a (x:y) = quantos a x + quantos1 a y

--(c)
ok :: Matriz -> Bool
ok [] = False
ok [x] = True
ok (x:y:xs) | length x == length y = ok (y:xs)
         | otherwise = False

--d)
zero :: Int -> Int -> Matriz
zero x 0 = []
zero x y = (poe0 x): (zero x (y-1))

poe0 :: Int -> Linha
poe0 x  | x > 0 = 0: (poe0 (x-1))
         |otherwise = []


--24
compMaisLonga :: [String] -> Int
compMaisLonga [s] = length s
compMaisLonga (s:ss) = max (length s) (compMaisLonga ss)

--alternativa
comp2 :: [String] -> Int
comp2 [] = 0
comp2 s = maximum (map length s)

--25
nomesProp :: [String] -> Int
nomesProp [] = 0
nomesProp (h:t) | proprio h = 1 + (nomesProp t)
                  | otherwise = nomesProp t
proprio (c:_) = isUpper (c)

--alternativa
prop2 :: [String] -> Int
prop2 [] = 0
prop2 s = length (filter proprio s)

--26
conta :: Eq a => a -> [a] -> Int
conta x [] = 0
conta x (h:t) | x==h = 1 + conta x t
               | otherwise = conta x t

--alternativa
conta2 :: Eq a => a -> [a] -> Int
```

```
conta2 x [] = 0
conta2 x s = length ( filter (x==) s)

--27
areaQuadrados :: [(Int,Int)] -> Int
areaQuadrados [] = 0
areaQuadrados (h:t) | quadrado h = area h + areaQuadrados t
                    | otherwise = areaQuadrados t
quadrado (h,v) = h==v
area (h,v) = h*v

--alternativa
area2 :: [(Int,Int)] -> Int
area2 [] = 0
area2 l = sum (map area (filter quadrado l))

--28
diferentes :: Eq a => [a] -> Bool
diferentes [] = True
diferentes [x] = True
diferentes (x:y:ys) | x/=y && (diferentes (x:ys)) && (diferentes
(y:ys))


--29
leq :: String -> String -> Bool
leq [] [] = False
leq [] _ = True
leq _ [] = True
leq (x:xs) (y:ys) | x < y =True
                  | x > y = False
                  |x == y = leq xs ys

--30
type TabTemp = [(Data,TempMin,TempMin)]
type Data = (Int,Int,Int)
type TempMin = Float
type TempMax = Float

   --a
medias :: TabTemp -> [(Data,Float)]
medias [] = []
medias ((a,b,c):xs) = (a, (( b+c)/2)):(medias xs)

   --b
minMin :: TabTemp -> Float
minMin [] =0
minMin [(a,b,c)] = b
minMin ((a,b,c):(d,e,f):xs) | b < e = minMin ((a,b,c):xs)
                            |otherwise = minMin ((d,e,f):xs)

--31
type Jornada = [Jogo]
type Jogo = ((Equipa,Golos),(Equipa,Golos))
type Equipa = String
type Golos = Int
```

```
   --a
golosMarcados :: Jornada -> Int
golosMarcados [] = 0
golosMarcados  (((a,b),(c,d)):xs) =  b + d + (golosMarcados xs)

   --b
pontos :: Jornada -> [(Equipa,Int)]
pontos [] = []
pontos (((a,b),(c,d)):xs) = ((a,(tabpt b d)):((c, (tabpt d
b))):(pontos xs))

tabpt :: Int -> Int -> Int
tabpt x y | (x - y) > 0 = 3
          | (x - y) < 0 = 0
          | (x - y) ==0 = 1

--32
type TabTemp2 = [(Data2,TempMin2,TempMin2,Precipacao2)]
type Data2 = (Int,Int,Int)
type TempMin2 = Float
type TempMax2 = Float
type Precipacao2 = Float

   --a
amplTerm :: TabTemp2 -> [(Data2,Float)]
amplTerm [] = []
amplTerm ((a,b,c,d):xs) = ((a, (c-b)):(amplTerm xs))

   --b
maxChuva :: TabTemp2 -> (Data2,Float)
maxChuva [] = ((0,0,0),0)
maxChuva [(a,b,c,d)] = (a , d)
maxChuva ((a,b,c,d):(e,f,g,h):xs) | d > h = maxChuva
((a,b,c,d):xs)
                                   |otherwise= maxChuva
((e,f,g,h):xs)

--33--33

type Tabela = [(Nom,Nota)]
type Nom = String
type Nota = Int

--a)

parte :: Tabela -> ([Nom],[Nom])
parte [] = ([],[])
parte l = (reprovados l, aprovados l)

reprovados :: Tabela -> [Nom]
reprovados [] = []
reprovados ((n,g):xs) | g<10 = n : reprovados xs
                      | otherwise = reprovados xs

aprovados :: Tabela -> [Nom]
```

```
aprovados [] = []
aprovados ((n,g):xs) | g>=10 = n : aprovados xs
                     | otherwise = aprovados xs


--b)

nota :: Nom -> Tabela -> Maybe Nota
nota _ [] = Nothing
nota l ((x,y):xs) | l==x = Just y
                  | otherwise = nota l xs



   --b
nota :: Num2 -> Tabela -> Maybe Nota
nota k ((x,y):xs) | k == x = Just (y)
                  |otherwise = nota k xs
nota _ []= Nothing

--34
splitAt2 :: Int -> [a] -> ([a],[a])
splitAt2 0 l = ([],l)
splitAt2 _ [] = ([],[])
splitAt2 y (x:xs) = (x:a , b)
                     where (a,b) = splitAt2 (y-1) xs

--35
maiorDoQue :: Int -> [Int] -> Maybe Int
maiorDoQue x [] = Nothing
maiorDoQue x (y:k:xs) | x == y = Just ( k)
                      |otherwise = maiorDoQue x (k:xs)


--36.
filtragem :: (a->Bool) -> [a] -> ([a],[a])
filtragem x (y:ys) = let (a,b) = filtragem x ys
                     in
                        if x y
                     then(y:a,b)
                     else (a,y:b)

--37.


data Avaliacao = A Float Float -- Teste, pratica
               | B Float
type Aluno = (Int, String, Avaliacao) -- Numero, Nome, Avaliacao
type Turma = [Aluno]
--a.

nota :: Avaliacao -> Maybe Float
nota (A y z) |y >= 8 && (0.7*y+0.3*z)>= 9.5 = Just (0.7*y+0.3*z)
             | otherwise = Nothing
nota (B x) | x>= 9.5 = Just x
         | otherwise = Nothing
```

```
--b.

pauta:: Turma -> [(String,String,String)]
pauta [] = []
pauta ((a,b,c):xs) |(nota c)>= Just 9.5 = ((show a), b,(show
(nota c))):pauta xs
               | otherwise = (((show a), b, "Rep" ):pauta xs)



--c.


melhorA :: Turma -> Maybe Int
melhorA [(a,b,c)] = Just a
melhorA ((a,b, (A x y)):(c,d,e):xs) | nota (A x y) >= nota e =
melhorA((a,b,(A x y)):xs)
                              |otherwise = melhorA ((c,d,e):xs)
melhorA ((a,b, (B x)):xs) = melhorA xs

--d.

finais :: Turma -> [(Int,Float)]
finais [] = []
finais ((a,b,c):xs) = ((a, calculaNota c): finais xs)

calculaNota :: Avaliacao -> Float
calculaNota (A y z) = (0.7*y+0.3*z)
calculaNota (B x) = x


--e.


aprovado :: Aluno -> Bool
aprovado (a,b,(A y x)) =  y>=8 && (0.7*y+0.3*x) >= 9.5
aprovado (a,b,(B x)) = x >= 9.5

--f.

stat:: Turma -> (Float,Float)
stat ((c,d,(A x y)):ls) = let (a,b) = stat ls
                           in if aprovado (c,d,(A x y))
                        then (a+1,b)
                           else (a,b)

stat ((c,d,(B x)):ls) = let (a,b) = stat ls
                     in if aprovado (c,d,(B x))
                        then (a,b+1)
                           else (a,b)

--38.

compMaisLonga2:: [String] -> Int
compMaisLonga2 [x] = length x
```

```
compMaisLonga2 l = maximum (map length l)


--39.

{-
nomesProprios :: [String] -> Int
nomesPorprios [] = 0
nomesProprios l = length (filter (proprio l) l)
-}

--40.

(!!) :: [a] -> Int -> a
l !! n = head (drop n l)

escolheIndice :: [a] -> Int -> a
escolheIndice l 0 = head l
escolheIndice (x:xs) y = escolheIndice xs (y-1)

--41.
{-
allBut :: [a] -> Int -> [a]
allBut l n = (a,tail b)
   where (a,b) = splitAt n l
-}

allBut2 :: [a] -> Int -> [a]
allBut2 [] x = []
allBut2 (x:xs) 1 = xs
allBut2 (x:xs) y = (x: allBut2 xs (y-1))


42
split :: (Num a) => a -> [a] -> ([a],[a])
split p [] = ([],[])
split 1 (x:xs) = ([x],xs)
split p (x:xs) = let (a,b) = split (p-1) xs
             in (x:a,b)
43
extractmultiples :: [Int] -> Int -> ([Int],[Int])
extractmultiples [] p = ([],[])
extractmultiples (x:xs) p = let (a,b) = extractmultiples xs p
                 in if (mod x p) == 0
               then (x:a,b)
                      else (a,x:b)

44
catMaybesz :: [Maybe a] -> [a]
catMaybesz [] = []
catMaybesz (Nothing:xs) = catMaybesz xs
catMaybesz ((Just x):xs) = x:(catMaybesz xs)

45
data Tree a = Leaf a
             |Fork (Tree a) (Tree a)
```

```
folhas:: Tree a -> [a]
folhas (Leaf a) = [a]
folhas (Fork esq dir) = folhas esq ++ folhas dir

altura :: Tree a -> Int
altura (Leaf x) = 1
altura (Fork esq dir) = 1 + (max (altura esq) (altura dir))
```

46

```
data Nat = Zero
     |Succ Nat

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