



# Introdução ao Haskell (cont.)

---

Prof<sup>a</sup>. Rachel Reis  
rachel@inf.ufpr.br



# Plataforma Haskell

---

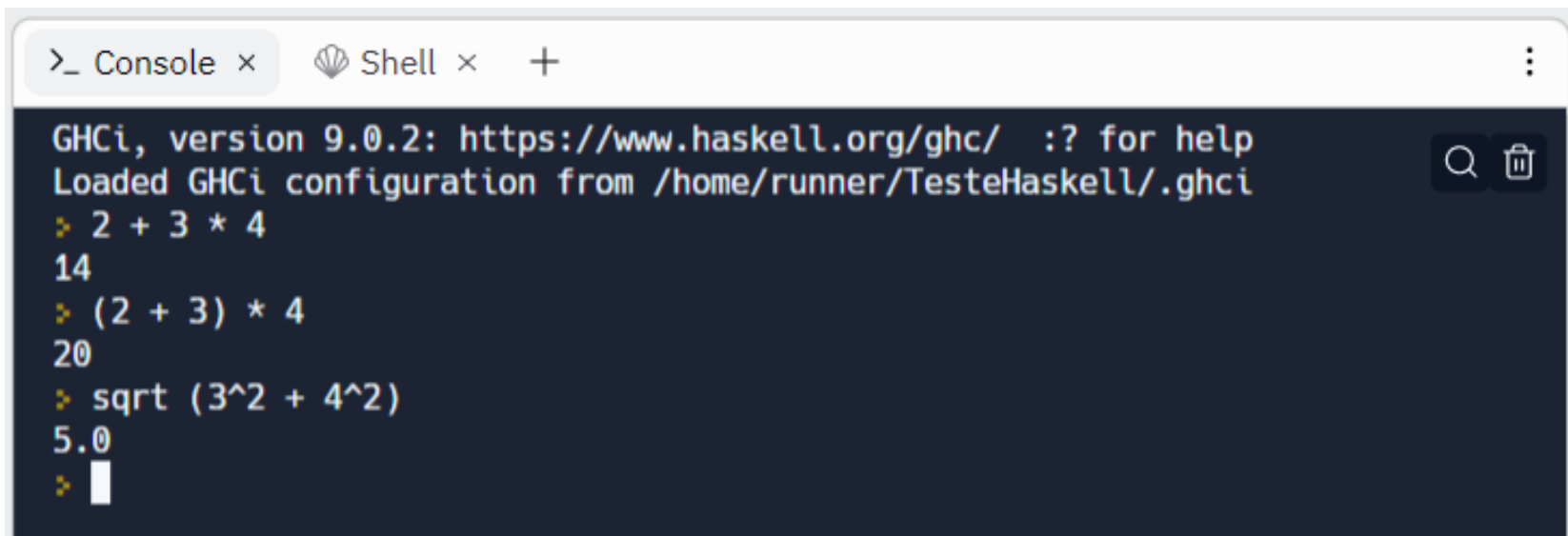
- Link de instalação: <https://www.haskell.org/downloads/>
- A Plataforma Haskell é formada:
  - Compilador GHC (*The Glasgow Haskell Compiler*)
  - Várias bibliotecas prontas para serem usadas.
- Compilador **GHC** compreende
  - Compilador de linha de comando: gera código executável.
  - Ambiente interativo GHCi: permite a avaliação de expressões de forma interativa.



# Ambiente interativo GHCi

---

- O Replit (<https://replit.com/>) possui suporte ao desenvolvimento de código em Haskell.
- Na aba console o GHCi está pronto para avaliar expressões.



```
>_ Console x Shell x +  
GHCi, version 9.0.2: https://www.haskell.org/ghc/ :? for help  
Loaded GHCi configuration from /home/runner/TesteHaskell/.ghci  
> 2 + 3 * 4  
14  
> (2 + 3) * 4  
20  
> sqrt (3^2 + 4^2)  
5.0  
> 
```



# Módulos

---

- Programas em Haskell são organizados em módulos. Um módulo é formado por um conjunto de definições (tipos, funções, etc).
- O módulo principal carrega outros módulos para fazer algo de útil.
- Exemplo

```
module Main where
main :: IO ()
main = do
    putStrLn "Hello world"
```



# Comentários

---

- Uma linha: demarcado pela sequência `--`
- Múltiplas linhas: demarcados por `{ - e - }`

```
modulo Main where -- modulo Main
```

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

```
main = do
```

```
{ -
```

```
    Instruções do módulo Main utilizando  
    várias linhas
```

```
- }
```



# Biblioteca Padrão

---

- A biblioteca padrão é formada por um conjunto de módulos disponíveis automaticamente para todos os programas em Haskell.
- A biblioteca Prelude.hs oferece um grande número de funções definidas através do módulo Prelude.
- O módulo Prelude é importado automaticamente em todos os módulos de uma aplicação Haskell.
- Todas as definições do módulo Prelude podem ser listadas no GHCi usando o comando *:browse Prelude*



# Biblioteca Padrão

---

- O módulo Prelude oferece várias funções: aritméticas, para manipulação de listas e outras estruturas de dados.
- Exemplo (funções matemáticas):

- `sqrt :: a -> a`

```
sqrt 25  
> 5
```

- `mod :: a -> a -> a`

```
mod 10 3  
> 1
```



# Biblioteca Padrão

---

- Exemplo 1 (manipulação de listas):

`length:` calcula o tamanho da lista.

```
length [1, 2, 3, 4, 5]
```

```
> 5
```

```
length []
```

```
> 0
```





# Biblioteca Padrão

---

- Exemplo 2 (manipulação de listas):

**!!: seleciona o n-ésimo elemento de uma lista.**

```
[1, 2, 3, 4, 5] !! 2
```

```
> 3
```

```
[1, 2, 3, 4, 5] !! 10
```

```
> *** Exception: Prelude.(!!): index too large
```



# Biblioteca Padrão

---

- Exemplo 3 (manipulação de listas):

**take:** seleciona os primeiros *n* elementos de uma lista.

```
take 3 [1, 2, 3, 4, 5]  
> [1, 2, 3]
```



# Biblioteca Padrão

---

- Exemplo 4 (manipulação de listas):

**drop**: remove os primeiros *n* elementos de uma lista.

```
drop 3 [1, 2, 3, 4, 5]  
> [4, 5]
```



# Aplicação de Função

	<b>Matemática</b>	<b>Haskell</b>
Aplicação de função	parênteses	espaço
Multiplicação	justaposição	operador *

## ■ Exemplo

■ Matemática

$f(a, b) + cd$

■ Haskell

`f a b + c * d`



# Aplicação de Função

---

➤ Qual das opções representa a função  $f(a + b)$  ?

a)  $f(a + b)$  ✗

b)  $(f a) + b$  ✓

- A aplicação de função tem precedência maior do que todos os outros operadores.



# Aplicação de Função

- Exemplos

Matemática	Haskell
$f(x)$	<code>f x</code>
$f(x, y)$	<code>f x y</code>
$f(g(x))$	<code>f (g x)</code>
$f(x, g(y))$	<code>f x (g y)</code>
$f(x)g(y)$	<code>f x * g y</code>



# Funções

---

- Além de usar as funções do módulo Prelude, o programador pode também **definir** e **usar** suas próprias funções.

- Formato:

`<nome><lista de parâmetros> = <expressão>`

- Exemplo

`multiplica x y = x * y`



# Funções em Scripts

---

- Para organizar melhor o código, o programador pode criar seu módulo com as funções que deseja definir.
- É recomendado que os módulos sejam salvos em *scripts*, ou seja, arquivos com a extensão “.hs” (Haskell Script).
- É recomendado que o módulo tenha o mesmo nome do *script*.





# Funções em Scripts

```
module Operacoes where  
multiplica x y = x * y  
soma x y = x + y
```

- 1) Crie um *script* com o nome `Operacoes.hs`
- 2) Crie uma função para `mutiplicar` dois números
- 3) Crie uma função para `somar` dois números



# Funções em Scripts

---

- Vamos testar o *script* isoladamente usando o ambiente interativo GHCi
  - Carregar o novo script

```
:l Operacoes.hs
```

- Executar as funções:

```
multiplica 10 20  
soma 10 20
```



# Funções em Scripts

---

- Testando as funções no ambiente interativo GHCi

```
> :l Operacoes.hs
[1 of 1] Compiling Operacoes
Ok, one module loaded.
> multiplica 10 20
200
> soma 10 20
30
> 
```



# Funções em Scripts

---

- Um módulo pode importar funções de outros módulos.

*script.* Operacoes.hs

```
module Operacoes where  
multiplica x y = x * y  
soma x y = x + y
```

*script.* Calculadora.hs

```
module Calculadora where  
import Operacoes  
opSoma x y = soma x y  
opMult x y = multiplica x y
```



# Funções em Scripts

---

- Vamos usar o ambiente interativo GHCi
  - Carregar os *scripts*

```
:l Calculadora.hs
```

- Executar as funções

```
opSoma 10 20  
opMult 10 20
```



# Funções em Scripts

---

- Testando as funções no ambiente interativo GHCi a partir de um outro modulo.

```
> :l Calculadora
[1 of 2] Compiling Operacoes      ( Operacoes.hs, interpreted )
[2 of 2] Compiling Calculadora    ( Calculadora.hs, interpreted )
Ok, two modules loaded.
> opSoma 10 20
30
> opMult 10 20
200
```



# Funções em Scripts

---

- Como chamar a operação de soma a partir do módulo Main?

```
module Main where  
main :: IO ()  
main = do
```

```
module Operacoes where  
multiplica x y = x * y  
  
soma x y = x + y
```



# Funções em Scripts

---

- Como chamar a operação de soma a partir do módulo Main?

```
module Main where
import Operacoes
main :: IO ()
main = do
    let resp = soma 20 50
    print resp
```

```
module Operacoes where
multiplica x y = x * y

soma x y = x + y
```





# Funções

---

- Convenção para nomear função: iniciar com letra minúscula. Além disso, pode conter letras, dígitos, sublinhado e apóstrofo (aspas simples).
  - Exemplos:  
soma, quadrado', maiorQue, calcula\_Area
- Convenção para nomear parâmetros de função: todas as letras em minúsculo.
  - Exemplos:  
x, num1, valor\_2



# Tipos

---

- Um tipo é uma coleção de valores relacionados.
- Em Haskell nomes de tipos devem começar com letra maiúscula.
- Exemplo:
  - Bool contém os valores lógicos True e False.



# Tipos Numéricos

---

- **Int:**

- Valores inteiros de precisão fixa.
- Limitado, representa os valores numéricos no intervalo de  $-2^{63}$  até  $2^{63} - 1$ .
- Exemplo: 750, 2023

- **Integer:**

- Valores inteiros de precisão arbitrária.
- Ilimitado, representam valores inteiros de qualquer precisão.
- Exemplo: 17, 7546789872345605678



# Tipos Numéricos

---

- **Float:**

- Valores em ponto-flutuante de precisão simples (32 bits).
- Em média, representa números com até 7 dígitos.
- Exemplo: `4.56`, `0.205`

- **Double:**

- Valores em ponto-flutuante de precisão dupla (64 bits).
- Em média, representa números com quase 16 dígitos.
- Exemplos: `78937.5`, `987.3201E-60`



# Tipos Lógico e Caractere

---

- **Bool:**

- Contém os valores lógicos: verdadeiro e false.
- Expressões booleanas podem ser executadas com os operadores && (e), || (ou) e not.
- Exemplos: **True**, **False**

- **Char:**

- Contém todos os caracteres do sistema Unicode.
- Exemplos: `'B'` , `'!'` , `'\n'`



# Tipos Lista

---

- **[t]**

- Sequência de valores do mesmo tipo
- Exemplo:    [ 'O' , 'L' , 'A' ]                      [Char]  
                         [1, 2, 3, 4]                                      [Int]

- **String**

- Sequência de caracteres delimitados por aspas duplas
- Sinônimo para [Char]
- Exemplo: "UFPR"                      [ 'U' , 'F' , 'P' , 'R' ]



# Tipos Tupla

---

- $(t_1 \dots t_2)$ 
  - Sequência de valores possivelmente de tipos diferentes.
  - Não existe tupla de um único componente.
  - Exemplo:  

<code>('O', 'I')</code>	<code>(Char, Char)</code>
<code>("Joel", 'M', 22)</code>	<code>(String, Char, Int)</code>



# Assinaturas de Tipo

---

- Qualquer expressão pode ter o seu tipo anotado.
- Se *exp* é uma expressão e *t* é um tipo, então

*exp* :: *t*

lê-se: “*exp* é do tipo *t*”

- :: tem precedência menor do que todos os operadores de Haskell.





# Assinaturas de Tipo

---

- Exemplos

<code>'a'</code>	<code>:: Char</code>
<code>"joao da silva"</code>	<code>:: String</code>
<code>45</code>	<code>:: Int</code>
<code>2 &gt; 7</code>	<code>:: Bool</code>



# Consulta de Tipo no GHCi

---

- No GHCi, o comando `:type` (ou de forma abreviada `:t`) exibe o tipo de uma expressão.

```
> :type 'A'
'A' :: Char

> :t 2 > 7
2 > 7 :: Bool

> :t not False
not False :: Bool
```



# Tipos e Funções

---

```
x :: Int  
x = 3
```

- O sinal de igual **não** representa atribuição, e sim definição
- Alguns autores consideram a definição acima como função: *“x é uma função que não recebe parâmetros e retorna um inteiro constante”*



# Tipos e Funções

---

- Ao definir uma função, o seu tipo pode ser anotado (boa prática de programação).

```
x :: Int -> Float -> Bool -> Int
```

- “x” é o nome da função
- o último tipo especificado identifica o tipo de dado a ser retornado.
- os três tipos do **meio** são os tipos dos argumentos da função



# Tipos e Funções

---

- Definição da função multiplica com seu tipo anotado.

```
module Operacoes where  
  
multiplica :: Int -> Int -> Int  
multiplica x y = x * y
```



# Tipos e Funções

---

- Definição da função soma com seu tipo anotado.

```
module Operacoes where  
  
multiplica :: Int -> Int -> Int  
multiplica x y = x * y  
  
soma :: Int -> Int -> Int  
soma x y = x + y
```



# Tipos e Funções

---

- Que alterações devem ser feitas na função `multiplica` para que ela tenha três parâmetros?

```
module Operacoes where
```

```
multiplica :: Int -> Int -> Int -> Int
```

```
multiplica x y z = x * y * z
```



# Consulta de Tipo no GHCi

---

- Se quiser verificar a assinatura de uma função no GHCi, basta digita: `:t` ou `:type` <nome da função>

```
> :type multiplica
multiplica :: Int -> Int -> Int

> :t soma
soma :: Int -> Int -> Int
```



# Para praticar...

- Crie um módulo chamado de figuras geométricas e salve em um script.
- Declare três funções para calcular a área de três figuras geométricas: quadrado ( $\text{lado} * \text{lado}$ ), retângulo ( $\text{base} * \text{altura}$ ) e triângulo ( $(\text{base} * \text{altura}) / 2$ ).
- Adicione a assinatura de tipo para as três funções.
- Teste o seu script usando o ambiente interativo GHCi:  
Carregue o novo script:  
    :l <nome do script>  
Teste as três funções  
    <nome da funcao> <argumentos>

- 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

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