

Paradigma Funcional Haskell



ERICK GALANI MAZIERO
erick.maziero@ufla.br

Departamento de Ciências da Computação
Universidade Federal de Lavras



Haskell - I



Haskell é uma é uma linguagem de programação puramente funcional e de propósito geral.

Foi criada em 1990, derivada de outras linguagens funcionais como Miranda e ML. Surgiu a partir de esforços para padronizar linguagens funcionais, a partir do interesse nesse tipo de linguagem.

Haskell é fortemente tipada: parâmetros podem possuir tipos de dados específicos vinculados.

Ela pode ser compilada ou interpretada.





Haskell - II



Para melhor entender os exemplos que serão dados, é importante instalar um compilador ou interpretador Haskell e reproduzir os exemplos

Compilador pode ser baixado no link abaixo para várias plataformas:

GHC (<https://www.haskell.org/ghc/>)



About GHC

[Home](#)
[License](#)
[Documentation](#)
[Blog](#)
[Download](#)
[Report a bug](#)
[Developers Wiki](#)

About Haskell

[Haskell.org](#)
[Haskell 2010 Report](#)
[Haskell Mailing Lists](#)

Links

[Haskell Platform Hackage](#)

Latest News

25 August 2019

GHC 8.8.1 Released! [[download](#)]

23 April 2019

GHC 8.6.5 Released! [[download](#)]

5 March 2019

GHC 8.6.4 Released! [[download](#)]

7 December 2018

GHC 8.6.3 Released! [[download](#)]

2 November 2018

GHC 8.6.2 Released! [[download](#)]

What is GHC?

GHC is a state-of-the-art, open source, compiler and interactive environment for the functional language [Haskell](#). Highlights:

- GHC supports the entire [Haskell 2010 language](#) plus a wide variety of [extensions](#).
- GHC has particularly good support for [concurrency](#) and [parallelism](#), including support for [Software Transactional Memory \(STM\)](#).




Haskell - II



Pode usar também, para sistemas baseados no UNIX:
Linux e Mac OS

GHCUP (<https://www.haskell.org/ghcup/>)





ghcup is an installer for
the general purpose language **Haskell**

Run the following in your terminal (as a user other than root),
then follow the onscreen instructions.

```
$ curl https://get-ghcup.haskell.org -sSf | sh
```

If you don't like `curl | sh`, see [other installation methods](#).
You appear to be running macOS. If not, [display all supported installers](#).

Need help? [Ask on #haskell](#).




Um exemplo - Fatorial em C e em Haskell

// Fatorial em C

```
int fatorial (int n) {  
    int i, res;  
    res = 1;  
    for (i = n; i > 1;  
i--)  
        res = res * i;  
    return res;  
}
```

-- Fatorial em Haskell

```
fac n = if n == 0 then 1  
       else n * fac (n-1)
```



Fatorial em uma linha, em Haskell

Como interpretar?

Usando o interpretador `ghci`, indicado pelo prompt `prelude`

```
prelude> let fac n = if n == 0 then 1 else n * fac (n-1)
prelude> fac 43
60415263063373835637355132068513997507264512000000000
```


```
prelude>:l functions.hs
prelude> fac 43
60415263063373835637355132068513997507264512000000000
```

Arquivo functions.hs
fac n = if n == 0 then 1 else n * fac (n-1)



Exemplo da função Fibonacci

```
fib 0 = 0  
fib 1 = 1  
fib n = fib (n - 2)  
  
main = do  
    print $ fib 6
```



A ordem dos comandos influencia no resultado da chamada da função fib, porque?





Exemplos Básicos em Haskell

```
idade = 19    -- define a função idade, sem argumentos, não uma variável,  
que sempre retorna a constante 19
```

```
maiorDeIdade = (idade >= 18) -- função maiorDeIdade, sem  
argumentos, que retorna um valor booleano dependendo da comparação
```

```
quadrado x = x * x -- função quadrado com um argumento (x)
```

```
result = succ 9 + max 5 4 + 1 -- função result como  
composição de outras funções
```





Exemplos Básicos em Haskell

```
Prelude> succ 5
```


```
6
```

```
Prelude> truncate 6.59
```

```
6
```

```
Prelude> round 6.59
```

```
7
```



Usando o interpretador Haskell,
exemplo de algumas funções
numéricas



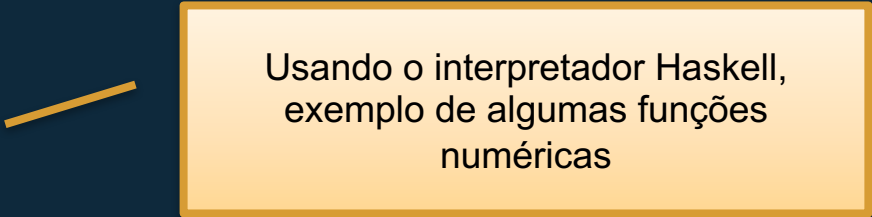


Exemplos Básicos em Haskell

```
Prelude> sqrt 2  
1.4142135623730951
```

```
Prelude> not (5 < 3)  
True
```

```
Prelude> gcd 21 14  
7
```



Usando o interpretador Haskell,
exemplo de algumas funções
numéricas



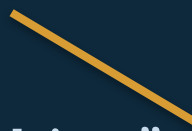


Exemplos Básicos em Haskell

```
Prelude> putStrLn "Hello, Haskell"  
Hello, Haskell
```

```
Prelude> putStr "No newline"  
No newline
```

```
Prelude> print (5 + 4)  
9
```



Exemplos de funções para exibir
conteúdo





Bloco do

```
Prelude> do { putStr "2 + 2 = " ; print  
(2 + 2) }  
2 + 2 = 4 —
```

Usa-se `do { ... }` para colocar, em uma mesma linha, vários comandos separados por `;`

```
Prelude> do { putStrLn "ABCDE" ;  
putStrLn "12345" }  
ABCDE  
12345
```





Exemplos Básicos em Haskell

```
Prelude> do { n <- readLn ;  
print (n^2) }
```

4

16




readLn lê uma entrada do usuário





Blocos em Haskell

```
main = do putStrLn "What is 2 + 2?"  
        x <- readLn  
        if x == 4  
            then putStrLn "You're right!"  
            else putStrLn "You're wrong!"
```



Exemplo simples de um programa
interativo, definido na função `main`





Tipos: inferência

```
Prelude> :t True
```


```
True :: Bool
```

```
Prelude> :t 'X'
```

```
'X' :: Char
```

```
Prelude> :t "Hello, Haskell"
```

```
"Hello, Haskell" :: [Char]
```




: t seguido de uma expressão, retorna o seu tipo





Tipos: void ou null

```
Prelude> ()  
()
```



() indica o tipo vazio, ou nulo

```
Prelude> :t ()  
() :: ()
```





Operações Infixas e Pré-fixas

resto1 = mod 10 3 —

Operação pré-fixa

resto2 = 10 `mod` 3 —

Operação infix

soma1 = 10 + 3 —

Operação infix

soma2 = (+) 10 3 —

Operação pré-fixa





Mais Exemplos

-- ou exclusivo

$x \text{ `xor` } y = (x \mid\mid y) \ \&\& \ \text{not} \ (x \ \&\& \ y)$

-- and

$x \text{ `and` } y = x \ \&\& \ y$

-- pi (constante) e ponto flutuante

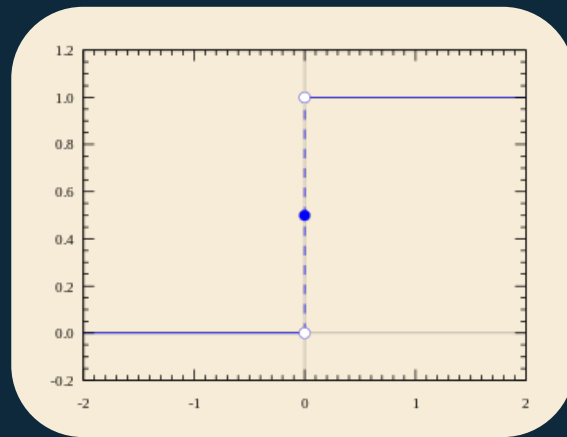
$\text{circumference } r = 2 * \text{pi} * r$



Condicionais em Haskell - Guardas - i



Para entender guardas em Haskell, é necessário antes rever algumas formas de definições de funções. Seja a função de Heavside, por exemplo:

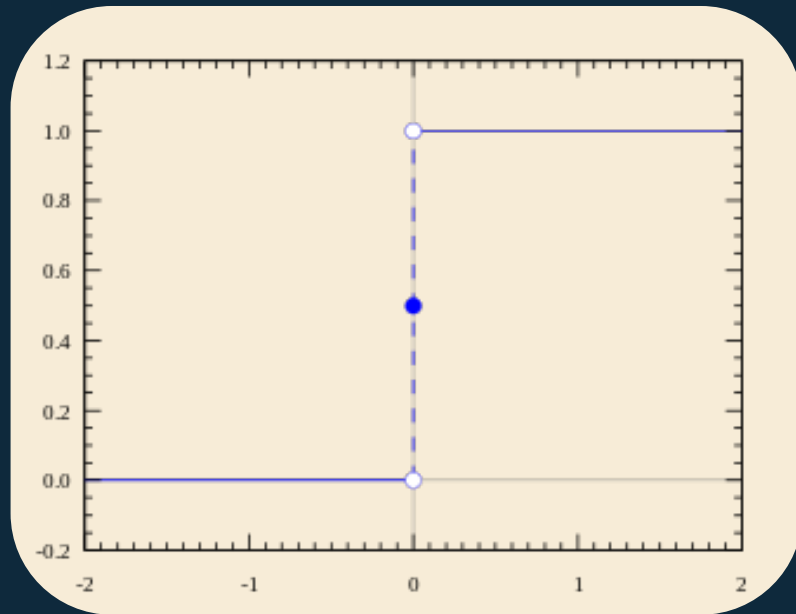


Condicionais em Haskell - Guardas - ii

A função de Heavside é dada pela seguinte expressão:

$$U(x) = \frac{1 + \text{sgn}(x)}{2} = \begin{cases} 0, & x < 0 \\ \frac{1}{2}, & x = 0 \\ 1, & x > 0 \end{cases}$$

em que $\text{sgn}(x)$ é a função sinal (-1 se x é negativo e 1 se x é positivo, 0 se x vale 0)



Como fazer isso em Haskell?

Condicionais em Haskell - Guardas - iii

Com o paradigma imperativo, a solução geralmente envolve utilizar condicionais (if ... then ... else).

Em Haskell, utilizamos guardas (|):

sgn x

```
| x < 0 = -1  
| x == 0 = 0  
| x > 0 = 1
```

A função sinal (sgn) recebe -1 quando x for menor que zero, 1 quando x for maior e 0 quando x for nulo. Portanto:

heavside x = (1 + sgn x) / 2

ou

heavside x

```
| x < 0 = 0  
| x == 0 = 1/2  
| x > 0 = 1
```

Mais exemplos de guardas

```
fat1 0 = 1
```

```
fat1 n
```

```
| n > 0 = n * fat1 (n-1)
```

```
fat2 n
```

```
| n == 0 = 1
```

```
| n > 0 = n * fat2 (n-1)
```

Duas outras possíveis
implementações da função fatorial
utilizando guardas

Outros Casos - otherwise

É possível especificar casos padrões utilizando-se a expressão otherwise:

```
fat3 n
| n == 0 = 1
| n > 0 = fat3 (n - 1) * n
| otherwise =
    error "não tem!"
```

|

Duas outra possível implementação da função fatorial com otherwise

```
mult m n
| n == 0 = 0
| n > 0 = m + mult m (n-1)
| otherwise =
    - (mult m (-n))
```

|

O que faz a função mult?



Mais exemplo de guarda e caso padrão

```
mini a b
```

```
| a <= b = a
```

```
| otherwise = b
```

O que faz a função mini?



Recursão Mútua (usando guardas)

```
par n
| n == 0 = True
| n > 0 = impar (n-1)
| otherwise =
    error "não tem"
```

```
impar n
| n == 0 = False
| n > 0 = par (n-1)
| otherwise =
    error "não tem"
```

|

O que faz as funções par e impar?

Evitando problemas usando guardas e caso padrão

Suponha que queiramos saber quantos múltiplos de 7 existem até um dado valor.

É fácil fazer isso como mostra o código ao lado. Basta ir somando 1 a cada 7 números decrementados.

```
mult7 1 = 0
```

```
mult7 2 = 0
```

```
mult7 3 = 0
```

```
mult7 4 = 0
```

```
mult7 5 = 0
```

```
mult7 6 = 0
```

```
mult7 7 = 1
```

```
mult7 n = 1 + mult7(n-7)
```

Evitando problemas usando guardas e caso padrão

Suponha que queiramos saber quantos múltiplos de 7 existem até um dado valor.

É fácil fazer isso como mostra o código ao lado. Basta ir somando 1 a cada 7 números decrementados.

```
mult7 1 = 0
mult7 2 = 0
mult7 3 = 0
mult7 4 = 0
mult7 5 = 0
mult7 6 = 0
mult7 7 = 1
mult7 n = 1 + mult7(n-7)
```

O que acontecerá se a última linha for deslocada de posição?

Ordem impacta no resultado e pode gerar erros!

```
mult7 1 = 0
mult7 3 = 0
mult7 4 = 0
mult7 5 = 0
mult7 6 = 0
mult7 7 = 1
mult7 n = 1 + mult7(n-7)
mult7 2 = 0
```

Execução:

```
* Main> mult7 4
0
* Main> mult7 10
1
* Main> mult7 9
*** Exception: stack overflow
```



Solução prática e curta para o problema

```
mult7 7 = 1
```

```
mult7 n
```

```
  | (n >= 1) && (n <= 6) = 0
```

```
  | otherwise = 1 + mult7 (n - 7)
```





Sobrecarga de Operadores

-- sobrecarga de operadores

a << b

| a < b = a

| otherwise = b



Variáveis Anônimas

f x y z

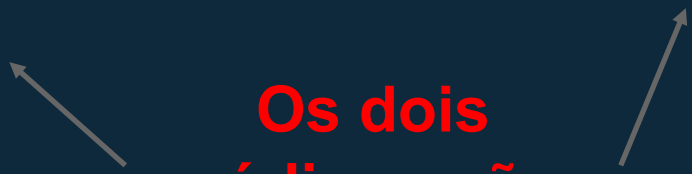
```
| (x == 7) = 10  
| (y == 8) = 20  
| (z == 9) = 30  
| otherwise = 0
```

```
f 7 _ _ = 10  
f _ 8 _ = 20  
f _ _ 9 = 30  
f _ _ _ = 0
```



O conteúdo
de uma
variável
anônima,
indicada por
_, não
interfere no
cálculo.

**Os dois
códigos são
equivalentes!**





Expressões temporárias - let

```
-- calculando a área de um cilindro
areaCilindro r h =
    let    areaLateral = 2 * pi * r * h
          areaTopo    = pi * r^2
    in    areaLateral + 2 * areaTopo
```



Expressões temporárias - where (I)

```
imcMsg :: (RealFloat a) => a -> a -> String
```

```
imcMsg peso altura
| peso / altura ^ 2 <= 19.0 = "Você esta abaixo do peso!"
| peso / altura ^ 2 <= 25.0 = "Peso normal."
| peso / altura ^ 2 <= 32.0 = "Você esta acima do peso!"
| otherwise                = "Você esta muito acima do peso!"
```

Não tem como evitar ficar repetindo
peso / altura ^ 2
??????





Expressões temporárias - where (II)

```
imcMsg :: (RealFloat a) => a -> a -> String
```

```
imcMsg peso altura
```

```
    | imc <= 19.0 = "Você esta abaixo do peso!"
```

```
    | imc <= 25.0 = "Peso normal."
```

```
    | imc <= 32.0 = "Você esta acima do peso!"
```

```
    | otherwise = "Você esta muito acima do peso!"
```

```
  where imc = peso / altura ^ 2
```





Expressões temporárias - where (III)

```
imcMsg :: (RealFloat a) => a -> a -> String
```

```
imcMsg peso altura
```

```
    | imc <= magro   = "Você esta abaixo do peso!"
```

```
    | imc <= normal = "Peso normal."
```

```
    | imc <= gordo  = "Você esta acima do peso!"
```

```
    | otherwise    = "Você esta muito acima do peso!"
```

```
  where imc = peso / altura ^ 2
```

```
        magro = 19.0
```

```
        normal = 25.0
```

```
        gordo = 32.0
```





Listas e Tuplas





Listas

Em Haskell, listas são estruturas de dados homogêneas, armazenando vários elementos do mesmo tipo.

Várias operações podem ser realizadas em uma lista, que é reconhecida como sendo composta de **cabeça** e **cauda**. Exemplo:

```
L1 = [1, 2, 3, 4] -- 1 é a cabeça da lista, [2, 3, 4] é a cauda
```

A **inserção** na cabeça é feita com o operador `:` e a **concatenação** com o operador `++`:

```
L2 = 0 : L1
```

```
L3 = L1 ++ [5, 6, 7, 8]
```



Operações em Listas

```
list = [1, 2, 3, 4, 5]
main = do
    print list
    print $ head list
    print $ tail list
    print $ last list
    print $ init list
    print $ list !! 3
    print $ elem 3 list
```

```
print $ length list
print $ null list
print $ reverse list
print $ take 2 list
print $ drop 2 list
print $ minimum list
print $ maximum list
print $ sum list
print $ product list
```

■

Teste os comandos acima

Obs: `print $ head list` é igual a `print (head list)`

Exemplos de Código com Listas

-- inversão de lista

```
inv :: [a] -> [a]
```

```
inv [] = []
```

```
inv (x:xs) = inv xs ++ [x]
```

-- comprimento de lista

```
tam :: [a] -> Int
```

```
tam [] = 0
```

```
tam ( x:xs ) = 1 + tam xs
```

-- maior elemento

```
max :: (Ord a) => [a] -> a
```

```
max [] = error "lista vazia!"
```

```
max [x] = x
```

```
max (x:xs)
```

```
    | x > maxTail = x
```

```
    | otherwise = maxTail
```

```
where maxTail = max xs
```



Strings são listas de caracteres

-- Iniciais de um nome

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l]
                              ++ ". "
    where (f:_) = firstname
          (l:_) = lastname
```



Listas podem ser comparadas

```
b1 :: Bool
```

```
b1 = [3,2,1] > [2,10,100]
```

```
b2 :: Bool
```

```
b2 = [3,4,2] > [3,4]
```

```
b3 :: Bool
```

```
b3 = [3,4,2] > [3,5]
```

Execução:

```
Prelude> b1
```

```
True
```

```
Prelude> b2
```

```
True
```

```
Prelude> b3
```

```
False
```

**Comparação
Lexicográfica!**

Uso de faixas e repetições

É possível utilizar faixas e repetições para descrever listas numéricas.

-- repetições

```
11 :: [Int]
11 = take 10 (cycle [1,2,3])
```

```
12 :: [Int]
12 = take 10 (repeat 5)
```

-- faixas

```
13 :: [Int]
13 = [2,4..20]
```

```
14 :: [Int]
14 = [3,6..20]
```

```
15 :: [Float]
15 = [0.1, 0.3 .. 1]
```

```
16 :: String
16 = ['A'..'Z']
```



Geração de listas (List Comprehension)

List Comprehension é uma maneira de se descrever uma lista inspirada na notação de conjuntos.

Por exemplo, se a lista `list1` é `[1, 2, 3]`, pode-se duplicar o valor dos elementos desta lista da seguinte maneira:

```
list2 = [ 2 * a | a <- list1 ]
```

Nesse caso, `list2` torna-se `[2, 4, 6]`. O operador `<-` é chamado de gerador.

Os geradores podem ser combinados com predicados que devolvem valores booleanos (`a->Bool`). Ex:

```
list3 = [ a | a<-list, even a]
```



Exemplos de geração de listas - I

```
11 :: [Int]
11 = [ x*2 | x <- [1..10] ]
```

```
12 :: [Int]
12 = [ x*2 | x <- [1..10],
           x*2 >= 12 ]
```

```
13 :: [Int]
13 = [ x | x <- [50..100],
           x `mod` 7 == 3 ]
```

```
14 :: [Int]
14 = [ x | x <- [10..20],
           x /= 13,
           x /= 15,
           x /= 19 ]
```

```
15 :: [Int]
15 = [ x*y | x <- [2,5,10],
             y <- [8,10,11],
             x*y > 50 ]
```



Exemplos de geração de listas - II

```
nomes :: [String]
nomes = ["farofa", "cerveja", "galinha"]

adjetivos :: [String]
adjetivos = ["frita", "gelada", "quente"]

ru :: [String]
ru = [nome ++ " " ++ adjetivo |
      adjetivo <- adjetivos, nome <- nomes]
```





Exemplos de geração de listas - III

-- comprimento da lista

```
tamanho :: [a] -> Int
```

```
tamanho xs = sum [1 | _ <- xs]
```

-- remoção de caracteres não maiúsculos

```
removeNonUppercase :: String -> String
```

```
removeNonUppercase st = [ c | c <- st,  
c `elem` ['A'..'Z']]
```





Tuplas

Tuplas são como listas, entretanto existem algumas diferenças fundamentais. Listas devem ter elementos do mesmo tipo de dados, independente da quantidade de elementos.

Tuplas, no entanto, possuem tamanho definido mas podem misturar elementos de tipos diferentes.

Tuplas são caracterizadas por parênteses com seus componentes separados por vírgulas. Ex:

```
aluno1 = ("joao", 20160504, 9.8)
aluno2 = ("maria", 20160524, 2.1)
aluno3 = ("pedro", 20160532, 5.9)
```



Exemplos de código de tuplas

-- definindo tuplas

```
tuple = (1, 2)
```

```
tuple3 = (1, 2, 3)
```

-- funções em tuplas

```
first (a, _, _) = a
```

```
second (_, b, _) = b
```

```
third (_, _, c) = c
```

```
main = do
```

```
  print tuple
```

```
  print $ fst tuple
```

```
  print $ snd tuple
```

```
  print tuple3
```

```
  print $ first tuple3
```

```
  print $ second tuple3
```

```
  print $ third tuple3
```

■

Teste os comandos acima



Exemplos de código

```
w :: (Integer, [Char])
w = ( let a = 100; b = 200; c = 300 in a*b*c,
      let foo="Hey "; bar = "there! "; fufu = 3.8 in
      foo ++ bar ++ (show fufu) )
```

```
gt1 :: [(Int, Char)]
gt1 = zip [1..] ['A'..'Z']
```

```
triret :: [(Int, Int, Int)]
triret = [ (a,b,c) | c <- [1..20], b <- [1..c], a <- [1..b],
                a^2 + b^2 == c^2]
```



Gerando lista de tuplas

```
l1 :: [Char]
```

```
l1 = ['a' .. 'e']
```

```
l2 :: [Int]
```

```
l2 = [1 .. 5]
```

```
l3 :: [(Char, Int)]
```

```
l3 = zip l1 l2
```

Execução

```
Prelude> print l3
```

```
[('a',1),('b',2),('c',3),  
 ('d',4),('e',5)]
```



Referência Bibliográfica

Sebesta, R. W. (2011). *Conceitos de Linguagens de Programação*. 9 ed. Bookman.

Capítulo 15

<http://learnyouahaskell.com/chapters>

<https://www.tutorialspoint.com/haskell/index.htm>

