Stenio Longo Araújo
Gustavo Pereira Nunes



# EXERCÍCIOS RESOLVIDOS UTILIZANDO HASKELL

Série Cadernos Didáticos, v. 37

UESB
Universidade Estadual
do Sudoeste da Bahia

EDIÇÕES
UESB

**UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA (UESB)**

**Reitor**
Prof. Dr. Luiz Otávio de Magalhães

**Vice-Reitor**
Prof. Dr. Marcos Henrique Fernandes

**Pró-Reitora de Extensão e Assuntos Comunitários (PROEX)**
Profª Drª Gleide Magali Lemos Pinheiro

**Diretora da Edições UESB**
Me. Manuella Lopes Cajaíba

**Editor**
Yuri Chaves Souza Lima

**COMITÊ EDITORIAL**
**Presidente**
Profª Drª Gleide Magali Lemos Pinheiro (PROEX)
**Representantes dos Departamentos/Áreas de Conhecimento**
Profª Drª Alba Benemérita Alves Vilela (DS II)
Profª Drª Delza Rodrigues de Carvalho (DCSA)
Prof. Dr. Cezar Augusto Casotti (DS 1)
Prof. Dr. Flávio Antônio Fernandes Reis (DELL)
Prof. Dr. José Antonio Gonçalves dos Santos (DCSA)
Prof. Dr. José Rubens Mascarenhas de Almeida (DH)
Prof. Dr. Luciano Brito Rodrigues (DTRA)
Profª Drª Sylvana Naomi Matsumoto (DFZ)

**Representantes da Edições UESB**
Me. Manuella Lopes Cajaíba (Diretora)
Esp. Yuri Chaves Souza Lima (Editor)
Adm. Jacinto Braz David Filho (Revisor)
Dr. Natalino Perovano Filho (Portal de Periódicos)

**PRODUÇÃO EDITORIAL**
**Normalização Técnica**
Jacinto Braz David Filho
**Revisão de linguagem**
Yuri Chaves Souza Lima
**Editoração Eletrônica**
Jacinto Braz David Filho
**Capa**
Imagem disponível em: https://www.pngwing.com/pt/free-png-dzoge/download

Publicado em: 25/07/2025.

**Stenio Longo Araújo**
**Gustavo Pereira Nunes**

# EXERCÍCIOS RESOLVIDOS UTILIZANDO HASKELL

Série Cadernos Didáticos, v. 37

EDIÇÕES UESB
Vitória da Conquista – Bahia
2025

Copyright © 2025 by Autores.
Todos os direitos desta edição são reservados a Edições UESB.
A reprodução não autorizada desta publicação, no todo ou em parte,
constitui violação de direitos autorais (Lei 9.610/98).

Publicado conforme o Regulamento Da Publicação de Textos Didáticos na UESB

A692e

Araújo, Stenio Longo

Exercícios resolvidos utilizando Haskell. / Stenio Longo Araújo, Gustavo Pereira Nunes. - - Vitória da Conquista: Edições UESB, 2025. (Série Cadernos Didáticos, v. 37).

38 p.

ISBN 978-65-87106-89-2

1. Haskell (Linguagem de programação de computador). 2. Programação funcional (Computação). I. Nunes, Gustavo Pereira. II. T.

CDD: 005.13

Catalogação na fonte: Karolyne Alcântara Profeta — CRB 5/2134
Biblioteca Universitária Professor Antonio de Moura Pereira
UESB – Campus de Vitória da Conquista

Editora filiada à:

ASSOCIAÇÃO BRASILEIRA DAS EDITORAS UNIVERSITÁRIAS

ABEC BRASIL 40 Anos
Avançando na qualificação e inovação da editoria científica

EDIÇÕES UESB

Campus Universitário – Caixa Postal 95 – Fone: 77 3424-8716
Estrada do Bem-Querer, s/n – Módulo da Biblioteca, 1° andar
45031-900 – Vitória da Conquista – Bahia
https://www2.uesb.br/editora/ – E-mail: edicoesuesb@uesb.edu.br

# INTRODUÇÃO

Haskell é uma linguagem de programação funcional pura. Este material, desenvolvido a partir da monitoria da disciplina Programação Declarativa do Curso de Bacharelado em Ciência da Computação da UESB, tem como objetivo apresentar diversos exercícios resolvidos em Haskell visando auxiliar os discentes na compreensão da programação funcional.

Para maiores informações sobre a sintaxe e a semântica da linguagem de programação Haskell, consultar Acióly e Araújo (2008), Sá e Silva (2006) e Thompson (2011).

# EXERCÍCIOS

### 1. Elaborar uma função para calcular a área de um triângulo:

```
-------------------------------------------------------------------------------------------
area_triangulo :: Float -> Float -> Float
area_triangulo base altura = (base * altura) / 2.0

-------------------------------------------------------------------------------------------
resultado :: Bool -> String
resultado True  = "PASSOU"
resultado False = "FALHOU"

-------------------------------------------------------------------------------------------
testAreaTriangulo :: IO ()
testAreaTriangulo = do
    putStrLn $ "- area_triangulo 10 4 == 20: "     ++ resultado (area_triangulo 10 4 == 20)
    putStrLn $ "- area_triangulo 5 6 == 15: "      ++ resultado (area_triangulo 5 6 == 15)
    putStrLn $ "- area_triangulo 0 5 == 0: "       ++ resultado (area_triangulo 0 5 == 0)
    putStrLn $ "- area_triangulo 3.5 2 == 3.5: "   ++ resultado (area_triangulo 3.5 2 == 3.5)
    putStrLn $ "- area_triangulo 1 1 == 0.5: "     ++ resultado (area_triangulo 1 1 == 0.5)

-------------------------------------------------------------------------------------------
main :: IO ()
main = do putStrLn "Inicio dos testes..."
          testAreaTriangulo
          putStrLn "Fim dos testes..."
```

### 2. Elaborar uma função que recebe três inteiros e retorna o menor:

```
-------------------------------------------------------------------------------------------
menor :: Int -> Int -> Int -> Int
menor numero1 numero2 numero3
    | numero1 < numero2 && numero1 < numero3 = numero1
    | numero2 < numero3                      = numero2
    | otherwise                              = numero3

-------------------------------------------------------------------------------------------
resultado :: Bool -> String
resultado True  = "PASSOU"
resultado False = "FALHOU"

-------------------------------------------------------------------------------------------
testMenor :: IO ()
testMenor = do
    putStrLn $ "- menor 3 5 7 == 3: " ++ resultado (menor 3 5 7 == 3)
    putStrLn $ "- menor 9 2 8 == 2: " ++ resultado (menor 9 2 8 == 2)
    putStrLn $ "- menor 12 10 5 == 5: " ++ resultado (menor 12 10 5 == 5)
    putStrLn $ "- menor 4 4 4 == 4: " ++ resultado (menor 4 4 4 == 4)
    putStrLn $ "- menor 1 4 4 == 1: " ++ resultado (menor 1 4 4 == 1)
    putStrLn $ "- menor 4 3 4 == 3: " ++ resultado (menor 4 3 4 == 3)
    putStrLn $ "- menor 4 4 0 == 0: " ++ resultado (menor 4 4 0 == 0)
    putStrLn $ "- menor 0 (-1) 1 == -1: " ++ resultado (menor 0 (-1) 1 == -1)


-------------------------------------------------------------------------------------------
main :: IO ()
main = testMenor
```

**3. Elaborar uma função que recebe quatro inteiros e verifica se todos são iguais:**

```
-------------------------------------------------------------------------------------------
saoIguais :: Int -> Int -> Int -> Int -> Bool
saoIguais numero1 numero2 numero3 numero4 =
 numero1 == numero2 && numero2 == numero3 && numero3 == numero4

-------------------------------------------------------------------------------------------
resultado :: Bool -> String
resultado True  = "PASSOU"
resultado False = "FALHOU"

-------------------------------------------------------------------------------------------
testaSaoIguais :: IO ()
testaSaoIguais = do
 putStrLn "Testes de saoIguais"
 putStrLn $ "- saoIguais 1 1 1 1 == True: "     ++ resultado (saoIguais 1 1 1 1 == True)
 putStrLn $ "- saoIguais 0 0 0 0 == True: "     ++ resultado (saoIguais 0 0 0 0 == True)
 putStrLn $ "- saoIguais 9 9 9 8 == False: "    ++ resultado (saoIguais 9 9 9 8 == False)
 putStrLn $ "- saoIguais 4 5 4 4 == False: "    ++ resultado (saoIguais 4 5 4 4 == False)
 putStrLn $ "- saoIguais 2 2 3 2 == False: "    ++ resultado (saoIguais 2 2 3 2 == False)
 putStrLn $ "- saoIguais 7 6 5 4 == False: "    ++ resultado (saoIguais 7 6 5 4 == False)


-------------------------------------------------------------------------------------------
main :: IO ()
main = testaSaoIguais
```

**4. Elaborar uma função para calcular o XOR (ou exclusivo). Para maiores informações sobre essa função, consultar Gersting (2017).**

| Tabela XOR | | |
|---|---|---|
| **A** | **B** | **A XOR B** |
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | False |

```
-------------------------------------------------------------------------------------------
-- Versao 1
-------------------------------------------------------------------------------------------
xor :: Bool -> Bool -> Bool
xor a b
    | a == True && b == False = True
    | a == False && b == True = True
    | otherwise = False


-------------------------------------------------------------------------------------------
-- Versao 2
-------------------------------------------------------------------------------------------
xor' :: Bool -> Bool -> Bool
xor' a b = (a && not b) || (not a && b)
```

```
-----------------------------------------------------------------------------------------
resultado :: Bool -> String
resultado True  = "PASSOU"
resultado False = "FALHOU"

-----------------------------------------------------------------------------------------
testaXor :: IO ()
testaXor = do
    putStrLn $ "- xor True True == False: "   ++ resultado (xor True True == False)
    putStrLn $ "- xor True False == True: "   ++ resultado (xor True False == True)
    putStrLn $ "- xor False True == True: "   ++ resultado (xor False True == True)
    putStrLn $ "- xor False False == False: " ++ resultado (xor False False == False)
    putStrLn ""
    putStrLn $ "- xor' True True == False: "   ++ resultado (xor' True True == False)
    putStrLn $ "- xor' True False == True: "   ++ resultado (xor' True False == True)
    putStrLn $ "- xor' False True == True: "   ++ resultado (xor' False True == True)
    putStrLn $ "- xor' False False == False: " ++ resultado (xor' False False == False)

-----------------------------------------------------------------------------------------
main :: IO ()
main = testaXor
```

**5. Elaborar uma função para converter uma letra maiúscula para minúscula:**

```
import Data.Char

-----------------------------------------------------------------------------------------
toMin :: Char -> Char
toMin c
    | isLetter c && isUpper c = toLower c
    | otherwise = c

-----------------------------------------------------------------------------------------
resultado :: Bool -> String
resultado True  = "PASSOU"
resultado False = "FALHOU"

-----------------------------------------------------------------------------------------
testaToMin :: IO ()
testaToMin = do
   putStrLn $ "- toMin 'A' == 'a': " ++ resultado (toMin 'A' == 'a')
   putStrLn $ "- toMin 'Z' == 'z': " ++ resultado (toMin 'Z' == 'z')
   putStrLn $ "- toMin 'M' == 'm': " ++ resultado (toMin 'M' == 'm')
   putStrLn $ "- toMin 'a' == 'a': " ++ resultado (toMin 'a' == 'a')
   putStrLn $ "- toMin '1' == '1': " ++ resultado (toMin '1' == '1')
   putStrLn $ "- toMin '?' == '?': " ++ resultado (toMin '?' == '?')
   putStrLn $ "- toMin ' ' == ' ': " ++ resultado (toMin ' ' == ' ')

-----------------------------------------------------------------------------------------
main :: IO ()
main = testaToMin
```

**6. Elaborar uma função para somar os algarismos de um número inteiro N. Considerar que N ≥ 0 e N ≤ 999.**

```
-----------------------------------------------------------------------------------------
somaAlgarismo :: Int -> Int
somaAlgarismo numero
    | numero < 0 || numero >= 1000 = error "Numero invalido!!!"
    | otherwise = centena numero + dezena numero + unidade numero
    where
        centena :: Int -> Int
```

```
        centena numero = div numero 100

        dezena :: Int -> Int
        dezena numero = mod (div numero 10) 10

        unidade :: Int -> Int
        unidade numero = mod numero 10

-------------------------------------------------------------------------------------------
resultado :: Bool -> String
resultado True  = "PASSOU"
resultado False = "FALHOU"

-------------------------------------------------------------------------------------------
testaSomaAlgarismo :: IO ()
testaSomaAlgarismo = do
    putStrLn $ "- somaAlgarismo 123 == 6: " ++ resultado (somaAlgarismo 123 == 6)
    putStrLn $ "- somaAlgarismo 456 == 15: " ++ resultado (somaAlgarismo 456 == 15)
    putStrLn $ "- somaAlgarismo 789 == 24: " ++ resultado (somaAlgarismo 789 == 24)
    putStrLn $ "- somaAlgarismo 10 == 1: " ++ resultado (somaAlgarismo 10 == 1)
    putStrLn $ "- somaAlgarismo 99 == 18: " ++ resultado (somaAlgarismo 99 == 18)
    putStrLn $ "- somaAlgarismo 0 == 0: " ++ resultado (somaAlgarismo 0 == 0)

-------------------------------------------------------------------------------------------
main :: IO ()
main = testaSomaAlgarismo
```

### 7. Qual é o resultado da avaliação de *expr1* e *expr2*?

```
-------------------------------------------------------------------------------------------
(%%%) :: Int -> Int -> Int
(%%%) a b
     | a == b = b
     | a <= b = a + (a+1) %%% b

-------------------------------------------------------------------------------------------
func1 :: Int -> Int
func1 1 = 25
func1 n = func1 (n-1)

-------------------------------------------------------------------------------------------
func2 :: Int -> Int
func2 1 = 25
func2 n = n + func2 (n-1)

-------------------------------------------------------------------------------------------
func3 :: Int -> Int
func3 6 = 0
func3 n = n + func3 (n+1)

-------------------------------------------------------------------------------------------
main :: IO()
main = do let expr1 = func1 7 + func2 6 + 1 + func3 1
          let expr2 = 1 %%% 5 %%% 16
          putStrLn(show(expr1)) -- 86
          putStrLn(show(expr2)) -- 31
```

**8. Definir uma função para somar todos os pares de 1 até N, inclusive:**

```
-------------------------------------------------------------------------------------------
somaPar :: Int -> Int
somaPar limite
    | limite < 1 = 0
    | ehPar limite = limite + somaPar (limite-2)
    | otherwise = somaPar (limite-1)

-------------------------------------------------------------------------------------------
ehPar :: Int -> Bool
ehPar numero
    | mod numero 2 == 0 = True
    | otherwise = False

-------------------------------------------------------------------------------------------
resultado :: Bool -> String
resultado True  = "PASSOU"
resultado False = "FALHOU"

-------------------------------------------------------------------------------------------
testaEhPar :: IO ()
testaEhPar = do
    putStrLn $ "- ehPar 4 == True: " ++ resultado (ehPar 4 == True)
    putStrLn $ "- ehPar 5 == False: " ++ resultado (ehPar 5 == False)
    putStrLn $ "- ehPar 0 == True: " ++ resultado (ehPar 0 == True)
    putStrLn $ "- ehPar (-2) == True: " ++ resultado (ehPar (-2) == True)
    putStrLn $ "- ehPar (-3) == False: " ++ resultado (ehPar (-3) == False)

-------------------------------------------------------------------------------------------
testaSomaPar :: IO ()
testaSomaPar = do
    putStrLn $ "- somaPar 5 == 6: " ++ resultado (somaPar 5 == 6)
    putStrLn $ "- somaPar 6 == 12: " ++ resultado (somaPar 6 == 12)
    putStrLn $ "- somaPar 1 == 0: " ++ resultado (somaPar 1 == 0)
    putStrLn $ "- somaPar 0 == 0: " ++ resultado (somaPar 0 == 0)
    putStrLn $ "- somaPar 10 == 30: " ++ resultado (somaPar 10 == 30)
    putStrLn $ "- somaPar (-3) == 0: " ++ resultado (somaPar (-3) == 0)

-------------------------------------------------------------------------------------------
main :: IO ()
main = do
    testaEhPar
    putStrLn ""
    testaSomaPar
```

**9. Definir uma função para calcular x elevado a y. Considerar x e y inteiros positivos.**

```
-------------------------------------------------------------------------------------------
-- Versão 1
-------------------------------------------------------------------------------------------

exponencial :: Int -> Int -> Int
exponencial numero 1 = numero
exponencial numero expoente = numero * exponencial numero (expoente-1)

-------------------------------------------------------------------------------------------
-- Versão 2
-------------------------------------------------------------------------------------------

exponencial' :: Integer -> Integer -> Integer
exponencial' numero 1 = numero
exponencial' numero expoente = numero * exponencial' numero (expoente-1)

-------------------------------------------------------------------------------------------
```

```
-- Versão 3
-----------------------------------------------------------------------------------------

exponencial'' :: Integer -> Integer -> Integer
exponencial'' numero 1 = numero
exponencial'' numero expoente
  | expoente `mod` 2 == 0 = let metade = exponencial'' numero (expoente `div` 2)
                            in metade * metade
  | otherwise = numero * exponencial'' numero (expoente-1)

-----------------------------------------------------------------------------------------
resultado :: Bool -> String
resultado True  = "PASSOU"
resultado False = "FALHOU"

-----------------------------------------------------------------------------------------
testaExponencial :: IO ()
testaExponencial = do
    putStrLn "Testes da Versao 1"
    putStrLn $ "- exponencial 2 1 == 2: " ++ resultado (exponencial 2 1 == 2)
    putStrLn $ "- exponencial 2 3 == 8: " ++ resultado (exponencial 2 3 == 8)
    putStrLn $ "- exponencial 3 4 == 81: " ++ resultado (exponencial 3 4 == 81)
    putStrLn $ "- exponencial 5 2 == 25: " ++ resultado (exponencial 5 2 == 25)
    putStrLn $ "- exponencial 7 1 == 7: " ++ resultado (exponencial 7 1 == 7)
    putStrLn $ "- exponencial 1 100 == 1: " ++ resultado (exponencial 1 100 == 1)
    putStrLn $ "- exponencial 2 10 == 1024: " ++ resultado (exponencial 2 10 == 1024)
    putStrLn $ "- exponencial 10 5 == 100000: " ++ resultado (exponencial 10 5 == 100000)
    putStrLn ""
    putStrLn "Testes da Versao 2"
    putStrLn $ "- exponencial 2 1 == 2: " ++ resultado (exponencial' 2 1 == 2)
    putStrLn $ "- exponencial 2 3 == 8: " ++ resultado (exponencial' 2 3 == 8)
    putStrLn $ "- exponencial 3 4 == 81: " ++ resultado (exponencial' 3 4 == 81)
    putStrLn $ "- exponencial 5 2 == 25: " ++ resultado (exponencial' 5 2 == 25)
    putStrLn $ "- exponencial 7 1 == 7: " ++ resultado (exponencial' 7 1 == 7)
    putStrLn $ "- exponencial 1 100 == 1: " ++ resultado (exponencial' 1 100 == 1)
    putStrLn $ "- exponencial 2 10 == 1024: " ++ resultado (exponencial' 2 10 == 1024)
    putStrLn $ "- exponencial 10 5 == 100000: " ++ resultado (exponencial' 10 5 == 100000)
    putStrLn ""
    putStrLn "Testes de Versao 3"
    putStrLn $ "- exponencial 2 1 == 2: " ++ resultado (exponencial'' 2 1 == 2)
    putStrLn $ "- exponencial 2 3 == 8: " ++ resultado (exponencial'' 2 3 == 8)
    putStrLn $ "- exponencial 3 4 == 81: " ++ resultado (exponencial'' 3 4 == 81)
    putStrLn $ "- exponencial 5 2 == 25: " ++ resultado (exponencial'' 5 2 == 25)
    putStrLn $ "- exponencial 7 1 == 7: " ++ resultado (exponencial'' 7 1 == 7)
    putStrLn $ "- exponencial 1 100 == 1: " ++ resultado (exponencial'' 1 100 == 1)
    putStrLn $ "- exponencial 2 10 == 1024: " ++ resultado (exponencial'' 2 10 == 1024)
    putStrLn $ "- exponencial 10 5 == 100000: " ++ resultado (exponencial'' 10 5 == 100000)

-----------------------------------------------------------------------------------------
main :: IO ()
main = testaExponencial
```

**10. Elaborar uma função para gerar uma tabela verdade. Para maiores informações sobre lógica e tabela verdade, consultar Gersting (2017).**

```
import Text.Printf (printf)

-------------------------------------------------------------------------------------
(\/) :: Bool -> Bool -> Bool
(\/) p q = p || q

-------------------------------------------------------------------------------------
(/\) :: Bool -> Bool -> Bool
(/\) p q = p && q

-------------------------------------------------------------------------------------
formula1 :: Bool -> Bool -> Bool
formula1 p q = not ((not p) \/ (not q))

-------------------------------------------------------------------------------------
formula2 :: Bool -> Bool -> Bool
formula2 p q = not p \/ q

-------------------------------------------------------------------------------------
formula3 :: Bool -> Bool -> Bool
formula3 p q = not ((not p) /\ (not q))

-------------------------------------------------------------------------------------
gerarTabelaVerdade :: (Bool -> Bool -> Bool) -> String
gerarTabelaVerdade f = cabecalho ++ gerarlinhas f
  where cabecalho = printf "%-6s%-6s%-10s\n" "p" "q" "resultado"

-------------------------------------------------------------------------------------
gerarlinhas :: (Bool -> Bool -> Bool) -> String
gerarlinhas f = concat [ gerarlinha f v1 v2 | v1 <- [True, False], v2 <- [True, False] ]

-------------------------------------------------------------------------------------
gerarlinha :: (Bool -> Bool -> Bool) -> Bool -> Bool -> String
gerarlinha f v1 v2 = printf "%-6s%-6s%-10s\n" (mostrar v1) (mostrar v2) (mostrar (f v1 v2))

-------------------------------------------------------------------------------------
mostrar :: Bool -> String
mostrar True  = "V"
mostrar False = "F"

-------------------------------------------------------------------------------------
main :: IO ()
main = do putStrLn (gerarTabelaVerdade formula1)
          putStrLn (gerarTabelaVerdade formula2)
          putStrLn (gerarTabelaVerdade formula3)
```

**11. Os números de Fibonacci constituem uma sequência de números na qual os dois primeiros elementos são 0 e 1 e os demais, a soma dos dois elementos imediatamente anteriores na sequência (exercício adaptado da Questão nº 18 da prova do ENADE 2008 – CIÊNCIA DA COMPUTAÇÃO – BACHARELADO).**

**Considerando que a função Haskell abaixo não reutilize resultados previamente computados, quantas chamadas são feitas à função *fibo* para computar *fibo 5*?**

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

```
fibo 5
├── fibo 4
│   ├── fibo 3
│   │   ├── fibo 2
│   │   │   ├── fibo 1
│   │   │   └── fibo 0
│   │   └── fibo 1
│   └── fibo 2
│       ├── fibo 1
│       └── fibo 0
└── fibo 3
    ├── fibo 2
    │   ├── fibo 1
    │   └── fibo 0
    └── fibo 1
```

**15 chamadas recursivas para calcular *fibo 5*.**

**12. Considere as funções *f'* e *g'* abaixo. Qual será a saída de *f' 3*?**

```
-------------------------------------------------------------------------------------------
f' :: Int -> Int
f' 0 = 0
f' n = n + g' (n - 1)

-------------------------------------------------------------------------------------------
g' :: Int -> Int
g' 0 = 0
g' n = n + f' (n - 1)

-------------------------------------------------------------------------------------------
main :: IO ()
main =  do putStr "f' 3 = "
           print(f' 3) -- 6
```

**13. Qual é o resultado da expressão *h 27*? (exercício adaptado da Questão nº 23 da prova do ENADE 2014 – CIÊNCIA DA COMPUTAÇÃO – BACHARELADO).**

```
-------------------------------------------------------------------------------------------
main :: IO ()
main =  do putStr "h 27 = "
           print(h 27) -- 16

-------------------------------------------------------------------------------------------
h :: Int -> Int
h n
   | n <= 10 = 2*n
   | otherwise = h (h (div n 3))
```

**14. Considere a seguinte definição da função de *Ackermann* em Haskell. Calcule o valor de *ackermann 1 2*:**

```
-------------------------------------------------------------------------------------------
ackermann :: Int -> Int -> Int
ackermann 0 n = n + 1
ackermann m 0 = ackermann (m - 1) 1
ackermann m n = ackermann (m - 1) (ackermann m (n - 1))

-------------------------------------------------------------------------------------------
main :: IO ()
main =  do putStr "ackermann 1 2 = "
           print(ackermann 1 2) -- 4
```

**15. Definir uma função que, dado um valor inteiro positivo N, retorna a soma dos quadrados de todos os inteiros compreendidos entre 0 e N, inclusive:**

```
-------------------------------------------------------------------------------------------
-- Versão 1
-------------------------------------------------------------------------------------------
somaQuadrados :: Integer -> Integer
somaQuadrados n
    | n < 0     = error "Valor deve ser positivo"
    | otherwise = n * (n + 1) * (2 * n + 1) `div` 6

-------------------------------------------------------------------------------------------
-- Versão 2
-------------------------------------------------------------------------------------------
somaQuadrados' :: Integer -> Integer
somaQuadrados' n
    | n < 0     = error "Valor deve ser positivo"
    | n == 0    = 0
    | otherwise = n^2 + somaQuadrados' (n-1)

-------------------------------------------------------------------------------------------
main :: IO ()
main = do
    putStrLn "Teste da Versao 1"
    print (somaQuadrados 0 == 0)
    print (somaQuadrados 1 == 1)
    print (somaQuadrados 2 == 5)
    print (somaQuadrados 3 == 14)
    print (somaQuadrados 5 == 55)
    putStrLn ""
    putStrLn "Teste da Versao 2"
    print (somaQuadrados' 0 == 0)
    print (somaQuadrados' 1 == 1)
    print (somaQuadrados' 2 == 5)
    print (somaQuadrados' 3 == 14)
    print (somaQuadrados' 5 == 55)
```

**16. Escreva uma função para verificar se uma string (cadeia de caracteres) é um palíndromo. Segundo Gersting (2017), um palíndromo é uma cadeia de caracteres que é lida da mesma forma normalmente ou de trás para a frente.**

```
-------------------------------------------------------------------------------------------
-- Versão 1
-------------------------------------------------------------------------------------------
ehPalindromo :: String -> Bool
ehPalindromo s = s == reverse s

-------------------------------------------------------------------------------------------
-- Versão 2
-------------------------------------------------------------------------------------------
ehPalindromo' :: String -> Bool
ehPalindromo' s
    | length s <= 1           = True
    | head s /= last s        = False
    | otherwise               = ehPalindromo' (init (tail s))

-------------------------------------------------------------------------------------------
main :: IO ()
main = do
        putStrLn "Testes da Versao 1"
        print (ehPalindromo "arara"     == True)
        print (ehPalindromo "reviver"   == True)
        print (ehPalindromo "haskell"   == False)
        print (ehPalindromo "abba"      == True)
```

```
    print (ehPalindromo "abcba"     == True)
    print (ehPalindromo ""          == True)
    print (ehPalindromo "a"         == True)
    print (ehPalindromo "ab"        == False)
    putStrLn ""
    putStrLn "Testes da Versao 2"
    print (ehPalindromo' "arara"     == True)
    print (ehPalindromo' "reviver"   == True)
    print (ehPalindromo' "haskell"   == False)
    print (ehPalindromo' "abba"      == True)
    print (ehPalindromo' "abcba"     == True)
    print (ehPalindromo' ""          == True)
    print (ehPalindromo' "a"         == True)
    print (ehPalindromo' "ab"        == False)
```

**17. Definir uma função para a sequência de Fibonacci. Para maiores informações sobre a sequência de Fibonacci, consultar Gersting (2017).**

```
-----------------------------------------------------------------------------------------
-- Versão 1
-----------------------------------------------------------------------------------------
fibonacci :: Integer -> Integer
fibonacci numero
    | numero < 0 = error "Valor invalido"
    | numero == 0 = 0
    | numero == 1 = 1
    | otherwise = fibonacci (numero-1) + fibonacci (numero-2)

-----------------------------------------------------------------------------------------
-- Versão 2
-----------------------------------------------------------------------------------------
fibonacci' :: Integer -> Integer
fibonacci' n
    | n < 0     = error "Valor invalido"
    | otherwise = fibAux n 0 1
  where
    fibAux 0 a _ = a
    fibAux n a b = fibAux (n-1) b (a+b)

-----------------------------------------------------------------------------------------
main :: IO ()
main = do
    putStrLn "Versao 1"
    print (fibonacci 0)
    print (fibonacci 1)
    print (fibonacci 2)
    print (fibonacci 3)
    print (fibonacci 4)
    print (fibonacci 5)
    print (fibonacci 6)
    putStrLn "Versao 2"
    print (fibonacci' 0)
    print (fibonacci' 1)
    print (fibonacci' 2)
    print (fibonacci' 3)
    print (fibonacci' 4)
    print (fibonacci' 5)
    print (fibonacci' 6)
```

18. Seja a função *sucessor :: Int -> Int* que calcula qual o sucessor de um número natural, defina a função *soma :: Int -> Int -> Int* para somar dois números naturais a partir da função *successor*:

```
-------------------------------------------------------------------------------------------
sucessor :: Int -> Int
sucessor n = n + 1

-------------------------------------------------------------------------------------------
soma :: Int -> Int -> Int
soma x 0 = x
soma x y = sucessor (soma x (y - 1))

-------------------------------------------------------------------------------------------
testar :: String -> Int -> Int -> IO()
testar nome esperado obtido =
  if esperado == obtido
     then putStrLn $ "[OK] " ++ nome
     else putStrLn $ "[ERRO] " ++ nome

-------------------------------------------------------------------------------------------
main :: IO ()
main = do
  putStrLn "Testes da função sucessor:"
  testar "sucessor 0 = 1" 1 (sucessor 0)
  testar "sucessor 5 = 6" 6 (sucessor 5)
  putStrLn "\nTestes da função soma:"
  testar "soma 2 0 = 2" 2 (soma 2 0)
  testar "soma 0 3 = 3" 3 (soma 0 3)
  testar "soma 2 3 = 5" 5 (soma 2 3)
  testar "soma 4 5 = 9" 9 (soma 4 5)
  testar "soma 0 0 = 0" 0 (soma 0 0)
  testar "soma 5 0 = 5" 5 (soma 5 0)
```

**19. Defina uma função para obter os dígitos de uma string:**

```
-------------------------------------------------------------------------------------------
-- Versão 1
-------------------------------------------------------------------------------------------

import Data.Char (isDigit)

extrairDigitos :: String -> String
extrairDigitos [] = []
extrairDigitos (x:xs)
 | isDigit x = x : extrairDigitos xs
 | otherwise = extrairDigitos xs

-------------------------------------------------------------------------------------------
-- Versão 2
-------------------------------------------------------------------------------------------

digitos :: String -> String
digitos = filter isDigit

-------------------------------------------------------------------------------------------
main :: IO ()
main = do
    putStrLn "Versao 1"
    print (extrairDigitos "abc123def45")
    print (extrairDigitos "Haskell2025")
    print (extrairDigitos "haskell !!!!!")
    putStrLn "Versao 2"
    print (digitos "abc123def45")
    print (digitos "Haskell2025")
    print (digitos "haskell !!!!!")
```

**20. Elaborar uma função para somar os elementos de uma lista de inteiros:**

```
-----------------------------------------------------------------------------------------
somarLista :: [Int] -> Int
somarLista [] = 0
somarLista (x:xs) = x + somarLista xs

-----------------------------------------------------------------------------------------
runTest :: String -> Int -> Int -> IO ()
runTest nomeDoTeste esperado resultadoObtido =
 if esperado == resultadoObtido
 then putStrLn $ "OK: " ++ nomeDoTeste
 else putStrLn $ "FALHOU: " ++ nomeDoTeste

-----------------------------------------------------------------------------------------
testes :: IO ()
testes = do
 runTest "Testa lista vazia" 0 (somarLista [])
 runTest "Testa lista com positivos" 15 (somarLista [1, 2, 3, 4, 5])
 runTest "Testa lista com negativos" (-9) (somarLista [-2, -3, -4])
 runTest "Testa lista com mistos" 5 (somarLista [-5, 10])
 runTest "Testa lista com um elemento" 7 (somarLista [7])

-----------------------------------------------------------------------------------------
main :: IO ()
main = do putStrLn "Inicio dos testes"
          testes
          putStrLn "Fim dos testes"
```

**21. Elaborar uma função para ordenar uma lista de inteiros utilizando o algoritmo de ordenação por inserção (*Insertion Sort*). Maiores informações sobre algoritmos de ordenação, consulte Cormen *et al.* (2002).**

```
-----------------------------------------------------------------------------------------
insertionSort :: [Int] -> [Int]
insertionSort [] = []
insertionSort (numero:lista_restante) = insertion numero (insertionSort lista_restante)

-----------------------------------------------------------------------------------------
insertion :: Int -> [Int] -> [Int]
insertion numero [] = [numero]
insertion numero (elemento:lista_restante)
 | numero <= elemento = numero : (elemento:lista_restante)
 | otherwise = elemento : insertion numero lista_restante

-----------------------------------------------------------------------------------------
runTest :: String -> [Int] -> [Int] -> IO ()
runTest nomeDoTeste esperado resultadoObtido =
 if esperado == resultadoObtido
 then putStrLn $ "OK: " ++ nomeDoTeste
 else putStrLn $ "FALHOU: " ++ nomeDoTeste

-----------------------------------------------------------------------------------------
testesInsertionSort :: IO ()
testesInsertionSort = do
 runTest "Testa lista vazia" [] (insertionSort [])
 runTest "Testa lista com um elemento" [5] (insertionSort [5])
 runTest "Testa lista já ordenada" [1, 2, 3, 4, 5] (insertionSort [1, 2, 3, 4, 5])
 runTest "Testa lista em ordem reversa" [1, 2, 3, 4, 5] (insertionSort [5, 4, 3, 2, 1])
 runTest "Testa lista com elementos repetidos" [2, 2, 5] (insertionSort [2, 5, 2])
 runTest "Testa lista com números negativos" [-3, -1, 0, 2] (insertionSort [0, -1, 2, -3])

-----------------------------------------------------------------------------------------
main :: IO ()
main = testesInsertionSort
```

**22. Defina a função *membroNum :: [Int] -> Int -> Int* em que *membroNum lista elemento* retorna o número de vezes que o *elemento* aparece na lista *lista*:**

```
-----------------------------------------------------------------------------------------
membroNum :: [Int] -> Int -> Int
membroNum [] _ = 0
membroNum (elemento:lista_restante) numero
    | elemento == numero = 1 + membroNum lista_restante numero
    | otherwise = membroNum lista_restante numero

-----------------------------------------------------------------------------------------
main :: IO ()
main = do
    putStrLn "Digite uma lista de números separados por espaço:"
    entradaLista <- getLine
    let lista = map read (words entradaLista) :: [Int]

    putStrLn "Digite o número que deseja contar:"
    entradaNumero <- getLine
    let numero = read entradaNumero :: Int

    let resultado = membroNum lista numero
    putStrLn ("quantidade = " ++ show resultado)
```

**23. Defina a função** operadorE :: [Bool] -> Bool **que retorna a conjunção de uma lista de booleanos:**

```
-----------------------------------------------------------------------------------------
-- Versão 1
-----------------------------------------------------------------------------------------

operadorE :: [Bool] -> Bool
operadorE [] = True
operadorE (elemento:lista_restante) = elemento && operadorE lista_restante

-----------------------------------------------------------------------------------------
-- Versão 2
-----------------------------------------------------------------------------------------
operadorE' :: [Bool] -> Bool
operadorE' = foldr (&&) True

-----------------------------------------------------------------------------------------
main :: IO ()
main = do
    putStrLn "Testes da Versao 1"
    print (operadorE [True, True, True] == True)
    print (operadorE [True, False, True] == False)
    print (operadorE [False, False] == False)
    print (operadorE [True] == True)
    print (operadorE [False] == False)
    putStrLn "Testes da Versao 2"
    print (operadorE' [True, True, True] == True)
    print (operadorE' [True, False, True] == False)
    print (operadorE' [False, False] == False)
    print (operadorE' [True] == True)
    print (operadorE' [False] == False)
```

**24. Defina a função** paraMinuscula :: String -> String **para converter todos os caracteres de uma String para minúscula:**

```
import Data.Char

-----------------------------------------------------------------------------------------
paraMinuscula :: String -> String
paraMinuscula [] = []
paraMinuscula (letra:palavra) = minuscula letra : paraMinuscula palavra
    where
        minuscula :: Char -> Char
        minuscula letra
            | letra >= 'A' && letra <= 'Z' = chr (ord (letra) + 32)
            | otherwise = letra

-----------------------------------------------------------------------------------------
testeParaMinuscula1 :: Bool
testeParaMinuscula1 = paraMinuscula "Ola Mundo" == "ola mundo"


-----------------------------------------------------------------------------------------
testeParaMinuscula2 :: Bool
testeParaMinuscula2 = paraMinuscula "STRING" == "string"

-----------------------------------------------------------------------------------------
testeParaMinuscula3 :: Bool
testeParaMinuscula3 = paraMinuscula "ja eh minuscula" == "ja eh minuscula"

-----------------------------------------------------------------------------------------
testeParaMinuscula4 :: Bool
testeParaMinuscula4 = paraMinuscula "" == ""

-----------------------------------------------------------------------------------------
main :: IO ()
main = do
    putStrLn $ "Teste 1: " ++ show testeParaMinuscula1
    putStrLn $ "Teste 2: " ++ show testeParaMinuscula2
    putStrLn $ "Teste 3: " ++ show testeParaMinuscula3
    putStrLn $ "Teste 4: " ++ show testeParaMinuscula4
```

**25. Defina a função** concatenar :: [[Int]] -> [Int] **para concatenar lista de listas de inteiros:**

```
-----------------------------------------------------------------------------------------
-- Versão 1
-----------------------------------------------------------------------------------------
concatenar :: [[Int]] -> [Int]
concatenar [] = []
concatenar (primeiraLista:restoDasListas) = primeiraLista ++ concatenar restoDasListas

-----------------------------------------------------------------------------------------
-- Versão 2
-----------------------------------------------------------------------------------------
concatenar' :: [[Int]] -> [Int]
concatenar' = foldr (++) []

-----------------------------------------------------------------------------------------
main :: IO ()
main = do
    putStrLn "Versao 1"
    let lista = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
    let listaConcatenada = concatenar lista
    putStrLn $ "Lista original: " ++ show lista
    putStrLn $ "Lista concatenada: " ++ show listaConcatenada
    putStrLn ""
```

```
    putStrLn "Versao 2"
    let lista2 = [[1, 2, 3], [4, 5], [6, 7, 8, 9]]
    let listaConcatenada2 = concatenar' lista2
    putStrLn $ "Lista original: " ++ show lista2
    putStrLn $ "Lista concatenada: " ++ show listaConcatenada2
```

**26. Defina a função** *ordenada :: [Int] -> Bool* **que verifica se uma lista de inteiros está ordenada:**

```
-------------------------------------------------------------------------------------------
ordenada :: [Int] -> Bool
ordenada [] = True
ordenada [_] = True
ordenada (valor1 : valor2 : lista_restante)
 | valor1 <= valor2 = ordenada (valor2 : lista_restante)
 | otherwise = False

-------------------------------------------------------------------------------------------
testeOrdenada1 :: Bool
testeOrdenada1 = ordenada [] == True

-------------------------------------------------------------------------------------------
testeOrdenada2 :: Bool
testeOrdenada2 = ordenada [5] == True

-------------------------------------------------------------------------------------------
testeOrdenada3 :: Bool
testeOrdenada3 = ordenada [1, 2, 3, 4, 5] == True

-------------------------------------------------------------------------------------------
testeOrdenada4 :: Bool
testeOrdenada4 = ordenada [5, 4, 3, 2, 1] == False


-------------------------------------------------------------------------------------------
testeOrdenada5 :: Bool
testeOrdenada5 = ordenada [1, 1, 2, 2, 3] == True

-------------------------------------------------------------------------------------------
testeOrdenada6 :: Bool
testeOrdenada6 = ordenada [1, 3, 2] == False

-------------------------------------------------------------------------------------------
testeOrdenada7 :: Bool
testeOrdenada7 = ordenada [-5, -2, 0, 3] == True

-------------------------------------------------------------------------------------------
main :: IO ()
main = do
  putStrLn $ "Teste 1 (lista vazia): " ++ show testeOrdenada1
  putStrLn $ "Teste 2 (lista com um elemento): " ++ show testeOrdenada2
  putStrLn $ "Teste 3 (lista ordenada crescente): " ++ show testeOrdenada3
  putStrLn $ "Teste 4 (lista ordenada decrescente): " ++ show testeOrdenada4
  putStrLn $ "Teste 5 (lista com elementos repetidos): " ++ show testeOrdenada5
  putStrLn $ "Teste 6 (lista nao ordenada): " ++ show testeOrdenada6
  putStrLn $ "Teste 7 (lista com numeros negativos): " ++ show testeOrdenada7
```

**27. Implemente a função** duplicarLista :: [Int] -> [Int] **que recebe uma lista e retorna uma nova lista em que cada elemento aparece duas vezes seguidas:**

```
-------------------------------------------------------------------------------------------
-- Versão 1
-------------------------------------------------------------------------------------------
duplicarLista :: [Int] -> [Int]
duplicarLista [] = []
duplicarLista (elemento : lista_restante) = elemento : elemento :
    duplicarLista lista_restante

-------------------------------------------------------------------------------------------
-- Versão 2
-------------------------------------------------------------------------------------------
duplicarLista' :: [Int] -> [Int]
duplicarLista' lista = concat (map (\x -> [x, x]) lista)

-------------------------------------------------------------------------------------------
-- Versão 3
-------------------------------------------------------------------------------------------
duplicarLista'' :: [Int] -> [Int]
duplicarLista'' lista
    | null lista = []
    | otherwise = elemento : elemento  : duplicarLista'' (tail lista)
    where
    elemento = head lista

-------------------------------------------------------------------------------------------
main :: IO ()
main = do putStrLn (show (duplicarLista [1,2,3]))
          putStrLn (show (duplicarLista []))
          putStrLn (show (duplicarLista [5]))
          putStrLn ""
          putStrLn (show (duplicarLista' [1,2,3]))
          putStrLn (show (duplicarLista' []))
          putStrLn (show (duplicarLista' [5]))
          putStrLn ""
          putStrLn (show (duplicarLista'' [1,2,3]))
          putStrLn (show (duplicarLista'' []))
          putStrLn (show (duplicarLista'' [5]))
```

**28. Defina a função recursiva removerNegativos :: [Int] -> [Int] que recebe uma lista de inteiros e retorna apenas os números não negativos:**

```
-------------------------------------------------------------------------------------------
removerNegativos :: [Int] -> [Int]
removerNegativos [] = []
removerNegativos (numero : lista_restante)
 | numero < 0 = removerNegativos lista_restante
 | otherwise = numero : removerNegativos lista_restante

-------------------------------------------------------------------------------------------
compararListas :: [Int] -> [Int] -> Bool
compararListas lst1 lst2 = lst1 == lst2

-------------------------------------------------------------------------------------------
imprimirResultadoTeste :: String -> Bool -> IO ()
imprimirResultadoTeste nomeTeste resultado =
 if resultado
 then putStrLn $ "OK: " ++ nomeTeste
 else putStrLn $ "FALHOU: " ++ nomeTeste

-------------------------------------------------------------------------------------------
```

```
testeListaVazia :: IO ()
testeListaVazia = do
 let entrada = []
 let esperado = []
 let resultado = removerNegativos entrada
 imprimirResultadoTeste "Lista Vazia" (compararListas resultado esperado)

-----------------------------------------------------------------------------------------
testeApenasPositivos :: IO ()
testeApenasPositivos = do
 let entrada = [1, 2, 3, 4, 5]
 let esperado = [1, 2, 3, 4, 5]
 let resultado = removerNegativos entrada
 imprimirResultadoTeste "Apenas Positivos" (compararListas resultado esperado)

-----------------------------------------------------------------------------------------
testeApenasNegativos :: IO ()
testeApenasNegativos = do
 let entrada = [-1, -2, -3, -4, -5]
 let esperado = []
 let resultado = removerNegativos entrada
 imprimirResultadoTeste "Apenas Negativos" (compararListas resultado esperado)

-----------------------------------------------------------------------------------------
testeMisto :: IO ()
testeMisto = do
 let entrada = [1, -2, 3, -4, 5]
 let esperado = [1, 3, 5]
 let resultado = removerNegativos entrada
 imprimirResultadoTeste "Lista Mista" (compararListas resultado esperado)

-----------------------------------------------------------------------------------------
testeComZero :: IO ()
testeComZero = do
    let entrada = [-1, 0, 5, -3]
    let esperado = [0, 5]
    let resultado = removerNegativos entrada
    imprimirResultadoTeste "Com Zero" (compararListas resultado esperado)

-----------------------------------------------------------------------------------------
testeNumerosRepetidos :: IO ()
testeNumerosRepetidos = do
    let entrada = [1, -2, 1, -2, 3]
    let esperado = [1, 1, 3]
    let resultado = removerNegativos entrada
    imprimirResultadoTeste "Números Repetidos" (compararListas resultado esperado)

-----------------------------------------------------------------------------------------
testeUmNegativo :: IO ()
testeUmNegativo = do
    let entrada = [-7]
    let esperado = []
    let resultado = removerNegativos entrada
    imprimirResultadoTeste "Um Número Negativo" (compararListas resultado esperado)

-----------------------------------------------------------------------------------------
testeUmPositivo :: IO ()
testeUmPositivo = do
    let entrada = [7]
    let esperado = [7]
    let resultado = removerNegativos entrada
    imprimirResultadoTeste "Um Número Positivo" (compararListas resultado esperado)

-----------------------------------------------------------------------------------------
main :: IO ()
main = do
    putStrLn "Inicio dos testes"
```

```
    testeListaVazia
    testeApenasPositivos
    testeApenasNegativos
    testeMisto
    testeComZero
    testeNumerosRepetidos
    testeUmNegativo
    testeUmPositivo
    putStrLn "Fim dos testes"
```

**29. Defina a função recursiva** *ultimoElemento* **que retorna o último elemento de uma lista de inteiros:**

```
-----------------------------------------------------------------------------------------
main :: IO ()
main = do
      putStrLn "Digite uma lista de números separados por espaço:"
      entradaLista <- getLine
      let lista = map read (words entradaLista) :: [Int]
      case (ultimoElemento lista) of
        Nothing -> putStrLn "Lista Vazia!"
        Just e  -> putStrLn ("Ultimo elemento ="++ show(e))

-----------------------------------------------------------------------------------------
ultimoElemento :: [Int] -> Maybe Int
ultimoElemento  [] = Nothing
ultimoElemento [elemento] = Just elemento
ultimoElemento (elemento : lista_restante) = ultimoElemento lista_restante
```

**30. Defina** quadrados :: [Int] -> [Int] **que retorna uma lista com os quadrados de cada número:**

```
-----------------------------------------------------------------------------------------
quadrados :: [Int] -> [Int]
quadrados lista = [elemento * elemento | elemento <- lista]

-----------------------------------------------------------------------------------------
main :: IO ()
main = do
    putStrLn "Digite uma lista de números inteiros separados por espaço:"
    entrada <- getLine
    let lista = map read (words entrada) :: [Int]
    let resultado = quadrados lista
    putStrLn "Os quadrados dos números são:"
    print resultado
```

**31. Defina** filtrarLongas :: [String] -> [String] **que retorna apenas as strings com mais de 5 caracteres:**

```
-----------------------------------------------------------------------------------------
main :: IO()
main = do  let r1 = filtrarLongas ["haskell","python","c++","c","java"]
           putStrLn (show r1)


-----------------------------------------------------------------------------------------
filtrarLongas :: [String] -> [String]
filtrarLongas lista = [palavra | palavra <- lista, length palavra > 5]
```

**32. Defina** substituirNegativos :: [Int] -> [Int] **que substitui todos os números negativos por zero:**

```
-----------------------------------------------------------------------------------------
substituirNegativos :: [Int] -> [Int]
substituirNegativos lista = [if valor < 0 then 0 else valor | valor <- lista]

-----------------------------------------------------------------------------------------
main :: IO()
main = do  let r1 = substituirNegativos [-10..10]
           putStrLn (show r1)
```

**33. Defina a função** filtrarPares :: [Int] -> [Int] **que retorna apenas os números pares de uma lista:**

```
-----------------------------------------------------------------------------------------
filtrarPares :: [Int] -> [Int]
filtrarPares lista = [valor | valor <- lista, even valor]

-----------------------------------------------------------------------------------------
main :: IO()
main = do  let r1 = filtrarPares [-10..10]
           putStrLn (show r1)
```

**34. Defina uma função para receber uma String e filtrar os dígitos:**

```
-----------------------------------------------------------------------------------------
filtrarDigitos :: String -> String
filtrarDigitos palavra = [caractere | caractere <- palavra, ehDigito caractere]
    where
        ehDigito digito
            | digito >= '0' && digito <= '9' = True
            | otherwise = False

-----------------------------------------------------------------------------------------
main :: IO()
main = do  let r1 = filtrarDigitos "Haskell 1998"
           putStrLn (show r1)
```

**35. Defina a função** quadradosImpares **que recebe uma lista de números inteiros e retorna a lista contendo os quadrados apenas dos números ímpares:**

```
-----------------------------------------------------------------------------------------
quadradosImpares :: [Int] -> [Int]
quadradosImpares [] = []
quadradosImpares lista = [numero*numero | numero <- lista, ehImpar numero]
    where
    ehImpar :: Int -> Bool
    ehImpar numero
        | numero `mod` 2 == 0 = False
        | otherwise = True


-----------------------------------------------------------------------------------------
main :: IO()
main = do  let r1 = quadradosImpares [1..10]
           putStrLn (show r1)
```

**36. Em uma árvore binária, uma folha é um nó que não possui filhos. Elabore uma função para calcular o número de folhas de uma árvore binária. Maiores informações sobre árvores binárias e árvores binárias de busca, consultar Szwarcfiter e Markenzon (2010).**

```
-----------------------------------------------------------------------------------------
data Arv a = Vazia | No a (Arv a) (Arv a)

-----------------------------------------------------------------------------------------
numFolhas :: Arv a -> Int
numFolhas Vazia = 0
numFolhas (No _ Vazia Vazia) = 1
numFolhas (No _ esq dir) = numFolhas esq + numFolhas dir

-----------------------------------------------------------------------------------------
ar1 :: Arv Int
ar1 = Vazia

-----------------------------------------------------------------------------------------
ar2 :: Arv Int
ar2 = No 5 (No 3 Vazia Vazia) (No 7 Vazia Vazia)

-----------------------------------------------------------------------------------------
ar3 :: Arv String
ar3 = No "haskell" Vazia Vazia

-----------------------------------------------------------------------------------------
main :: IO ()
main = do print (numFolhas ar1)
          print (numFolhas ar2)
          print (numFolhas ar3)
```

**37. Elaborar uma função para calcular a distância entre dois pontos:**

```
-----------------------------------------------------------------------------------------
type Ponto = (Float, Float)
distanciaPontos :: Ponto -> Ponto -> Float
distanciaPontos (x1,y1) (x2,y2) = sqrt(((x2-x1)**2)+((y2-y1)**2))

-----------------------------------------------------------------------------------------
main :: IO ()
main = print (distanciaPontos (0,0) (3,4))
```

**38. Elaborar uma função para testar se um ponto está ou não dentro de um círculo:**

```
-----------------------------------------------------------------------------------------
type Ponto = (Float, Float)
type Circulo = (Centro, Raio)
type Centro = Ponto
type Raio = Float

-----------------------------------------------------------------------------------------
distanciaPontos :: Ponto -> Ponto -> Float
distanciaPontos (x1,y1) (x2,y2) = sqrt(((x2-x1)**2)+((y2-y1)**2))

-----------------------------------------------------------------------------------------
estaDentro :: Ponto -> Circulo -> Bool
estaDentro ponto (centro,raio)
  | distanciaPontos ponto centro <= raio = True
  | otherwise = False

-----------------------------------------------------------------------------------------
main :: IO ()
```

```
main = do
  let c = ((0, 0), 5)
  print $ estaDentro (2, 2) c
  print $ estaDentro (0, 0) c
  print $ estaDentro (6, 0) c
```

**39. Crie um banco de dados para um mercado contendo uma lista de produtos, em que cada produto possui código de barra, nome e preço. Após isso, crie uma função que, ao passar o código de barra, retorna o nome do produto e o preço:**

```
-------------------------------------------------------------------------------------------
type CodigoBarra = Int
type NomeProduto = String
type Preco = Float
type BancoDeDados = [(CodigoBarra, NomeProduto, Preco)]
 -------------------------------------------------------------------------------------------
bd :: BancoDeDados
bd = [(1001, "Refrigerante", 450),
 (1002, "Leite", 320),
 (1003, "Biscoito", 210),
 (1004, "Suco", 989),
 (1005, "Arroz", 345),
 (1006, "Feijao", 780)
 ]

-------------------------------------------------------------------------------------------
buscarBDaux :: CodigoBarra -> BancoDeDados -> Maybe (NomeProduto, Preco)
buscarBDaux codigo [] = Nothing
buscarBDaux codigo ((c,n,p) : restante)
 |codigo == c = Just (n,p)
 |otherwise = buscarBDaux codigo restante

-------------------------------------------------------------------------------------------
buscarBD :: CodigoBarra -> Maybe (NomeProduto, Preco)
buscarBD codigo = buscarBDaux codigo bd

-------------------------------------------------------------------------------------------
main :: IO ()
main = do putStrLn "Digite o código:"
          codigo<-readLn
          case buscarBD codigo of
            Nothing -> do putStrLn "produto nao existente! Tente novamente"
                          main
            Just (n,p) -> do putStrLn ("Nome = "++n)
                             putStrLn ("Preco = "++ show(p))
```

**40. Escreva uma função** troca :: (Int,Int) -> (Int,Int) **que troca o primeiro com o segundo elemento de um par de inteiros:**

```
-------------------------------------------------------------------------------------------
troca :: (Int, Int) -> (Int, Int)
troca (a, b) = (b,a)

-------------------------------------------------------------------------------------------
main :: IO()
main = putStrLn (show(troca(2,3)))
```

**41. Defina um tipo** número complexo **utilizando tuplas e escreva as operações de adição e subtração para números complexos:**

```
-----------------------------------------------------------------------------------------
type Complexo = (Double, Double)

-----------------------------------------------------------------------------------------
soma :: Complexo -> Complexo -> Complexo
soma (real1, imaginario1) (real2, imaginario2) = (real1 +
    real2, imaginario1 + imaginario2)

-----------------------------------------------------------------------------------------
subtrai :: Complexo -> Complexo -> Complexo
subtrai (real1, imaginario1) (real2, imaginario2) = (real1 - real2,
    imaginario1 – imaginario2)

-----------------------------------------------------------------------------------------
main :: IO()
main = do putStrLn $ show (soma (2,3) (1,1))
          putStrLn $ show (subtrai (2,3) (1,1))
```

**42. Defina uma função** maxocorre **que calcula o maior número de uma lista juntamente com o número de ocorrências do máximo. Por exemplo: maxocorre [3, 1, 3] = (3,2).**

```
-----------------------------------------------------------------------------------------
maxocorre :: [Int] -> Maybe (Int, Int)
maxocorre [] = Nothing
maxocorre lista = Just (maxOcorreAux lista, contaOcorrencias (maxOcorreAux lista) lista)
    where
    maxOcorreAux :: [Int] -> Int
    maxOcorreAux [elemento] = elemento
    maxOcorreAux (a:b) = max a (maxOcorreAux b)

    contaOcorrencias :: Int -> [Int] -> Int
    contaOcorrencias _ [] = 0
    contaOcorrencias numero (valor : restante)
        | numero == valor = 1 + contaOcorrencias numero restante
        | otherwise = contaOcorrencias numero restante


-----------------------------------------------------------------------------------------
main :: IO()
main = do putStrLn "Digite uma lista de inteiros:"
          lista<-getLine
          case maxocorre (read lista :: [Int]) of
            Nothing -> do putStrLn "Lista nao pode vazia!"
                          main
            Just (a,b) -> do putStrLn("Maior numero ="++show(a))
                             putStrLn("Num. ocorrencias ="++show(b))
```

**43. Defina uma função ultimo :: [Int] -> Int que recebe uma lista de inteiros e retorna o último elemento da lista:**

```
-----------------------------------------------------------------------------------------
ultimo :: [Int] -> Int
ultimo [a] = a
ultimo (a:b) = ultimo b

-----------------------------------------------------------------------------------------
main :: IO()
main = do print(ultimo [1..100])
          print(ultimo[2])
          print(ultimo[1,2,3,4,5,6])
```

**44. Defina uma função minEmax :: [Int] -> (Int,Int) que retorna o menor e maior valor de uma lista de inteiros:**

```
-------------------------------------------------------------------------------------------
minEmax :: [Int] -> (Int, Int)
minEmax lista = (minimo lista, maximo lista)
 where
  minimo :: [Int] -> Int
  minimo [elemento] = elemento
  minimo (a:b) = min a (minimo b)

  maximo :: [Int] -> Int
  maximo [elemento] = elemento
  maximo (a:b) = max a (maximo b)

-------------------------------------------------------------------------------------------
main :: IO()
main = do print(show (minEmax  [1..100]))
          print(show (minEmax [2]))
          print(show (minEmax [1,2,3,4,5,6]))
```

**45. Baseado nas definições abaixo, crie funções para:**

```
type Matricula = Int
type Nome = String
type Bairro = String
type Aluno = (Matricula,Nome,Bairro)
type Alunos = [Aluno]
```

**(a) Incluir um aluno na lista de alunos;**
**(b) Excluir um aluno da lista de alunos;**
**(c) Pesquisar o nome de todos os alunos que moram em um determinado bairro;**
**(d) Pesquisar os dados do aluno pelo seu número de matrícula.**

```
-------------------------------------------------------------------------------------------
type Matricula = Int
type Nome = String
type Bairro = String
type Aluno = (Matricula,Nome,Bairro)
type Alunos = [Aluno]

-------------------------------------------------------------------------------------------
incluirAluno :: Aluno -> Alunos -> Alunos
incluirAluno aluno lista = aluno : lista

-------------------------------------------------------------------------------------------
excluirAluno :: Matricula -> Alunos -> Alunos
excluirAluno _ [] = []
excluirAluno matricula (a@(m, _, _) : restante_alunos)
    | matricula == m = restante_alunos
    | otherwise = a : excluirAluno matricula restante_alunos

-------------------------------------------------------------------------------------------
pesquisarPorBairro :: Bairro -> Alunos -> [Nome]
pesquisarPorBairro _ [] = []
pesquisarPorBairro bairro ((_, nome, b) : restante_alunos)
    | bairro == b = nome : pesquisarPorBairro bairro restante_alunos
    | otherwise = pesquisarPorBairro bairro restante_alunos

-------------------------------------------------------------------------------------------
pesquisarPorMatricula :: Matricula -> Alunos -> Maybe Aluno
pesquisarPorMatricula _ [] = Nothing
pesquisarPorMatricula matricula (a@(m, _, _) : restante_alunos)
    | matricula == m = Just a
    | otherwise = pesquisarPorMatricula matricula restante_alunos
```

**46. Implementa a função polimorfica *contat'* utilizando compreensão de listas:**

**Exemplo:**

```
> concat' [[1,2,3],[4,5],[6,7]]
[1,2,3,4,5,6,7]

Resposta:
-------------------------------------------------------------------------------------------
concat :: [[a]] -> [a]
concat xss = [x | xs<-xss, x<-xs]
```

**47. Crie uma função duasVezes que recebe uma função f e um valor x, e aplica f duas vezes em x:**

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

-------------------------------------------------------------------------------------------
main :: IO()
main =  print (duasVezes (*2) 4)
```

**48. Crie uma função produtoLista que calcula o produto de todos os elementos de uma lista de inteiros:**

```
-------------------------------------------------------------------------------------------
produtoLista :: [Int] -> Int
produtoLista = foldl (*) 1

-------------------------------------------------------------------------------------------
main :: IO()
main = do print (produtoLista  [])
          print (produtoLista [2])
          print (produtoLista [1,2,3,4,5,6])
```

**49. Crie uma função quadrados que recebe uma lista de inteiros e retorna uma lista com os quadrados dos números:**

```
-------------------------------------------------------------------------------------------
-- Versão 1
-------------------------------------------------------------------------------------------
quadrados :: [Int] -> [Int]
quadrados = map (^2)

-------------------------------------------------------------------------------------------
-- Versão 2
-------------------------------------------------------------------------------------------
quadrados' :: [Int] -> [Int]
quadrados' = map (\x -> x^2)


-------------------------------------------------------------------------------------------
main :: IO()
main = do print (quadrados  [])
          print (quadrados [2])
          print (quadrados [1,2,3,4,5,6])
          putStrLn ""
          print (quadrados'  [])
          print (quadrados' [2])
          print (quadrados' [1,2,3,4,5,6])
```

**50. Qual é o resultado da avaliação da expressão abaixo?**

```
-----------------------------------------------------------------------------------------
quadrados = map ((/2).(+1))

-----------------------------------------------------------------------------------------
main :: IO()
main =  print (quadrados [1,2,3,4,5,6]) -- [1.0,1.5,2.0,2.5,3.0,3.5]
```

**51. Crie a função aplicarListaFuncoes, que recebe uma lista de funções [$f_1$, $f_2$, ..., $f_n$] e uma lista de valores [$x_1$, $x_2$, ..., $x_n$], aplicando $f_i$ a $x_i$ correspondente. Se as listas tiverem tamanhos diferentes, pare na menor.**

```
-----------------------------------------------------------------------------------------
aplicarListaFuncoes :: [Float -> Float] -> [Float] -> [Float]
aplicarListaFuncoes [] _ = []
aplicarListaFuncoes _ [] = []
aplicarListaFuncoes (f:fs) (v:vs) = f v : aplicarListaFuncoes fs vs

-----------------------------------------------------------------------------------------
main :: IO()
main = do let lista = aplicarListaFuncoes [sqrt,sin,cos] [1,2,3]
          putStrLn (show lista)
```

**52. Elaborar um programa Haskell para ler um arquivo chamado** *entrada.txt***, converter para** *maiusculo* **o conteúdo do arquivo e gravar o resultado no arquivo chamado** *saida.txt***. Utilize função de alta ordem para realizar a conversão.**

```
import Data.Char (toUpper)

-----------------------------------------------------------------------------------------
converterArquivoParaMaiusculo :: FilePath -> FilePath -> IO ()
converterArquivoParaMaiusculo entrada saida =
  do conteudo <- readFile entrada
     let conteudoMaiusculo = map toUpper conteudo
     writeFile saida conteudoMaiusculo

-----------------------------------------------------------------------------------------
main :: IO ()
main = converterArquivoParaMaiusculo "entrada.txt" "saida.txt"
```

**53. Qual é o resultado da avaliação da expressão abaixo?**

```
-----------------------------------------------------------------------------------------
main :: IO()
main = print (fs 2 3)

-----------------------------------------------------------------------------------------
fs :: Int -> Int -> Int
fs x y =
  let
    soma = a+b
  in
    div soma 5 -- 2
  where
    a :: Int
    a = x^2
    b :: Int
    b = y^2
```

**54. Crie uma função polimórfica que retorne o primeiro elemento de uma tupla:**

```
-----------------------------------------------------------------------------------------
primeiro :: (a, b) -> a
primeiro (a, _) = a

-----------------------------------------------------------------------------------------
main :: IO()
main = do print (primeiro  (2,3))
          print (primeiro  (True,False))
          print (primeiro  (2.6,3.8))
```

**55. Crie uma função polimórfica que retorne o maior elemento de uma lista:**

```
-----------------------------------------------------------------------------------------
maiorElemento :: (Ord a) => [a] -> a
maiorElemento [] = error "Lista vazia!!!"
maiorElemento [elemento] = elemento
maiorElemento (elemento:restante) = max elemento (maiorElemento restante)

-----------------------------------------------------------------------------------------
main :: IO ()
main =  do
  print (maiorElemento [1..100])
  print (maiorElemento "abcd")
  print (maiorElemento ['a'..'z'])
  print (maiorElemento [17.6,8.8,9.9])
```

**56. Crie uma função polimórfica que retorne a quantidade de elementos de uma lista:**

```
-----------------------------------------------------------------------------------------
contagemDeElementos :: [a] -> Int
contagemDeElementos [] = 0
contagemDeElementos (elemento:restante) = 1 + contagemDeElementos restante

-----------------------------------------------------------------------------------------
main :: IO ()
main =  do
  print (contagemDeElementos [1..100])
  print (contagemDeElementos "abcd")
  print (contagemDeElementos ['a'..'z'])
  print (contagemDeElementos [7.6,8.8,9.9])
```

**57. Defina uma função polimórfica chamada zipar, que recebe duas listas e devolve uma lista de pares:**

```
-----------------------------------------------------------------------------------------
zipar :: [a] -> [b] -> [(a,b)]
zipar [] [] = []
zipar [] _ = []
zipar _ [] = []
zipar (a:b) (c:d) = (a,c) : zipar b d

-----------------------------------------------------------------------------------------
main :: IO ()
main =  do
 putStrLn "Digite uma lista de inteiros:"
 lista1 <- getLine
 let lst1 = read lista1 :: [Int]
 putStrLn "Digite uma lista de caracteres:"
 lista2 <- getLine
 let lst2 = read lista2 :: [Char]
 putStrLn (show (zipar lst1 lst2))
```

**58. Defina uma função polimórfica chamada concatena, que recebe duas listas de mesmo tipo e retorna uma lista com todos os elementos:**

```
-----------------------------------------------------------------------------------------
concatena :: [a] -> [a] -> [a]
concatena [] lista = lista
concatena (elemento:lista1) lista2 = elemento: concatena lista1 lista2

-----------------------------------------------------------------------------------------
main :: IO ()
main =  do
          putStrLn (show (concatena [1,2] [3,4]))
          putStrLn (show (concatena [] [3,4]))
          putStrLn (show (concatena [1,2] []))
          putStrLn (show (concatena [1] [2..5]))
          putStrLn (show (concatena [1..4] [5]))
```

**59. Qual a saída das expressões abaixo?**

```
a) foldr (\x y -> x + 2 * y) 0 [1, 2, 3]

17

b) (foldr (-) 0 . map (\x -> x * x) . filter (\x -> x > 0)) [-10 .. 10]

-55

c) filter ((> 5) . sum) [[1, 2], [3, 4], [2, 5], [6]]

[[3,4],[2,5],[6]]

d) zipWith (\a b -> (a * 2) + b) [1, 2, 3] [4, 5, 6]

[6,9,12]

e) takeWhile (< 5) [1, 3, 5, 2, 4, 6]

[1,3]

f) Qual é o resultado da chamada sp 13 0 ?

sp :: Int -> Int -> Int
sp n acumulador
    | n < 0     = acumulador
    | even n    = sp (n - 1) (acumulador + n)
    | otherwise = sp (n - 11) acumulador

main :: IO()
main = do putStrLn "Chamada de sp 13 0 ="
          print (sp 13 0)

42

g) sum (dropWhile (>0) [4,3,-4,-2,1,2]))

-3

h) sum (take 5 (reverse [100,98 .. 1]))

30

i) map (+2) ([1..5] ++ [10 .. 1])

[3,4,5,6,7]

j) 0x10 + 0o10
24
```

```
l) signum 3 + abs (-10) + signum (-2) + round(6.7)

17

m) take 5 [4] : []

[[4]]

n) last (map fst (zip [1,2,3,4] [2,4,6]))

3

o) (nub . take 4 . cycle) [1,2]

[1,2]


p) [ (x,y) | x<-[1..5], y<-[10..20], even x, even y]

[(2,10),(2,12),(2,14),(2,16),(2,18),(2,20),(4,10),(4,12),(4,14),(4,16),(4,18),(4,20)]

q)

f :: Integer -> Integer -> Integer
f  = \ x y -> 2*x+y

main :: IO ()
main =  putStrLn $ show [ f a b  | a<-[1..5], b<-[1..3], a>b ]


[5,7,8,9,10,11,11,12,13]

r)
f :: [Int] -> Int
f xs = foldr (+) 0 (map (\x -> x * x) (filter (\x -> x > 0) xs))

main :: IO ()
main =  do
 putStr "f [-10, -8 .. 5] = "
 print(f [-10, -8 ..5])

f [-10, -8 .. 5] = 20

s) take 10 [ (i,j) | i <- [1,2], j <- [1..] ]

[(1,1),(1,2),(1,3),(1,4),(1,5),(1,6),(1,7),(1,8),(1,9),(1,10)]

t) (sum . take 4 . filter even) [10..]

52

u)

main :: IO()
main = putStrLn (show (f [[1,2,3],[1],[1,2]]))
 where
 f = foldl (+) 0 . map (+2) . filter (>1) . map length

9

v) (sum . map ((\x y -> x+y) 2)) [1,2,3]

12
```

**60. Defina uma função extenso que converte um número inteiro (que esteja no intervalo de 0 até 99, inclusive) para sua forma por extenso. Consulte Bird e Wadler (1998) para outra solução deste exercício.**

```
-----------------------------------------------------------------
-- main.hs
-----------------------------------------------------------------
module Main(main) where

import NumeroPorExtenso

main :: IO ()
main = entradaUsuario

entradaUsuario :: IO ()
entradaUsuario = do
 putStrLn "Digite um número entre 0 e 99: "
 numeroStr <- getLine
 let numero = read numeroStr :: Int
 if numero >= 0 && numero <= 99
 then putStrLn ("O número por extenso é: " ++ extenso numero)
 else putStrLn "Erro! Número fora do intervalo de 0 a 99"
 putStrLn "Deseja novamente s/n:"
 resposta<-getLine
 case resposta of
      "s" -> do entradaUsuario
      _   -> do putStrLn "Fim do programa"
                return ()



-----------------------------------------------------------------
-- extenso.hs
-----------------------------------------------------------------
module NumeroPorExtenso(extenso) where

extenso :: Int -> String
extenso 0 = "zero"
extenso 1 = "um"
extenso 2 = "dois"
extenso 3 = "tres"
extenso 4 = "quatro"
extenso 5 = "cinco"
extenso 6 = "seis"
extenso 7 = "sete"
extenso 8 = "oito"
extenso 9 = "nove"
extenso 10 = "dez"
extenso 11 = "onze"
extenso 12 = "doze"
extenso 13 = "treze"
extenso 14 = "quatorze"
extenso 15 = "quinze"
extenso 16 = "dezesseis"
extenso 17 = "dezessete"
extenso 18 = "dezoito"
extenso 19 = "dezenove"
extenso 20 = "vinte"
extenso 30 = "trinta"
extenso 40 = "quarenta"
extenso 50 = "cinquenta"
extenso 60 = "sessenta"
extenso 70 = "setenta"
extenso 80 = "oitenta"
extenso 90 = "noventa"
extenso n
```

```
| n > 20 && n < 100 = extenso ((div n 10) * 10) ++ " e " ++
imprime (mod n 10)
| otherwise = "Erro! Numero fora do intervalo de 0 a 99"
where
imprime :: Int -> String
imprime 0 = ""
imprime n = extenso n
```

## REFERÊNCIAS

ABELSON, Harold; SUSSMAN, Gerald Jay; SUSSMAN, Julie. *Structure and interpretation of computer programs*. 2. ed. Cambridge: MIT Press, 1996.

ACIÓLY, Benedito Melo; ARAÚJO, Stenio Longo. *Introdução ao Haskell*. Vitória da Conquista: Edições UESB, 2008. Disponível em: https://www2.uesb.br/editora/?p=68. Acesso em: 3 jun. 2025.

ALLEN, Christopher; MORONUKI, Julie. *Haskell Programming from First Principles*. Texas (US): Lorepub, 2020. Disponível em: https://haskellbook.com/. Acesso em: 3 jun. 2025.

BIRD, Richard; WADLER, Philip. *Introduction to functional programming*. New York: Prentice Hall, 1988. (Prentice-Hall International Series in Computer Science).

BRASIL. Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira (INEP). *Prova de Computação:* ENADE 2008. Brasília: INEP, 2008. Disponível em: https://download.inep.gov.br/download/Enade2008_RNP/COMPUTACAO.pdf . Acesso em: 4 jun. 2025

BRASIL. Ministério da Educação. Instituto Nacional de Estudos e Pesquisas Educacionais Anísio Teixeira (INEP). *Exame Nacional de Desempenho dos Estudantes (ENADE):* prova de Ciência da Computação – Bacharelado. Brasília, DF: INEP, 2014. Disponível em: https://download.inep.gov.br/educacao_superior/enade/provas/2014/03_computacao_bacharelado.pdf. Acesso em: 5 jun. 2025

CORMEN, Thomas H *et al. Algoritmos:* teoria e prática. São Paulo: Campus, 2002.

DAVIE, Antony J. T. *An introduction to functional programming systems using Haskell*. Cambridge: Cambridge University Press, 1992.

DONS, Bryan *et al.* Real World Haskell. Disponível em: http://book.realworldhaskell.org/. Acesso em: 27 abr. 2025.

GERSTING, Judith L. *Fundamentos matemáticos para a ciência da computação:* matemática discreta e suas aplicações*.* 7. ed. Tradução de Valéria de Magalhães Iorio. Rio de Janeiro: LTC, 2017.

HASKELL. Haskell Programming Language. Disponível em: https://www.haskell.org/. Acesso em: 27 abr. 2025.

HASKELL DOCUMENTATION. The Haskell Programming Language. Disponível em: https://www.haskell.org/documentation/. Acesso em: 27 abr. 2025.

HASKELL LANGUAGE. *Haskell 2010 Language Report*. 2010. Disponível em: https://www.haskell.org/onlinereport/haskell2010/ . Acesso em: 24 maio 2025.

HASKELL STACK. *Haskell Stack documentation*. Disponível em: https://docs.haskellstack.org/en/stable/. Acesso em: 4 jun. 2025.

HASKELL WIKI. Haskell Programming Language. Disponível em: https://wiki.haskell.org. Acesso em: 27 abr. 2025.

HUDAK, Paul. *Learning functional programming (The haskell schooll of expression)*. New York, NY: Cambridge University Press, 2000.

HUTTON, Graham. *Programming in Haskell*. 2nd ed. Cambridge: Cambridge University Press, 2016.

JDoodle. Haskell Online Compiler. Disponível em: https://www.jdoodle.com/execute-haskell-online. Acesso em: 27 abr. 2025.

JONES, Simon Peyton. *The Implementation of Functional Programming Languages*. New York: Prentice Hall International, 1987.

MICROSOFT CORPORATION. *Visual Studio Code*. Redmond, WA: Microsoft, [2025?]. Disponível em: https://code.visualstudio.com/ . Acesso em: 3 jun. 2025.

MIRAN, Lipovaca. *Learn You a Haskell for Great Good!*. San Francisco (CA): No Starch Press, 2011. Disponível em: http://learnyouahaskell.com/ . Acesso em: 24 maio 2025.

PAULSON, Lawrence C. *ML for the working programmer*. 2. ed. Cambridge: Cambridge University Press, 1996.

REPL.IT. Haskell Online Compiler and Editor. Disponível em: https://replit.com/languages/haskell. Acesso em: 27 abr. 2025.

SÁ, Claudio Cesar de; SILVA, Márcio Ferreira da. *Haskell:* uma abordagem prática. São Paulo: Novatec Editora, 2006.

SEBESTA, Robert W. *Conceitos de linguagens de programação.* 5. ed. Porto Alegre: Bookman, 2003.

SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. *Estruturas de dados e seus algoritmos*. 3. ed. Rio de Janeiro: LTC, 2010.

THE GLASGOW HASKELL COMPILER. *GHC: The Glasgow Haskell Compiler*. 2025. Disponível em: https://www.haskell.org/ghc/ . Acesso em: 24 maio 2025.

THOMPSON, Simon. *Haskell:* the craft of functional programming. Harlow: Addison-Wesley, 1996.

THOMPSON, Simon. *Haskell:* the craft of functional programming. 2nd ed. Harlow: Addison-Wesley, 1999.

THOMPSON, Simon. *Haskell:* the craft of functional programming. 3rd ed. Harlow: Addison-Wesley, 2011.

TRY HASKELL. Try Haskell Online. Disponível em: https://tryhaskell.org/. Acesso em: 27 abr. 2025.

WANDBOX. *Wandbox: Online Compiler and Sharing Tool*. [S.l.]: Wandbox, [2025?]. Disponível em: https://wandbox.org/. Acesso em: 3 jun. 2025.

WATT, David A. *Programming language concepts and paradigms.* Saddle River, NJ (US): Prentice Hall, 1990.

Haskell é uma linguagem de programação funcional pura. Este Caderno Didático, desenvolvido a partir da monitoria da disciplina Programação Declarativa do Curso de Bacharelado em Ciência da Computação da UESB, tem como objetivo apresentar diversos exercícios resolvidos em Haskell visando auxiliar os discentes na compreensão da programação funcional.




UESB
Universidade Estadual
do Sudoeste da Bahia

ISBN 978-65-87106-89-2
9 786587 106892

GOVERNO DO ESTADO
BAHIA
SECRETARIA DA EDUCAÇÃO