```
-- ********** Exercícios ********

-- Usando somente ranges e funções pré-definidas sobre
-- listas, defina funções para

-- 1. Retornar o segundo elemento de uma lista
segundo xs
   | length xs >= 2 = head (tail xs)
   | otherwise = error "list has not enough elements"

-- 2. Calcular quantos dígitos decimais têm um número
nroDigitos :: Int -> Int
nroDigitos n = length (show (abs n))

-- 3. Retornar o i-ésimo elemento de uma lista
esimo :: Int -> [Int] -> Int
esimo i ns
  | 0 <=i && i < length ns = head (drop i ns)
  | otherwise              = error "index out of bounds"

-- 4. Calcular a média aritmética de uma lista de números

-- 5. Ver se um número Integer é palíndromo. Um número é palíndromo se
--    resulta ser o mesmo quando lido de trás para frente.
--   (Dica: usar show e reverse)
palindromo :: Int -> Bool
palindromo n = nStr == reverse nStr
  where nStr = show n

-- 6. Verificar se todos os elementos de uma lista são iguais entre
--    eles. (Dica: usar maximum e minimum)

-- 7. Dado um inteiro n positivo, calcular o produto dos números
--    impares de 1 até n.

-- 8. Calcular o número de combinações de m elementos pegos de um
--    universo de n elementos. A fórmula das combinações é:
--    n! / m!(m-n)!
--    Tente que sua função realize o menor número de multiplicações


-- ************************************

-- *** Exercícios sobre compreensões ****

-- 9. Defina uma função que dada uma lista cheque se todos são
múltiplos de 5
todosMultiplos5 :: [Int] -> Bool
todosMultiplos5 ns = ns == [n | n <- ns, n `mod` 5 == 0]
-- todosMultiplos5 ns = [] == [n | n <- ns, n `mod` 5 /= 0]

-- 10. Escreva uma função que devolva uma lista em que os elementos
--     ímpares da lista de entrada aparecem triplicados.

-- 11. Sem usar maximum nem minimum, escrever uma função que checa
--     se uma lista está formada pela repetição de um único
```

```
--     elemento.

-- 12. Defina uma função que receba como entrada um inteiro n e uma
lista
--     ns de inteiros. A função deve retornar o número de vezes que n
ocorre
--     dentro da lista ns.

-- 13. Defina uma função que calcule o valor aproximado de Pi
--     utilizando a seguinte série de Leibniz
--     1 - 1/3 + 1/5 - 1/7 + 1/9 - ... = pi/4
--     A função recebe como argumento o número de termos a ser usado
--     para aproximar.

-- ****************************************

-- *** Exercícios sobre tuplas e compreensões

-- Considere as seguintes definições de tipos e funções

type ItemCompra = (String, Float)
type CarrinhoCompras = [ItemCompra]

nomeProduto :: ItemCompra -> String
nomeProduto (nm, _) = nm

preco :: ItemCompra -> Float
preco (_, p) = p

total :: CarrinhoCompras  -> Float
total itens = sum [ preco item | item <- itens]

-- Defina funções para

-- 14. Dado um carrinho e o nome de um item, retornar o total
--     no carrinho considerando somente itens com o nome dado.
totalItem :: String -> CarrinhoCompras -> Float
totalItem nm cs = sum [ pr | (nm1, pr) <- cs, nm == nm1 ]

-- 15. Calcular o total gasto com itens cujo valor unitário é inferior
ou
--     ou igual a 5.
totalInf5 :: CarrinhoCompras -> Float
totalInf5 itens = sum [ p | (n, p) <- itens, p <= 5 ]

-- 16. Calcular o total gasto com itens cujo valor unitário ultrapassa
--     100.

-- 17. Dado um carrinho de compras, retornar a lista dos
--     nomes dos itens mais caros
maisCaros :: CarrinhoCompras -> [String]
maisCaros itens = [ n | (n, p) <- itens, p == maiorPreco ]
  where maiorPreco = maximum [ p | (_, p) <- itens]


-- 18. Calcular o total considerando que, como oferta, a cada dois
itens com
```

```
--     nome "leite" há um desconto de 0.5 (5 centavos).
totalNaOferta :: CarrinhoCompras -> Float
totalNaOferta itens = total itens - desconto
  where desconto = fromIntegral (quantLeite `div` 2) * 0.5
        quantLeite =  sum [ 1 |("leite", _) <- itens ]
```