

# Introdução a Haskell

Haskell é uma linguagem de programação funcional pura, com avaliação preguiçosa (lazy evaluation) e um sistema de tipos forte e estático. Isso significa:

- **Funcional pura:** As funções não têm efeitos colaterais. Elas sempre retornam o mesmo resultado para os mesmos argumentos.
- **Lazy evaluation:** valores só são computados quando realmente são necessários.
- **Tipagem estática:** erros de tipo são detectados em tempo de compilação, garantindo mais segurança.
- **Imutabilidade:** valores não podem ser alterados após criados

## Exemplo:

```
-- Definindo uma função pura
dobrar :: Int -> Int
dobrar x = x * 2

main :: IO ()
main = print (dobrar 5) -- Saída: 10
```

## Tipos básicos:

- **Int** - números inteiros limitados
- **Integer** - números inteiros arbitrariamente grandes
- **Float** - números de ponto flutuante
- **Double** - números de ponto flutuante com mais precisão
- **Bool** - valores lógicos (True / False)
- **Char** - um único caractere
- **Text** (via pacote text) - texto eficiente
- **[a]** - listas (por exemplo: [1,2,3])
- **(a,b)** - tuplas (por exemplo: (1, "Haskell"))

## Ferramentas:

Para trabalhar com Haskell, você pode usar:

1. **GHC (Glasgow Haskell Compiler):** compilador oficial

```
bash

ghc arquivo.hs
./arquivo
```

2. GHCi: interpretador interativo do Haskell (ótimo para testes rápidos).

```
bash

ghci
Prelude> 2 + 3
5
```

Comandos úteis:

- ghci -> abre o interpretador
- :l [arquivo.hs](#) -> carrega arquivo
- :r -> recarrega o arquivo
- :t expressa -> mostra o tipo de uma expressão
- :quit -> sai do GHCi

3. stack: ferramenta para gerenciar projetos, dependências e builds.

```
bash

stack new meu-projeto
cd meu-projeto
stack run
```

4. Cabal: outra ferramenta de build e gerenciamento de pacotes.

```
bash

cabal update
cabal build
```

5. Hoogle: buscador de funções e tipos Haskell(<http://hoogle.haskell.org>).

# Funções

## 1) Introdução às funções.

Em Haskell, funções são cidadãs de primeira classe - você pode passá-las como argumento, retorná-las de outras funções e armazená-las em estruturas de dados.

### Exemplo simples:

```
-- Uma função que soma dois números
soma :: Int -> Int -> Int
soma x y = x + y

main :: IO ()
main = print (soma 3 4) -- Saída: 7
```

Diferente de linguagens imperativas, funções em Haskell não tem efeitos colaterais, a não ser que estejam dentro de um contexto explícito, com **IO**.

## 2) Definição com padrões (Pattern Matching)

Você pode definir funções diferentes dependendo do padrão dos argumentos:

### Exemplo:

```
fatorial :: Int -> Int
fatorial 0 = 1 -- Caso base
fatorial n = n * fatorial (n-1) -- Caso recursivo

main :: IO ()
main = print (fatorial 5) -- Saída: 120
```

### Outro exemplo usando lista:

```
cabeca :: [a] -> a
cabeca [] = error "Lista vazia!" -- padrão para lista vazia
cabeca (x:_) = x -- padrão para lista não-vazia
```

### 3 Definições com guardas

Guardas ( | ) permitem expressar condições mais legíveis do que vários **if-then-else**.

Exemplo:

```
classificarNota :: Int -> Text
classificarNota n
  | n >= 90   = "Excelente"
  | n >= 70   = "Boa"
  | n >= 50   = "Regular"
  | otherwise = "Reprovado"

main :: IO ()
main = print (classificarNota 85) -- Saída: "Boa"
```

**otherwise** é apenas um alias para **True**.

### 4 Definições locais

Você pode definir variáveis ou funções locais com **where** ou **let ... in**.

Exemplo com *where*.

```
hipotenusa :: Double -> Double -> Double
hipotenusa a b = sqrt (quadrado a + quadrado b)
  where
    quadrado x = x * x
```

Exemplo com *let ... in*.

```
main :: IO ()
main = print resultado
  where
    resultado = let x = 3; y = 4 in x * y
```

## 5Currying

Em Haskell, todas as funções recebem um argumento por vez. Uma função de dois parâmetros, na verdade, retorna uma nova função.

**Exemplo:**

```
soma :: Int -> Int -> Int
soma x y = x + y

somaCinco :: Int -> Int
somaCinco = soma 5 -- função parcialmente aplicada

main :: IO ()
main = print (somaCinco 10) -- Saída: 15
```

Isso permite criar funções especializadas a partir de funções mais gerais.

## 6Inferência de tipos

O compilador do GHC pode deduzir os tipos das funções automaticamente, mesmo sem anotação explícita.

```
soma x y = x + y -- GHC infere: soma :: Num a => a -> a -> a
```

Mesmo assim, é recomendado declarar os tipos para deixar o código mais legível.

## 7Notação infixa e prefixa

Em Haskell, a maior parte das funções é refixa (o nome da função vem antes dos argumentos), mas você também pode usá-las de infixa usando crases.

**Exemplo:**

```
-- Prefixa:
soma 2 3      -- Saída: 5

-- Infixa:
2 `soma` 3    -- Também 5
```

Operadores padrão como `+`, `-`, `*` já são naturalmente infixos, você também pode definir seus próprios operadores usando símbolos.

```
(.*) :: Int -> Int -> Int
a .* b = a * b + 1

main :: IO ()
main = print (2 .* 3) -- Saída: 7
```

# Tuplas

## 1) Introdução às tuplas

Uma tupla é uma estrutura de dados que pode conter um número fixo de valores de tipos diferentes.

- São delimitadas por parênteses.
- Os elementos são separados por vírgula.
- Diferente de listas, as tuplas não precisam ser homogêneas (algo que tem a mesma natureza, estrutura ou composição)

### Exemplo:

```
peessoa :: (Text, Int)
peessoa = ("Alice", 30) -- Nome (Text) e idade (Int)
```

As tuplas de dois elementos são chamadas de *pares*, mas podem ter mais de dois valores (**a,b,c, ...**).

## 2) Acesso as tuplas

Você pode acessar o elementos de uma tupla de duas posições usando funções pré-definidas:

```
fst :: (a, b) -> a    -- Retorna o primeiro elemento
snd :: (a, b) -> b    -- Retorna o segundo elemento
```

### Exemplo:

```
main :: IO ()
main = do
    let par = ("Haskell", 1990)
    print (fst par) -- Saída: "Haskell"
    print (snd par) -- Saída: 1990
```

Para tuplas maiores que 2 elementos, geralmente usamos pattern matching.

### 3 Decompondo tuplas em padrões

Você pode extrair os valores diretamente usando pattern matching:

Exemplo:

```
dados :: (Text, Int, Bool)
dados = ("Bob", 25, True)

main :: IO ()
main = do
  let (nome, idade, ativo) = dados
  print nome      -- Saída: "Bob"
  print idade     -- Saída: 25
  print ativo     -- Saída: True
```

Também funciona em definições de função

```
saudacao :: (Text, Int) -> Text
saudacao (nome, idade) = "Olá, " <> nome <> "! Você tem " <> show idade <> " anos."
```

### 4 Tupla vazia


A tupla vazia `()` também é conhecida como `unit`.

Ela é um tipo especial que possui exatamente um valor. `()`.

Exemplo:

```
-- Tipo unit usado quando não há valor significativo de retorno
printUnit :: IO ()
printUnit = print ()
```

`()` é usado em Haskell principalmente para representar “nenhuma informação relevante” (parecido com **void** em outras linguagens).

 Resumo rápido:

Conceito	Exemplo	Observação
Tupla simples	<code>("Haskell", 1990)</code>	Agrupar valores de tipos diferentes
Acesso direto	<code>fst ("a", 10) -&gt; "a"</code>	Apenas para pares
Decomposição	<code>let (x, y) = (1, 2)</code>	Pattern matching
Tupla vazia	<code>()</code>	Apenas um valor, usada como <b>unit</b>

# Listas

## 1) Introdução a listas

Uma lista em Haskell é uma sequência de elementos do **mesmo tipo**.  
Elas são **imutáveis**, ou seja, não podem ser modificadas depois de criadas.

### Exemplos:

```
nums :: [Int]
nums = [1, 2, 3, 4]

letras :: [Char]
letras = ['a', 'b', 'c']
```

**[char]** é apenas um alias para **String**, mas em aplicações modernas é preferível **Text** (do pacote **text**).

## 2) Construção e implementação

Operador cons ( : )

O operador ( : ) adiciona um elemento no início de uma lista:

```
lista = 1 : 2 : 3 : [] -- Resultado: [1, 2, 3]
```

### Concatenação ( ++ )

O operador ( ++ ) concatena duas listas:

```
listaA = [1, 2]
listaB = [3, 4]
resultado = listaA ++ listaB -- [1, 2, 3, 4]
```

### Compreensão de listas

```
quadrados = [x*x | x <- [1..5]] -- [1, 4, 9, 16, 25]
pares = [x | x <- [1..10], even x] -- [2, 4, 6, 8, 10]
```



### 3 Listas e padrões

Você pode usar pattern matching para processar listas:

```
cabeca :: [a] -> a
cabeca [] = error "Lista vazia!"
cabeca (x:_) = x

cauda :: [a] -> [a]
cauda [] = []
cauda (_,xs) = xs
```

### 4 Sintaxe em padrões

Além de **x:xs**, podemos usar outras sintaxes para decompor listas:

```
primeirosDois :: [a] -> (a, a)
primeirosDois (x:y:_) = (x, y)
primeirosDois _ = error "Lista com menos de dois elementos!"
```

Também podemos usar o **padrão @** para nomear a lista inteira e ao mesmo tempo decompô-la:

```
analisar :: [Int] -> Text
analisar lista@(x:_) = "Primeiro: " <> show x <> ", lista completa: " <> show lista
analisar [] = "Lista vazia"
```

### 5 Textos

Embora **String** seja representada como **[Char]**, usar **Text** é melhor por eficiência:

```
import qualified Data.Text as T

saudacao :: T.Text -> T.Text
saudacao nome = "Olá, " <> nome <> "!"
```

## 6 Funções usuais para listas

Haskell possui várias funções prontas no módulo **Prelude**:

Função	Exemplo	Resultado
head	head [1, 2, 3]	1
tail	tail [1, 2, 3]	[2, 3]
length	length [1, 2, 3]	3
null	null [ ]	True
take	take 2 [1, 2, 3, 4]	[1, 2]
drop	drop 2 [1, 2, 3, 4]	[3, 4]
reverse	reverse [1, 2, 3]	[3, 2, 1]
map	map (*2) [1, 2, 3]	[2, 4, 6]
filter	filter odd [1, 2, 3, 4]	[1, 3]
foldl	foldl (+) 0 [1, 2, 3]	6
foldr	foldr (:) [] [1, 2]	[1, 2]

💡 **Dica:** Muitas funções de listas funcionam com **Text** através de conversões (**T.unpack**, **T.pack**).

📊 Tabela comparativa de Lista usando **Char** vs **Text** 📊

Característica	[Char] (String tradicional)	Text (do módulo <b>Data.Text</b> )
Representação	Lista de caracteres ([Char])	Estrutura otimizada com array imutável
Eficiência	Lenta para textos grandes	Muito mais eficiente para I/O e manipulação
Complexidade	Operações como <b>length</b> são O(n)	Operações internas otimizadas com memória contígua
Imutabilidade	Sim	Sim
Funções disponíveis	Padrão do <b>Prelude</b>	Precisa importar <b>Data.Text</b>
Conversão	Nativa	Requer <b>T.pack</b> e <b>T.unpack</b>
Uso Principal	Pequenos exemplos, código didático	Produção, manipulação de grandes textos



### Quando usar [Char]

- Em exemplos simples ou aprendizados
- Quando já está usando funções do **Prelude** que trabalham com listas
- Para operações pontuais de baixo custo em strings pequenas

#### Exemplo:

```
mensagem :: String
mensagem = "Olá, mundo!"
```



### Quando usar Text

- Para manipulação de texto em sistemas reais
- Ao lidar com arquivos grandes, APIs ou banco de dados
- Quando a performance é relevante

#### Exemplo:

```
import qualified Data.Text as T

saudacao :: T.Text -> T.Text
saudacao nome = "Olá, " <> nome <> "!"
```



### Conversões entre Strings e Text

```
import qualified Data.Text as T

-- String -> Text
texto :: T.Text
texto = T.pack "Olá, Haskell"

-- Text -> String
string :: String
string = T.unpack texto
```



### Regra prática

Comece com **Text** se estiver desenvolvendo algo em produção.  
Use **[Char]** apenas em protótipos ou aprendizado inicial.

# Funções

## 1 Funções de ordem superior

Uma *função de ordem superior* é aquela que:

- Recebe uma ou mais funções como argumento, ou
- Retorna uma função como resultado

Isso permite compor funções e escrever código mais genérico e reutilizável.

### Exemplos:

```
-- map aplica uma função a cada elemento de uma lista
dobrar :: [Int] -> [Int]
dobrar = map (*2) -- map é função de ordem superior

main :: IO ()
main = print (dobrar [1, 2, 3]) -- Saída: [2, 4, 6]
```

outro exemplo usando **filter**

```
pares :: [Int] -> [Int]
pares = filter even -- filter também é função de ordem superior
```

## 2 Funções anônimas (lambdas)

Funções anônimas (ou lambdas) são funções definidas sem nome.

São úteis quando você precisa passar uma função como argumento rapidamente.

### Sintaxe:

```
\x -> expressão
```

### Exemplo:

```
main :: IO ()
main = do
  print (map (\x -> x * 3) [1, 2, 3]) -- Saída: [3, 6, 9]
  print (filter (\x -> x > 5) [2, 5, 7, 10]) -- Saída: [7, 10]
```

💡 Lambdas também podem ter múltiplos parâmetros:

```
somaLambda = \x y -> x + y
```

### 3 Seções

Seções são uma forma de criar *funções parcialmente aplicadas* usando operadores.

**Exemplo:**

```
mais10 :: Int -> Int
mais10 = (+ 10) -- seção à direita

menos10 :: Int -> Int
menos10 = (10 -) -- seção à esquerda

main :: IO ()
main = do
    print (mais10 5) -- Saída: 15
    print (menos10 3) -- Saída: 7
```

isso é muito usado com operadores como `*`, `/`, `++` etc.

#### Resumo visual

Conceito	Exemplo	Resultado
Função de ordem superior	<code>map (*2) [1, 2, 3]</code>	<code>[2, 4, 6]</code>
Função anônima	<code>\x -&gt; x + 1</code>	função sem nome
seção	<code>(10 -)</code> e <code>(5 +)</code>	funções parciais

### 4 Composição de funções ( `.` )

O operador ( `.` ) combina duas funções, criando uma nova função.  
se temos **f** e **g**, então:

```
(f . g) x = f (g x)
```

Ou seja, a saída de **g** se torna a entrada de **f**

**Exemplo:**

```
import Data.Char (toUpper)

maiusculoReverso :: String -> String
maiusculoReverso = reverse . map toUpper

main :: IO ()
main = print (maiusculoReverso "haskell")
-- Saída: "LLEKSAH"
```

 Aqui **map toUpper** roda primeiro, e depois **reverse** é aplicado

## 5 Combinando tudo

Podemos juntar composição e aplicação para escrever pipelines:

```
import Data.Char (toUpper)

processar :: String -> String
processar = reverse . map toUpper . filter (/= ' ')

main :: IO ()
main = print $ processar "haskell é incrível"
-- Saída: "LEVICNÍÉLKSAH"
```

Aqui:

1. **filter** (/= ' ') remove espaços
2. **map toUpper** transforma em maiúsculas
3. **reverse** invert a string

## 6 Funções de alta ordem + composição

Exemplo com **map** e **filter** compondo funções de forma elegante:

```
paresDobrados :: [Int] -> [Int]
paresDobrados = map (*2) . filter even

main :: IO ()
main = print $ paresDobrados [1..10]
-- Saída: [4, 8, 12, 16, 20]
```

Note que não podemos escrever argumentos explícitos!

## 7 Uso com *foldr* e *foldl*

Podemos até mesmo criar pipelines funcionais mais complexos:

```
somaQuadrados :: [Int] -> Int
somaQuadrados = foldr ((+) . (^2)) 0

main :: IO ()
main = print $ somaQuadrados [1..5]
-- Saída: 55
```

## ✓ Resumo das técnicas

Conceito	Exemplo	Utilidade
Composição ( . )	$f . g \$ x$	Encadear funções
Aplicação ( \$ )	$f \$ g \$ h x$	Remover parênteses
Order superior	<b>map, filter, foldr</b>	Processar coleções de forma funcional

## 📌 Funções de primeira ordem comuns

### 1 2 3 4 Matemáticas

Função	Tipo	Descrição	Exemplo	
<b>succ</b>	<b>Enum a =&gt; a -&gt; a</b>	retorna o sucessor	<b>succ 4</b>	<b>5</b>
<b>pred</b>	<b>Enum a =&gt; a -&gt; a</b>	retorna o antecessor	<b>pred 4</b>	<b>3</b>
<b>abs</b>	<b>Num a =&gt; a -&gt; a</b>	valor absoluto	<b>abs (-3)</b>	<b>3</b>
<b>signum</b>	<b>Num a =&gt; a -&gt; a</b>	Sinal do número (-1,0,1)	<b>signum (-8)</b>	<b>-1</b>
<b>negate</b>	<b>Num a =&gt; a -&gt; a</b>	inverte o sinal	<b>negate 5</b>	<b>-5</b>

### a b c d Manipulação de listas e strings

Função	Tipo	Descrição	Exemplo	
<b>head</b>	<b>[a] -&gt; a</b>	primeiro elemento da lista	<b>head [1,2,3]</b>	<b>[1]</b>
<b>tail</b>	<b>[a] -&gt; [a]</b>	todos os elementos menos o 1º	<b>tail [1,2,3]</b>	<b>[2,3]</b>
<b>last</b>	<b>[a] -&gt; a</b>	último elemento da lista	<b>last [1,2,3]</b>	<b>3</b>
<b>init</b>	<b>[a] -&gt; [a]</b>	todos os elementos menos o último	<b>init [1,2,3]</b>	<b>[2,3]</b>
<b>length</b>	<b>[a] -&gt; Int</b>	comprimento da lista	<b>length [1,2,3]</b>	<b>3</b>
<b>null</b>	<b>[a] -&gt; Bool</b>	lista vazia	<b>null [ ]</b>	<b>True</b>
<b>reverse</b>	<b>[a] -&gt; [a]</b>	inverte uma lista	<b>reverse [1,2,3]</b>	<b>[3,2,1]</b>

### Acesso e busca

Função	Tipo	Descrição	Exemplo	
!!	[a] -> Int -> a	índice da lista	[10,20,30] !! 1	20
elem	Eq a => a -> [a] -> Bool	está na lista	elem 3 [1,2,3]	True
notElem	Eq a => a -> [a] -> Bool	não está na lista	notElem 4 [1,2,3]	True

### Concatenação e construção

Função	Tipo	Descrição	Exemplo	
( ++ )	[a] -> [a] -> [a]	concatenar listas	[1,2] ++ [3,4]	[1,2,3,4]
replicate	Int -> a -> [a]	repete o valor n vezes	replicate 3 'A'	"AAA"
take	Int -> [a] -> [a]	pega primeiros n elementos	take 2 [1,2,3]	[1,2]
drop	Int -> [a] -> [a]	remove n primeiros elementos	drop 2 [1,2,3]	[3]
splitAt	Int -> [a] -> ([a], [a])	divide a lista em duas	splitAt 2 [1,2,3,4]	( [1,2], [3,4] )

### Operações lógicas

Função	Tipo	Descrição	Exemplo	
not	Bool -> Bool	negação lógica	Not True	False
&&	Bool -> Bool -> Bool	E lógico	True && False	False

### Resumo

- **Primeira ordem:** funções que trabalham diretamente com valores (ex: **length**, **abs**, **head**).
- **Ordem superior:** funções que recebem ou retornam funções (ex: **map**, **filter**, **foldr**).



## ↑ Funções de Ordem superior mais comuns

### 1 Transformação de listas

Função	Tipo	Descrição	Exemplo	
map	$(a \rightarrow b) \rightarrow [a] \rightarrow [b]$	aplica função em cada elemento	map (*2) [1,2,3]	[2,4,6]
zipWith	$(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$	combina duas listas com função	zipWith (+) [1,2] [3,4]	[4,6]

### 2 Filtragem

Função	Tipo	Descrição	Exemplo	
filter	$(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$	filtra elementos	filter even [1..6]	[2,4,6]
takeWhile	$(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$	pega elementos até condição falhar	takeWhile (< 5) [1..10]	[1,2,3,4]
dropWhile	$(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$	remove elementos até condição falhar	dropWhile (<= 5) [1..10]	[6,7,8,9,10]

### 3 Redução

Função	Tipo	Descrição	Exemplo	
foldl	$(b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$	reduz lista da esquerda pra direita	foldl (+) 0 [1..4]	10
foldr	$(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$	reduz lista da direita para esquerda	foldr (:) [] [1,2]	[1,2]
scanl	$(b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$	similar ao foldl, mas mantém parciais	scanl (+) 0 [1,2,3]	[0,1,3,6]
scanr	$(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow [b]$	similar o foldl, mas mantém parciais	scanr (+) 0 [1,2,3]	[6,5,3,0]

### 4 Funções composição

Função	Tipo	Descrição	Exemplo	
(.)	$(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$	composição de função	(negate . abs) (-5)	-5
(\$)	$(a \rightarrow b) \rightarrow a \rightarrow b$	aplicação de função	sum & map (*2) [1..4]	20

## 5 Outras funções úteis

Função	Tipo	Descrição	Exemplo	
any	(a -> Bool) -> [a] -> Bool	algum elemento satisfaz?	any even [1,3,5,6]	True
all	(a -> Bool) -> [a] -> Bool	todos satisfazem	all (> 0) [1,2,3]	True
iterate	(a -> a) -> a -> [a]	lista infinita	take 5 \$ iterate (*2) 1	[1,2,4,8,10]
repeat	a -> [a]	lista infinita repetida	take 4 \$ repeat 7	[7,7,7,7]

## 🌲 Árvore binária

1 Uma árvore binária pode ser:

- Vazia
- Um **nó** com um valor e duas subárvores (esquerda e direita)

Exemplo:

```
data Arvore a
  = Vazia
  | No a (Arvore a) (Arvore a)
  deriving (Show, Eq)
```

Aqui:

- **Vazia** representa uma árvore sem elementos
- **No** representa um nó com valor (**a**) e duas subárvores

## 2 Criando árvores de exemplo

```
exemplo :: Arvore Int
exemplo =
  No 10
    (No 5 Vazia Vazia)
    (No 15
      (No 12 Vazia Vazia)
      Vazia)
```

Estrutura:

```
      10
     /  \
    5    15
     /
    12
```

### 3) Inserção em árvore de busca binária

Uma função recursiva para inserir elementos

```
inserir :: Ord a => a -> Arvore a -> Arvore a
inserir x Vazia = No x Vazia Vazia
inserir x (No valor esq dir)
  | x < valor = No valor (inserir x esq) dir
  | x > valor = No valor esq (inserir x dir)
  | otherwise = No valor esq dir -- não insere duplicados
```

Exemplo:

```
main :: IO ()
main = print $ inserir 8 exemplo
```

### 4) Busca em árvore binária

```
buscar :: Ord a => a -> Arvore a -> Bool
buscar _ Vazia = False
buscar x (No valor esq dir)
  | x == valor = True
  | x < valor  = buscar x esq
  | otherwise  = buscar x dir
```

### 5) Percursos em árvore binária

Percorrer uma árvore significa visitar seus nós em diferentes ordens:

**Pré-ordem (raiz -> esquerda -> direita)**

```
preOrdem :: Arvore a -> [a]
preOrdem Vazia = []
preOrdem (No v e d) = [v] ++ preOrdem e ++ preOrdem d
```

**Em ordem (esquerda -> raiz -> direita)**

```
emOrdem :: Arvore a -> [a]
emOrdem Vazia = []
emOrdem (No v e d) = emOrdem e ++ [v] ++ emOrdem d
```

## Pós-ordem (esquerda -> direita -> raiz)

```
posOrdem :: Arvore a -> [a]
posOrdem Vazia = []
posOrdem (No v e d) = posOrdem e ++ posOrdem d ++ [v]
```

### ✓ Resumo

- Árvores binárias são recursivas.
- Podemos construir funções como **inserir**, **buscar** e percursos usando pattern matching
- Árvores podem ser especializadas( ex: árvores de busca binária) usando constraints (**Ord**).

## Filas

### 1 Definição de uma fila

Podemos definir fila de várias maneiras. A mais simples usa listas:

```
data Fila a = Fila [a] deriving (Show)
```

Mas essa abordagem é ineficiente para remoções(**O(n)**).

Uma forma melhor é usar duas listas:

- uma para entrada (**entrada**)
- uma para saída (**saida**)

```
data Fila a = Fila [a] [a] deriving (Show)
```

### 2 Criando fila vazia

```
filaVazia :: Fila a
filaVazia = Fila [] []
```

### 3 Inserir elemento na fila (enqueue)

Adicionamos um elemento no final da fila (na lista **entrada**)

```
enfileirar :: a -> Fila a -> Fila a
enfileirar x (Fila entrada saida) = Fila (x:entrada) saida
```

#### 4 Remover elemento da fila (dequeue)

se **saida** estiver vazia, invertemos **entrada**:

```
desenfileirar :: Fila a -> (Maybe a, Fila a)
desenfileirar (Fila entrada (x:xs)) = (Just x, Fila entrada xs)
desenfileirar (Fila [] [])          = (Nothing, filaVazia)
desenfileirar (Fila entrada [])     = desenfileirar (Fila [] (reverse entrada))
```

#### 5 Espiar o próximo da fila (peek)

```
frente :: Fila a -> Maybe a
frente (Fila _ (x:_)) = Just x
frente (Fila [] [])   = Nothing
frente (Fila entrada []) = frente (Fila [] (reverse entrada))
```

Exemplo de uso:

```
main :: IO ()
main = do
  let f0 = filaVazia
  let f1 = enfileirar 1 f0
  let f2 = enfileirar 2 f1
  let (x, f3) = desenfileirar f2
  print x      -- Just 1
  print $ frente f3 -- Just 2
```

#### ✓ Resumo

- Fila eficiente em Haskell -> duas listas (**entrada** e **saida**)
- **enfileirar** adiciona no início de **entrada**
- **desenfileirar** lê de **saida**, e quando vazio, inverte **entrada**

## Tipos genéricos predefinidos

#### 10 que são tipos genéricos

Tipos genéricos são tipos que usam parâmetros de tipo. Em vez de criar um tipo específico para cada caso, usamos variáveis de tipo para que a estrutura funcione com qualquer tipo de dado.

Exemplo de variável de tipo:

```
-- 'a' é um parâmetro de tipo
data Caixa a = Caixa a deriving Show
```

Podemos criar:

```
c1 :: Caixa Int
c1 = Caixa 10

c2 :: Caixa Text
c2 = Caixa "Haskell"
```

## 2 Tipos genéricos predefinidos em Haskell

### ✓ Listas

```
-- Tipo: [a]
numeros :: [Int]
numeros = [1, 2, 3]

palavras :: [Text]
palavras = ["Haskell", "é", "legal"]
```

**a** é genérico, logo **[Int]**, **[Text]** ou **[Bool]** são todas listas, só muda o tipo de elemento.

### ✓ Tuplas

```
-- Tipo: (a, b)
pessoa :: (Text, Int)
pessoa = ("Alice", 30)
```

tuplas podem ter qualquer combinações de tipos: **(Int, Bool)**, **(Text, Double, Bool)** etc.

### ✓ Maybe

O tipo **Maybe** representa um valor que pode existir ou não.

```
-- Tipo: Maybe a
nome :: Maybe Text
nome = Just "Haskell"

semNome :: Maybe Text
semNome = Nothing
```

Usado para lidar com ausência de valores **null**.

### ✓ Either

Representa dois possíveis resultados: sucesso (**Right**) ou erro (**Left**)

```
-- Tipo: Either e a
dividir :: Double -> Double -> Either Text Double
dividir _ 0 = Left "Divisão por zero"
dividir x y = Right (x / y)
```

### ✓ IO

**IO a** representa uma ação que realiza efeitos colaterais e retorna um valor do tipo **a**.

```
-- Tipo: IO a
main :: IO ()
main = putStrLn "Olá, mundo!"
```

### ✓ Funções

Funções também são genéricas.

```
-- Tipo: (a -> b)
aplicar :: (a -> b) -> a -> b
aplicar f x = f x
```

## 3 Polimorfismo

A grande força dos tipos genéricos é permitir funções polimórficas:

```
identidade :: a -> a
identidade x = x
```

**identidade** funciona com qualquer tipo (**Int**, **Text**, **Bool**, etc).

### ✓ Resumo dos principais tipos genéricos

Tipo	Exemplo	Descrição
[a]	[1,2,3]	Lista de qualquer tipo
(a,b)	("Haskell", 30)	Tupla de dois valores
Maybe a	Just a / Nothing	valor opcional
Either e a	Right 42 / Left "Erro"	sucesso ou erro
IO a	putStrLn "oi"	Ação com efeitos colaterais
a -> b	(*2)	Função genérica

## Classes de tipos

### 10 que são classes de tipos

Em Haskell, classes de tipos são como interfaces em outras linguagens: elas definem um conjunto de funções que um tipo deve implementar para ser uma instância dessa classe.

#### Por exemplo:

- A classe **Eq** define a capacidade de comparar igualdades entre valores.
- A classe **Show** define a capacidade de converter valores em **Text**.

Se um tipo pertence a uma classe, significa que ele implementa os métodos definidos por ela.

### 2 Fundamentos genéricos predefinidos

Aqui estão as principais classes de tipos pré-definidas em Haskell.

#### Eq

Permite verificar igualdade e desigualdade

```
(==) :: Eq a => a -> a -> Bool
(/=) :: Eq a => a -> a -> Bool

main :: IO ()
main = do
  print (5 == 5)    -- True
  print ('a' /= 'b') -- True
```

#### Ord

Define ordenação (depende de **Eq**).

```
(<)  :: Ord a => a -> a -> Bool
(>)  :: Ord a => a -> a -> Bool
(<=) :: Ord a => a -> a -> Bool
(>=) :: Ord a => a -> a -> Bool

main :: IO ()
main = print (10 > 7) -- True
```

#### Show

converte valores para **Text** (geralmente usado para exibir resultados).

```
show :: Show a => a -> String

main :: IO ()
main = print (show 123) -- "123"
```



## Read

Converte **Text** para um tipo de dado.

```
read :: Read a => String -> a

main :: IO ()
main = print (read "42" :: Int) -- 42
```

## Num

Classe para tipos numéricos

```
(+) :: Num a => a -> a -> a
(-) :: Num a => a -> a -> a
(*) :: Num a => a -> a -> a

main :: IO ()
main = print (3 + 4) -- 7
```

## Integral

Subclasse de **Num** para inteiros (**Int** e **Integer**).

```
div :: Integral a => a -> a -> a
mod :: Integral a => a -> a -> a

main :: IO ()
main = print (div 10 3) -- 3
```

## Fractional

Subclasse de **Num** para números fracionários (**Float** e **Double**)

```
(/) :: Fractional a => a -> a -> a

main :: IO ()
main = print (7 / 2) -- 3.5
```

## Enum

Para tipos que podem ser enumerados.

```
main :: IO ()
main = do
  print [1..5]      -- [1,2,3,4,5]
  print ['a'..'e']  -- "abcde"
```

## Bounded

Define valores mínimos e máximos de um tipo.

```
minBound :: Bounded a => a
maxBound :: Bounded a => a

main :: IO ()
main = do
  print (minBound :: Int)
  print (maxBound :: Int)
```

## Functor e Monad (avançado)

Essas classes são usadas para trabalhar com contextos (listas, **Maybe**, **IO**, etc), mas entram em um nível mais avançado.

### Exemplo com Functor:

```
fmap :: Functor f => (a -> b) -> f a -> f b

main :: IO ()
main = print (fmap (*2) [1,2,3]) -- [2,4,6]
```

## Resumo

Classes de tipos são genéricas e permitem reutilizar funções para diferentes tipos. Quando você vê uma assinatura como:

```
(==) :: Eq a => a -> a -> Bool
```

significa que a função ( == ) funciona para qualquer tipo **a**, desde que **a** pertença à classe **Eq**.

## 3 Criando uma classe de tipos

A sintaxe básica é:

```
class NomeDaClasse a where
  funcao1 :: a -> Tipo
  funcao2 :: a -> a -> Tipo
```

**a** é um parâmetro de tipo. A classe define um “contrato”: qualquer tipo que fizer parte dessa classe precisa implementar essas funções.

### Exemplo simples:

Vamos criar uma classe **Exibivel** que converta um valor em **Text**.

```
{-# LANGUAGE OverloadedStrings #-}
import Data.Text (Text, pack)

class Exibivel a where
    exhibir :: a -> Text
```

Agora, podemos criar instâncias dessa classe para diferentes tipos

### 4 Criando instâncias

```
instance Exibivel Int where
    exhibir n = pack ("Número: " ++ show n)

instance Exibivel Bool where
    exhibir True  = "Verdadeiro"
    exhibir False = "Falso"
```

### Usando classe:

```
main :: IO ()
main = do
    putStrLn (show (exibir (42 :: Int))) -- "Número: 42"
    putStrLn (show (exibir True))       -- "Verdadeiro"
```

Cada tipo pode ter sua própria implementação

### 5 Criando tipos customizados com instâncias

Podemos criar tipo **Pessoa** e torná-lo instância de **Exibivel**

```
data Pessoa = Pessoa { nome :: Text, idade :: Int }

instance Exibivel Pessoa where
    exhibir (Pessoa n i) = "Nome: " <> n <> ", Idade: " <> pack (show i)

main :: IO ()
main = putStrLn (show (exibir (Pessoa "Ana" 25)))
-- Saída: "Nome: Ana, Idade: 25"
```

## 6 Herança de classes de tipos

Uma classe pode depender de outra.

**Exemplo: Comparavel** que depende de **Eq**.

```
class Eq a => Comparavel a where
    maiorQue :: a -> a -> Bool
    maiorQue x y = not (x == y) -- implementação padrão
```

Agora qualquer tipo que fizer parte de **Comparavel** também precisa ser instância de **Eq**.

## ✓ Resumo

- **class** define uma classe de tipos
- **instance** cria implementações específicas para tipos.
- Você pode ter implementações padrão dentro da classe.
- É possível criar hierarquias de classes (ex: **Ord** depende de **Eq**).

## Deriving

### 10 que é deriving

Quando você cria um tipo em Haskell, pode usar a palavra-chave **deriving** para pedir ao compilador que gere automaticamente implementações de algumas classes padrão, como **Eq**, **show**, **Ord**, **Enum**, **Bounded**, entre outras.

Isso evita que você escreva manualmente funções como **==** ou **show** para seus tipos

**Exemplo básico:**

```
data Pessoa = Pessoa
    { nome    :: String
    , idade  :: Int
    } deriving (Show, Eq)

main :: IO ()
main = do
    print (Pessoa "Ana" 25)           -- Usa Show automaticamente
    print (Pessoa "Ana" 25 == Pessoa "Ana" 25) -- Usa Eq automaticamente
```

## ✓ O GHC gera:

- Uma instância de **Show** para exibir **Pessoa**
- Uma instância de **Eq** para comparar **Pessoa**

## 2 Derivando múltiplas classes

Você pode derivar várias classes ao mesmo tempo:

```
data Cor = Vermelho | Verde | Azul
  deriving (Show, Eq, Ord, Enum, Bounded)

main :: IO ()
main = do
  print Vermelho           -- "Vermelho"
  print (Vermelho < Azul)  -- True (ordenação automática)
  print [minBound .. maxBound :: Cor] -- [Vermelho, Verde, Azul]
```

Aqui:

- **Show**: imprime as cores como texto.
- **Eq**: permite comparar igualdade.
- **Ord**: permite ordenar
- **Enum**: permite gerar listas ([Vermelho .. Azul])
- **Bounded**: fornece **minBound** e **maxBound**.

## 3 Customizando deriving

Com extensões como **DerivingStrategies** e **DerivingVia**, você pode controlar como o **deriving** funciona, mais isso é avançado e só é útil para cenários específicos.

**Exemplo com DerivingStrategies:**

```
{-# LANGUAGE DerivingStrategies #-}

newtype Idade = Idade Int
  deriving stock (Eq, Show)
```

### 🚫 Quando NÃO usar deriving

- quando você precisa de uma implementação personalizada (ex: exibir **Pessoa** de forma formatada).
- Quando quer restringir ou modificar o comportamento padrão

Nesse caso, você escreve manualmente a instância usando **instance**.

# Functors

## 1 Fundamentos de Functors

Um functor é algo sobre o qual você pode “mapear” uma função. Ele é definido pela classe de tipos.

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

**f** é um tipo que recebe um parâmetro (ex: **[ ]**, **Maybe**, **IO**)

**fmap** aplica uma função ao valor dentro do functor, sem remover o contexto

### Exemplo functor com Maybe

```
main :: IO ()
main = do
    print (fmap (*2) (Just 5)) -- Just 10
    print (fmap (*2) Nothing) -- Nothing
```

Aqui:

- **fmap (2) (Just 5)** aplica (\*2) ao valor de 5.
- **Nothing** não tem valor interno, então nada acontece.

### Exemplo com listas:

```
main :: IO ()
main = print (fmap (*2) [1, 2, 3]) -- [2, 4, 6]
```

**fmap** aqui é equivalente a **map**.

### Exemplo com IO:

```
main :: IO ()
main = do
    resultado <- fmap (++ "!") (return "Olá")
    putStrLn resultado -- "Olá!"
```

Mesmo dentro de **IO**, conseguimos transformar o valor.

## ✓ Resumo até aqui

- **fmap** aplica funções dentro de contextos.
- Ele não “quebra” o contexto (**Just** continua **Just**, **IO** continua **IO**).

## Entrada e saída (I/O)

### 1 Conceito de IO

Em Haskell, qualquer função que tenha efeitos colaterais retorna um valor do **IO a**, onde:

- **a** é o tipo do resultado (por exemplo, **IO Int** ou **IO ()**).
- **IO** é um “contexto” que encapsula a ação e garante que ela seja executada de forma controlada.

#### Exemplo simples:

```
main :: IO ()
main = putStrLn "Olá, mundo!"
```

Aqui:

- **putStrLn :: String -> IO ()**
- O tipo **()** indica que não há valor de retorno significativo, apenas o efeito colateral.

### 2 Leitura de dados

Para ler valores do usuário, usamos funções como **getLine**:

```
main :: IO ()
main = do
    putStrLn "Digite seu nome:"
    nome <- getLine
    putStrLn ("Olá, " ++ nome ++ "!")
```

Note o uso de **<-** para extrair o valor do contexto **IO**.

### 3 Escrita de dados

Além de **putStrLn**, temos outras funções:

- **putStr :: String -> IO () ->** imprime sem quebra de linha
- **print :: String a => a -> IO () ->** imprime qualquer valor que tenha instância de **Show**

#### Exemplo:

```
main :: IO ()
main = do
    putStr "Número: "
    print (42 :: Int)
```

#### 4 Leitura com conversão

Para ler números, usamos **read**:

```
main :: IO ()
main = do
    putStrLn "Digite um número:"
    entrada <- getLine
    let numero = read entrada :: Int
    print (numero * 2)
```

⚠ **read** gera erro se a entrada não for válida. Para evitar isso, podemos usar **readMaybe** do módulo **Text.Read**

#### 5 Ações sequenciais com **do**

O bloco **do** permite encadear várias ações **IO** de forma legível:

```
main :: IO ()
main = do
    putStrLn "Digite dois números:"
    a <- readLn
    b <- readLn
    putStrLn ("A soma é " ++ show (a + b))
```

Aqui:

- **readLn** é uma combinação de **getLine** + **read**

#### 6 Trabalhando com arquivos

Haskell também tem funções para leitura e escrita de arquivos:

```
main :: IO ()
main = do
    writeFile "saida.txt" "Olá, arquivo!"
    conteudo <- readFile "saida.txt"
    putStrLn conteudo
```

**writeFile :: FilePath -> String -> IO ()**

**readFile :: FilePath -> IO String**



## 6 Combinando I/O com funções puras

Boas práticas:

- Mantenha funções puras separadas de I/O:
- Faça a lógica de negócio em funções puras e use **I/O** apenas no **main**.

### Exemplo

```
dobrar :: Int -> Int
dobrar x = x * 2

main :: IO ()
main = do
    putStrLn "Digite um número:"
    n <- readLn
    print (dobrar n)
```

■ Quando falamos de API's em Haskell, temos algumas bibliotecas bem consolidadas que ajudam a criar, consumir e testar API's REST ou GraphQL.

### 1) Servant (para criar APIs)

O Servant é um dos frameworks mais populares para criação de APIs em Haskell. Ele usa tipos para definir endpoints, o que garante segurança em tempo de compilação.

Exemplo:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}

import Servant
import Network.Wai.Handler.Warp (run)

-- Definição da API
type API = "hello" :> Get '[PlainText] String

-- Implementação do servidor
server :: Server API
server = return "Olá, API com Haskell!"

main :: IO ()
main = run 8080 (serve (Proxy :: Proxy API) server)
```

✓ Agora, acessando <http://localhost:8080/hello>, você recebe “Olá, API com Haskell”

Pacotes relacionados:

- **servant**
- **servant-server**
- **servant-client** (para consumir APIs)
- **servant-swagger** (para documentação)

## 2 Scotty (para APIs simples)

O Scotty é inspirado no framework **Sinatra** (Ruby). Ele é minimalista e ótimo para criar APIs rapidamente

**Exemplo:**

```
{-# LANGUAGE OverloadedStrings #-}
import Web.Scotty

main :: IO ()
main = scotty 3000 $ do
  get "/hello" $ text "Olá, Scotty!"
```

✓ Rodar e acessar <http://localhost:3000/hello>

**Pacotes relacionados:**

- scotty
- **way** (base para aplicações web)
- **warp** (servidor web)

## 3 Yesod (para aplicações web completas)

O Yesod é um framework mais robusto, ideal para aplicações web grandes, com roteamento, templates e autenticação.

**Exemplo básico:**

```
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE QuasiQuotes #-}
import Yesod

data App = App

mkYesod "App" [parseRoutes|
  / HomeR GET
  |]

instance Yesod App

getHomeR :: Handler Html
getHomeR = defaultLayout [whamlet|<h1>Olá, Yesod!|]

main :: IO ()
main = warp 3000 App
```

#### 4 Http-client e Req (para consumir APIs)

Se você quiser consumir APIs externas, pode usar:

##### Http-client

```
import Network.HTTP.Simple

main :: IO ()
main = do
    response <- httpLBS "https://api.github.com"
    print (getResponseBody response)
```

##### req (interface mais simples)

```
{-# LANGUAGE OverloadedStrings #-}
import Network.HTTP.Req

main :: IO ()
main = runReq defaultHttpConfig $ do
    r <- req GET (https "api.github.com") NoReqBody jsonResponse mempty
    liftIO $ print (responseBody r :: Value) -- usa Aeson para JSON
```

#### 5 Aeson (Json)

Quase todas as APIs usam JSON. O Aeson é a biblioteca padrão para trabalhar com JSON.

```
{-# LANGUAGE DeriveGeneric #-}
import Data.Aeson
import GHC.Generics

data User = User { name :: String, age :: Int }
    deriving (Show, Generic)

instance FromJSON User
instance ToJSON User
```

Com isso, você pode converter valores de/para JSON automaticamente

## ✓ Resumo

Pacote	Uso
<b>servant</b>	Criação de APIs tipadas
<b>scotty</b>	APIs rápidas e simples
<b>yesod</b>	framework web completo
<b>http-client</b>	Consumo de APIs em baixo nível
<b>req</b>	Consumo de APIs de forma simplificada
<b>aeson</b>	Conversão de JSON