



Haskell

▼ Tipos:

▼ Booleanos:

True

False

▼ Literais:

String

Char

▼ Numéricos:

Inteiros

Fracionários

Funções:

▼ Funções devem ser usadas seguindo o seguinte padrão:

Nome da função seguido pelo argumento



Ex: `sqrt 25` (essa função devolve a raiz quadrada do argumento em questão, nesse caso ela retornaria `5`).

▼ Funções com operadores infixos:

A função é escrita entre os seus argumentos:



Ex: `2 + 2` (Função [operador] que retorna a soma de dois argumentos, retorna `4` nesse caso.)

▼ Precedência:




É a ordem em que as funções são executadas:

precedência	associativade	operador	descrição
9	esquerda	!!	índice de lista
	direita	.	composição de funções
8	direita	[^]	potenciação com expoente inteiro não negativo
		^{^^}	potenciação com expoente inteiro
		^{**}	potenciação com expoente em ponto flutuante
7	esquerda	*	multiplicação
		/	divisão fracionária
		'div'	quociente inteiro truncado em direção a $-\infty$
		'mod'	módulo inteiro satisfazendo $(\text{div } x \ y) * y + (\text{mod } x \ y) == x$
		'quot'	quociente inteiro truncado em direção a 0
		'rem'	resto inteiro satisfazendo $(\text{quot } x \ y) * y + (\text{rem } x \ y) == x$
6	esquerda	+	adição
		-	subtração
5	direita	:	construção de lista não vazia
		++	concatenação de listas
4	não associativo	==	igualdade
		/=	desigualdade
		<	menor que
		<=	menor ou igual a
		>	maior que
		>=	maior ou igual a
		'elem'	pertinência de lista
		'notElem'	negação de pertinência de lista
3	direita	&&	conjunção (e lógico)
2	direita		disjunção (ou lógico)
1	esquerda	>>=	composição de ações sequenciais
		>>	composição de ações sequenciais (ignora o resultado da primeira)
0	direita	\$	aplicação de função
		\$!	aplicação de função estrita
		'seq'	avaliação estrita




▼ Escolpo:

Os códigos em haskell seguem o seguinte modelo de escolpo:

Em uma **seqüência de definições**, cada definição deve começar precisamente na *mesma coluna*:

<pre>a = 10 b = 20 c = 30</pre>	<pre>a = 10 b = 20 c = 30</pre>	<pre> a = 10 b = 20 c = 30</pre>
		

Se uma definição for escrita em mais de uma linha, as linhas subsequentes à primeira devem começar em uma coluna mais à direita da coluna que começa a seqüência de definições.

<pre>a = 10 + 20 + 30 + 40 + 50 + 60 + 70 + 80 b = sum [10,20,30]</pre>	<pre> a = 10 + 20 + 30 + 40 + 50 + 60 + 70 + 80 b = sum [10,20,30]</pre>	<pre> a = 10 + 20 + 30 + 40 + 50 + 60 + 70 + 80 b = sum [10,20,30]</pre>
		

▼ Definições locais em funções:



Pode-se usar a clausa `where` para fazer uma definição local em uma função.

▼ Ex:

```
somaDobro :: Int -> Int
somaDobro a = s+s
  where s = a*2

somaDobro a = (a*2)+(a*2)
```

💡 Essa função faz o dobro da entrada (`a`) e após isso ela faz o dobro desse resultado (`s`).

💡 Para evitar a redundância de ter que fazer `a*2` duas vezes foi usado o comando `where` para fazer uma definição de variável de forma local. Ou seja, essa variável '`s`' só funcionará na função em questão (`somaDobro`)

Condicional:

💡 Estruturas condicionais são estruturas que permitem uma bifurcação no código. Ou seja, permite a escolha entre duas ou mais opções com base em uma condição pré-definida.

▼ If then else:

▼ Forma da expressão:

💡 `if` condição then exp¹ else ²

▼ Destrinchando:

💡 OBS: Relevem o meu inglês, ele é péssimo.

- O `if` é o mais conhecido dessa expressão, visto que ele é usado em grande parte das outras linguagens de programação. E o seu significado é 'se', portanto ele é o responsável por exemplificar a condição, que vem após eles. Dessa forma ele em conjunto com a condição são como o recepcionista do condicional, que decide e indica para onde você deve ir.
- A 'condição' é exatamente o que o seu nome diz. Ou seja, dependendo de seu resultado é possível colocar o código para um caminho diferente. Esse caminho diferente é 'escolhido' em caso a condição esteja sendo cumprida (**True**) ou não (**False**).

💡 A condição também pode ser chamada de predicado. Ou seja, é uma expressão booleana que deve ter como resultado '**true**' ou '**false**'

- O `then` pode ser definido como 'então', dessa forma ele apresenta um dos possíveis caminhos que deve ser seguido em sua função. Especificadamente no caso do `then` ele é responsável por definir o caminho em caso de (**True**), a menos que um `not` venha mudar isso, nesse caso ele passa a definir o caminho em caso de (**False**).

- O 'exp1' e o 'exp2' são respectivamente a primeira e a segunda consequência de seu condicional.
- O `else` também é bastante conhecido devido ao seu uso em outras linguagens de programação. Dessa forma ele é definido como 'senão', ou seja, ele faz o caminho inverso do `if`. Normalmente é usado caso a sua condição retorne (**False**), assim como no `if` existe a possibilidade de um `not` inverter isso.

▼ Exemplos:

▼ Ex 1:

```
maior :: Int -> Int -> Int
maior a b = if a > b then a else b
```

▼ Ex 2:

```
iguais :: Int -> Int -> Bool
iguais a b = if a == b then True else False
```

▼ Ex 3:

```
senal :: Int -> Int
senal n = if n < 0
then -1
else if n == 0
then 0
else 1
```

▼ Equações com guardas:



Funções podem ser definidas com o uso de guardas ' | '. Onde uma sequência de expressões lógicas é usada para escolher entre vários resultados.

▼ Estrutura da expressão:


```
função arg^1 ... arg^n
|guarda^1 = exp1
|guarda^n = expn
```

▼ Destrinchando:


- A 'função' é o nome dado a função a qual você se refere, ou cria.
- Os "arg^1 ... arg^n", são os diversos argumentos que aquela função irá receber.
- As "guarda^1 ... guarda^n", são as diversas condições que você ira usar para bifurcar o seu código.
- Os "exp1 ... expn", podem ser definidos como os resultados que são consequência da condição definida na guarda.

▼ Informações importantes:

1. Cada guarda deve ser uma expressão lógica, portanto deve devolver um booleano.
2. Os resultados, consequência das guardas, devem ser todos do mesmo tipo.
3. As guardas são verificadas na sequência em que foram escritas, dessa forma a primeira condição satisfeita é a que sinaliza o resultado a ser retornado.


 Lembrem-se de seguir o modelo de escopo do Haskell.

▼ Comando `otherwise`:

 O comando `otherwise` é semelhante ao `else` do `if then else`. Dessa forma ele é executado caso nenhuma das condições anteriores seja satisfeita.

| Código do comando "otherwise":

```
otherwise :: Bool
otherwise = True.
```

 Não é obrigatório vocês conhecerem a definição de todas as funções, entretanto é extremamente útil conhecer.

▼ Exemplos:

▼ Ex 1:

```
maior :: Int -> Int -> Int
maior a b
  | a > b = a
  | otherwise = b


-- " -- " É usado para fazer comentários em 1 linha no haskell.
{- Eu usei o "otherwise" no fim do código para evitar fazer uma guarda a mais
com a condição de " b > a ", isso deixou meu código menor, visto que eu também
teria que fazer outra condição para caso " b == a ".
-}

-- Ps: " {- -}" pode ser usado para fazer comentarios com mais de 1 linha
```

▼ Ex 2:

```
analisaImc :: Float -> Float -> String
analisaImc peso altura
  | imc <= baixo = "IMC baixo, cuidado!"
  | imc <= normal = "IMC normal."
  | imc <= alto = "IMC alto, cuidado!"
  | otherwise = "IMC muito alto, muito cuidado!"
  where
    imc = peso / (altura^2)
    baixo = 18.5
    normal = 25
    alto = 30
```

Tuplas:

 Tuplas são um tipo de estrutura de dados formado por uma sequência de valores, muitas vezes de tipos diferentes.



Um elemento de uma tupla é identificado pela posição em que ele ocorre na tupla.

▼ Estrutura da tupla:

```
(exp^1, exp^2, ..., exp^n)
```



Onde cada exp representa um valor usado na tupla.

▼ Tupla Vazia:



Uma tupla vazia é uma tupla que não contém elemento nenhum, ela é definida da seguinte forma: `()`



Não existe uma tupla contendo apenas um elemento. Exemplo: `(1)` ou `("a")`

▼ Exemplos de tuplas:

tupla	tipo
<code>('A', 't')</code>	<code>(Char, Char)</code>
<code>('A', 't', 'o')</code>	<code>(Char, Char, Char)</code>
<code>('A', True)</code>	<code>(Char, Bool)</code>
<code>("Joel", 'M', True, "COM")</code>	<code>(String, Char, Bool, String)</code>
<code>(True, ("Ana", 'f'), 43)</code>	<code>Num a => (Bool, (String, Char), a)</code>
<code>()</code>	<code>()</code>
<code>("nao eh tupla")</code>	<code>String</code>

▼ Algumas operações predefinidas no prelúdio:

1. `fst` : Seleciona o primeiro componente de um par.

```
fst ("Vinicius" 20)

--Isso retornará "Vinicius", quando executado no prelúdio
```

2. `snd`: Seleciona o segundo componente de um par.

```
snd ("Lucas" 20)

--Isso retornará 20, quando executado no prelúdio
```

Listas:



Listas em Haskell são definidas como um tipo de estrutura formado por uma sequência de valores, do mesmo tipo.



Um elemento de uma lista é identificado pela posição em que ele ocorre na lista.

▼ Estrutura da lista:

```
[exp^1, exp^2, ..., exp^n]
```



Onde cada exp representa um elemento da lista.

▼ Maneira alternativa de criar uma lista:



É possível criar uma lista de formas mais práticas.

Exemplo 1: `[1 .. 10]`

Retorna a lista: `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`

Exemplo 2: `[2, 4 .. 10]`

Retorna a lista: `[2, 4, 6, 8, 10]`



Explicando o "Exemplo 2": Ao definir uma lista como `[a, b .. c]`, isso irá retornar uma lista composta pelos elementos de `a` até `c`, passo que `b - a`.

- OBS: O último elemento da lista deverá ser o maior e ele será igual a `c` ou o mais próximo possível, segundo a condição definida.

▼ Lista Vazia:

1. Não contem nenhum elemento.
2. É denotada pelo construtor constante: `[]`

▼ Lista não vazia:

1. Contém pelo menos um elemento.
2. É formada por uma cabeça " head " (que é o primeiro elemento da lista) e por uma cauda " tail " (uma lista com os demais elementos da lista.).
3. É contruída pelo construtor `:` um operador binário infixo com associatividade à direita e precedência 5 (imediatamente inferior à precedência dos operadores aditivos (+) e (-))
 - O operando da esquerda é a cabeça da lista
 - O operando da direita é a cauda da lista

▼ Exemplos:

Por exemplo, a lista formada pela sequência dos valores 1, 8, e 6 pode ser escrita como

lista
<code>[1, 8, 6]</code>
<code>1 : [8, 6]</code>
<code>1 : (8 : [6])</code>
<code>1 : (8 : (6 : []))</code>
<code>1 : 8 : 6 : []</code>

notação estendida	notação básica
<code>['O', 'B', 'A']</code>	<code>'O' : 'B' : 'A' : []</code>
<code>['B', 'A', 'N', 'A', 'N', 'A']</code>	<code>'B' : 'A' : 'N' : 'A' : 'N' : 'A' : []</code>
<code>[False, True, True]</code>	<code>False : True : True : []</code>
<code>[[False], [], [True, False, True]]</code>	<code>(False : []) : [] : (True : False : True : []) : []</code>
<code>[1., 8., 6., 10.48, -5.]</code>	<code>1. : 8. : 6. : 10.48 : -5. : []</code>

▼ Algumas funções predefinidas no prelúdio:

1. `null`: Verifica se uma lista é nula.

```
null [ ]
--Retornará um booleano True

null [1, 2, 3]
--Retornará um booleano False
```

2. `head`: Seleciona a cabeça da lista.

```
head [1, 2, 3, 4, 5]
--Retornará o elemento 1 como resultado
```

3. `tail`: Seleciona a cauda da lista.

```
tail [1, 2, 3, 4, 5]
--Retornará [2, 3, 4, 5]
```

4. `length`: Calcula o tamanho de uma lista. (Quantidade de elementos)

```
length [1, 2, 3, 4, 5]
--Retornará 5 como resultado
```

5. `!!`: Seleciona o *i*-ésimo elemento de uma lista. ($0 \leq i < n$, onde *n* é o comprimento da lista)

```
[1, 2, 3, 4, 5] !! 2
--Retornará o elemento 3
--Lembre-se que a contagem começa com o índice em 0
```

6. `take`: Seleciona os primeiros '*n*' elementos de uma lista.

```
take 4 ['a', 'b', 'c', 'd', 'e']
--Retornará ['a', 'b', 'c', 'd']
```


7. drop: Remove os primeiros ' n ' elementos de uma lista.

```
drop 2 [1,2,3,4,5]
--Retornará [3,4,5]
```

8. sum: Calcula a soma dos elementos de uma lista de inteiros ou float

```
sum [1,2,3,4,5]
--Retornará 15
```

9. product: Calcula o produto dos elementos de uma lista de inteiros ou float.

```
product [2,3,4,5,10]
--Retornará 1200
```

10. (++): Concatena duas listas.

```
['D', 'i'] ++ ['d', 'i']
--Retornará "Didi"
--Lembre-se que uma String é uma lista formada por Char
```

11. reverse: Inverte a ordem dos elementos de uma lista.

```
reverse [1,2,3,4,5]
--Retornará [5,4,3,2,1]
```

12. zip: Junta duas listas em uma única lista formada pelos pares dos elementos correspondentes. (Retorna uma lista formada por pares de Tuplas.

```
zip ["Vinicius", "Mariana", "Sergio", "Jane", "Gustavo"] [1, 2, 3, 4, 5]
--Retornará [("Vinicius", 1), ("Mariana", 2), ("Sergio", 3), ("Jane",4), ("Gustavo",5)]
```

▼ Compreensão de listas:



A compreensão de listas é uma maneira de definir uma lista inspirada na notação de conjuntos.

Exemplo Inicial:

- Supondo uma lista de forma = `[1, 7, 14]` e iremos chamar essa lista de **y**

```
[2*a | a<-y]
```

```
--Essa função retornará a lista [2, 14, 28]
```

Explicando a forma:

- Na *list Comprehension* o `a <- y` é chamado de *generator* (gerador), pois ele gera os dados em que os resultados são construídos. Os geradores podem ser combinados com predicados (*predicates*) que são funções que devolvem valores booleanos `a->Bool`.

▼ Exemplo:

```
[a | a <- [2,3,4,5], even a]

--Isso retornará uma lista com todos os elementos pares da lista passada.
--Lista retornada = [2,4]
--Isso ocorre por o comando "even" retornar True sempre que um elemento é par
```

▼ Exemplo com Tupla:

```
somaPares:: [(Int,Int)]->[Int]
somaPares lista = [a+b|(a,b)<-lista]

{-Isso retornará uma lista onde cada elemento é a soma dos elementos de uma
tupla-}
```

Exemplos usados na aula de lista:

▼ Exemplo 1:

```
dividirMetade:: [t] -> ([t],[t])
dividirMetade lista = (take k lista, drop k lista)
  where
    k = div (length lista) 2
```

▼ Exemplo 2:

```
nossoInit:: [t] -> [t]
nossoInit lista = take (length lista) lista
```

▼ Exemplo 3:

```
pegarUltimo:: [t] -> t
pegarUltimo lista = lista !! ((length lista) -1)
```

▼ Exemplo 4:

```
bhaskara:: Float -> Float -> Float -> [Float]
bhaskara a b c
| a == 0 = []
| delta < 0 = []
| delta == 0 = [-b / (2*a)]
| otherwise = [(-b + sqrt delta)/2*a, (-b - sqrt delta)/2*a]
  where
    delta = b^2 - 4*a*c
```

▼ Exemplo 5:

```
baseDeDados :: [(Int, String, Float)]
baseDeDados = [(1, "Vinicius", 9.0), (2, "Lucas", 9.3), (3, "Sergio", 9.5), (4, "Gustavo", 9.8), (5, "Mariana", 10.0), (6, "Jane

retornaNomes :: [(Int, String, Float)] -> [String]
retornaNomes lista = [pegarNome a | a <- lista]
  where
    pegarNome (a,b,c) = b
```

▼ Exemplo 6:

```
meuDelete :: Int -> [Int] -> [Int]
meuDelete queroRemover lista = [a | a <- lista, a /= queroRemover]
```