



Introdução ao Haskell (cont.)

Prof^a. Rachel Reis
rachel@inf.ufpr.br



Tipos (revisão)

- Em Haskell nomes de tipos devem começar com letra maiúscula.
- Exemplo:
 - Integer, Int
 - Char, String
 - Float, Double
 - Bool
 - ...

- Algumas funções podem operar sobre vários tipos de dados.
- Exemplo: a função `head` recebe uma lista e retorna o primeiro elemento, não importa o tipo dos elementos.

```
head ['B', 'O', 'L', 'A']
```

```
> B
```

```
head ["Pedro", "Laura", "Marcos"]
```

```
> "Pedro"
```

Qual o tipo de `head`?

```
head :: [Char]    -> Char
```

```
head :: [String] -> String
```

*** `head` pode ter vários tipos ***



Variáveis de Tipo

- Quando um tipo pode ser qualquer tipo da linguagem, ele é representado por uma variável de tipo.
- No exemplo da função head, *a* representa o tipo dos elementos da lista passados como argumento

`head :: [a] -> a`

a é uma variável de tipo que pode ser substituída por qualquer tipo.

- Variáveis de tipo devem começar com letra minúscula e são geralmente denominadas *a*, *b*, *c*, etc.



Função Polimórfica

- Uma função é chamada **polimórfica** se o seu tipo contém uma ou mais variáveis de tipo.
- Exemplo 1

```
head :: [a] -> a
```

Leitura: para qualquer tipo *a*, *head* recebe uma lista de valores do tipo *a* e retorna um valor do tipo *a*



Função Polimórfica

- Exemplo 2

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

Leitura: para qualquer tipo a , *length* recebe uma lista de valores do tipo a e retorna um inteiro

- Exemplo 3

```
fst :: (a, b) -> a
```

Leitura: para quaisquer tipos a e b , *fst* recebe um par do tipo (a, b) e retorna um valor do tipo a .



Função Polimórfica

- Muitas funções definidas no módulo Prelude são polimórficas.
- Exemplos

head :: [a] -> a	-- seleciona o 1º item de uma lista
fst :: (a, b) -> a	-- seleciona o 1º item de um par
snd :: (a, b) -> b	-- seleciona o 2º item de um par
take :: Int -> [a] -> [a]	-- seleciona os 1ºs itens de uma lista



Erros de Tipo

- Toda expressão sintaticamente correta tem seu tipo calculado em tempo de compilação.
- Se não for possível determinar o tipo de uma expressão, ocorre um erro de tipo.
- A aplicação de uma função a um ou mais argumentos de tipo inadequado constitui um erro de tipo.



Erros de Tipo

■ Exemplo

```
> not 'A'
```

```
<interactive>:1:5: error:
```

- Couldn't match expected type 'Bool' with actual type 'Char'
- In the first argument of 'not', namely ''A''
In the expression: not 'A'
In an equation for 'it': it = not 'A'

```
>
```

- Explicação: a função *not* requer um valor Bool como argumento, porém, foi passado 'A' que é do tipo Char.



Checagem de Tipos

- Haskell é uma linguagem fortemente tipada, com um sistema de tipos muito avançado.
- Todos os possíveis erros de tipos são encontrados em tempo de compilação (tipagem estática).
- Vantagem: programas mais seguros e rápidos, eliminando a necessidade de verificação em tempo de execução.



Condicionais em Haskell

- Uma função em Haskell pode incluir estruturas condicionais para desviar o fluxo do programa.
- Isso pode ser feito de duas formas:
 - 1) Usando a estrutura **if-then-else** (comum na programação imperativa).
 - 2) Usando **guardas**, representado no código por uma barra vertical '|'



Exemplo 1

- Escreva uma função que receba **dois inteiros** e retorne o **maior**. Use a estrutura if-then-else.

```
-- Definir o tipo
```



Exemplo 1 - if-then-else

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

- O *if-then-else* pode ser escrito em uma única linha.
- A cláusula *else* não é opcional, omiti-la é um erro.
- O uso dos parênteses na condição é opcional.



Condicionais com guardas

- Guardas são equações condicionais que especificam cada uma das circunstâncias nas quais a definição da função pode ser aplicada.
- Pode ou não conter a palavra *otherwise* (de outra maneira) como a última condição em uma expressão condicional.
- Com guardas, a primeira expressão avaliada como verdadeira determina o valor da função.



Exemplo 1 - guardas

- Escreva uma função que receba dois inteiros e retorne o maior. Use guardas.

```
-- Definir o tipo
```

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

```
-- Usando guardas
maiorG :: Int -> Int -> Int
maiorG a b
  | a >= b = a
  | otherwise = 0
```

- Atenção para o sinal de igual.
- Atenção para indentação. Linhas de código no mesmo nível de indentação pertencem a um mesmo bloco.



Para praticar...

- Altere o exemplo abaixo para que a função retorne zero quando os valores **a** e **b** forem **iguais**.

```
-- Versão alternativa com otherwise
maiorG :: Int -> Int -> Int
maiorG a b
    | a >= b = a
    | otherwise = b
```



Regra de Layout

- Em uma **sequência de definições**, cada definição deve começar precisamente na mesma coluna.

```
a = 10  
b = 20  
c = 30
```



```
a = 10  
  b = 20  
c = 30
```



```
  a = 10  
b = 20  
    c = 30
```





Regra de Layout

- Se uma definição for escrita em **mais de uma linha**, as linhas subsequentes devem começar em uma coluna mais à direita da coluna que caracteriza a sequência de definições.

```
a = 10 + 20 +  
    30 + 40  
b = sum [10,20]
```



```
a = 10 + 20 +  
30 + 40  
b = sum [10,20]
```



```
a = 10 + 20 +  
30 + 40  
b = sum [10,20]
```





Regra de Layout

- A regra de *layout* **evita** a necessidade de uma sintaxe explícita para indicar o agrupamento de definições usando `{ }` e `;`.

{- agrupamento implícito -}

```
a = b + c
  where
    b = 1
    c = 2
d = a * 2
```

{- agrupamento explícito -}

```
a = b + c
  where { b = 1 ; c = 2 }
d = a * 2
```



Regra de Layout



- Evite o uso de caracteres de **tabulação**.



Regra de Layout

-- Usando if then else

```
maior :: Int -> Int -> Int
```

```
    maior a b = if a >= b
```

```
        then a
```

```
        else b
```



-- Usando guardas

```
maiorG :: Int -> Int -> Int
```

```
maiorG a b
```

```
| a >= b = a
```

```
| otherwise = 0
```





Praticando em laboratório...

- Crie um modulo FuncoesDecisao.hs
- Escreva uma função que informe se um dado número é **par** usando if-then-else e guardas.

```
-- Definir o tipo  
  
-- Declarar a função
```



Exercício 1 – if-then-else

- Escreva uma função que informe se um dado número é **par** usando **if-then-else** e guardas.

```
-- Definir o tipo
ehPar :: Int -> Bool
-- Declarar a função
ehPar x = if mod x 2 == 0
          then True
          else False
```




Exercício 1 - guardas

- Escreva uma função que informe se um dado número é **par** usando if-then-else e **guardas**.

```
-- Definir funcao
ehParG :: Int -> Bool
-- Declarar função
ehParG x
    | (mod x 2 == 0) = True
    | otherwise = False
```



Exercício 2

- Escreva uma função que receba **três números** e determine se eles podem **formar um triângulo**. Use if-then-else e guardas. Dica: a soma de dois lados quaisquer é sempre maior que o terceiro.

```
-- Definir função
```

Exercício 2 – if-then-else

```
-- Definir função
formarTriangulo :: Int -> Int -> Int -> Bool
-- Declarar função
formarTriangulo a b c =
    if a + b > c && a + c > b && b + c > a
    then True
    else False
```

Solução 1

- O uso do **where** permite definir “variáveis locais” no escopo da função **formarTriangulo**.

```
-- Definir função
formarTriangulo :: Int -> Int -> Int -> Bool
-- Declarar função
formarTriangulo a b c =
    if somaAB > c && somaAC > b && somaBC > a
    then True
    else False
→ where
    somaAB = a + b
    somaAC = a + c
    somaBC = b + c
```

- A variável `condicao` é atribuída ao resultado da verificação da condição utilizando `if-then-else`

```
-- Definir função
```

```
formarTriangulo :: Int -> Int -> Int -> Bool
```

```
-- Declarar função
```

```
formarTriangulo a b c = condicao
```

```
  where
```

```
    somaAB = a + b
```

```
    somaAC = a + c
```

```
    somaBC = b + c
```



```
    condicao = if somaAB > c && somaAC > b &&
               somaBC > a
               then True
               else False
```

Exercício 2 - guardas

- Escreva uma função que receba três números e determine se eles podem formar um triângulo. Use if-then-else e **guardas**.

```
-- Definir função
formarTriangulo :: Int -> Int -> Int -> Bool
-- Declarar função
formarTriangulo a b c
  | a + b > c && a + c > b && b + c > a = True
  | otherwise = False
```

Solução 1

Exercício 2 - guardas

- Escreva uma função que receba três números e determine se eles podem formar um triângulo. Use if-then-else e **guardas**.

```
-- Definir função
formarTriangulo :: Int -> Int -> Int -> Bool
-- Declarar função
formarTriangulo a b c
  | somaAB >c && somaAC >b && somaBC >a = True
  | otherwise = False
  where
    somaAB = a + b
    somaAC = a + c
    somaBC = b + c
```

Solução 2

Exercício 3

- Escreva uma função que receba três notas (p1, p2, p3) e calcule a média aritmética das provas. Se a média for maior ou igual a 7, a função deve retornar a mensagem “Aprovado”; caso contrário, “Reprovado”. Use **guardas** e **where**.

```
-- Definir função
```


Exercício 3 - guardas

- Escreva uma função que receba três notas (p1, p2, p3) e calcule a média aritmética das provas. Se a média for maior ou igual a 7, a função deve retornar a mensagem “Aprovado”; caso contrário, “Reprovado”. Use **guardas** e **where**.

```
-- Definir função
mediaP :: Float -> Float -> Float -> String
-- Declarar função
mediaP p1 p2 p3
  | media >= 7 = "Aprovado"
  | otherwise = "Reprovado"
where
  media = (p1 + p2 + p3) / 3
```



Função recursiva

- Em Haskell, como não é possível controlar o estado do programa ou de variáveis de controle, não existe estruturas de repetição.
- Toda repetição deve ser efetuada por meio de recursão.
- Uma função recursiva é formada por duas partes:
 - Caso base
 - Passo recursivo



Exemplo 1

- Escreva uma função recursiva para calcular o **fatorial** de um número.

```
-- Definir a função
```

Exemplo 1

```
-- Definir a função
fatorialG :: Int -> Int
-- Declarar a função (com guardas)
fatorialG n
    | n == 0 = 1
    | n > 0  = n * fatorialG (n-1)
```

```
-- Definir a função
fatorial :: Int -> Int
-- Declarar a função (sem guardas)
fatorial 0 = 1
fatorial n = n * fatorial (n-1)
```



Praticando em laboratório

- Crie um módulo `FuncoesRecursiva.hs`
- Escreva uma função recursiva em Haskell para calcular a potência de x^n , sendo $x > 0$ e $n \geq 0$. Implemente a função com guardas e sem guardas.

```
-- Definir a função
```

Exercício 1 - potência

```
-- Definir a função
potenciaG :: Int -> Int -> Int
-- Declarar a função (com guardas)
potenciaG x n
    | n == 0 = 1
    | n > 0  = x * potenciaG x (n-1)
```

```
-- Definir a função
potencia :: Int -> Int -> Int
-- Declarar a função (sem guardas)
potencia x 0 = 1
potencia x n = x * potencia x (n-1)
```



Exercício 2 - somatório

- Implemente uma função recursiva que calcule o somatório em um intervalo $[0, y]$, sendo y números inteiros, e $0 < y$.

```
-- Definir a função
```

Exercício 2 - somatório

```
-- Definir a função
somaG :: Int -> Int
-- Declarar a função (com guardas)
somaG n
    | n == 0 = 0
    | otherwise = n + somaG (n-1)
```

```
-- Definir a função
soma :: Int -> Int
-- Declarar a função (sem guardas)
soma 0 = 0
soma n = n + soma (n - 1)
```




Referências

- Oliveira, A. G. de (2017). Haskell: uma introdução à programação funcional. São Paulo, SP: Casa do Código.
- Sá, C. C. de, Silva, M. F. da (2006). Haskell: Uma abordagem Prática. Novatec. São Paulo, 2006.