



# Introdução ao Haskell

---

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



# História

---

- Surgiu por volta de 1990 com o objetivo de ser a primeira linguagem puramente funcional.
- Por muito tempo foi considerada uma linguagem acadêmica.
- Atualmente é utilizada em diversas empresas como bancos europeus, facebook, etc.



# Haskell - Características

---

- Códigos curtos e declarativos:
  - Programas em Haskell chegam a ser dezenas de vezes menores que em outras linguagens.
  - O programador declara o que o programa faz (*what to do*) e não como deve ser feito (*how to do*).
- Exemplo

```
take 100 [x | x <- nat, primo x]
```



# Haskell - Características

---

- Dados imutáveis:
  - Não existe o conceito de variável, apenas nomes e declarações.
  - Uma vez que o nome é declarado como um valor, ele não pode sofrer alterações.
- Exemplo

**x** = 1.0

**x** = 2.0

ERRO!!!



# Haskell - Características

---

- Funções recursivas:
  - Com a imutabilidade, o conceito de laços de repetição também não existe em linguagens funcionais. São implementados por meio de funções recursivas.



# Pergunta

---

- Por que em Haskell não é possível implementar algo parecido com um for?

```
int x = 1;  
for(int i = 1; i <= 10; i++)  
    x = x * 2;  
printf("%d\n", x);
```



# Iteração x Recursão

Código em C/Java

```
int x = 1;  
for(int i = 1; i <= 10; i++)  
    x = x * 2;  
printf("%d\n", x);
```

Código em Haskell

```
f 0 = 1  
f n = 2 * f (n - 1)  
print (f 10)
```



# Haskell - Características

---

- Funções de alta ordem:
  - São funções que podem receber funções como parâmetro.

```
print (aplique dobro [1, 2, 3, 4])  
> [2, 4, 6, 8]
```





# Haskell - Características

---

- Tipos polimórficos:
  - Permite definir funções genéricas que funcionam para classes de tipos.

```
fst :: (a, b) -> a  
fst (x, y) = x
```



# Haskell - Características

---

- Avaliação preguiçosa:
  - Ao aplicar uma função, o resultado só será computado, quando requisitado.
  - Evita computações desnecessárias e a criação de estruturas de dados infinitas.

```
listaInf = [1..] -- 1, 2, 3, ...  
print (take 10 listaInf)
```



# Plataforma Haskell

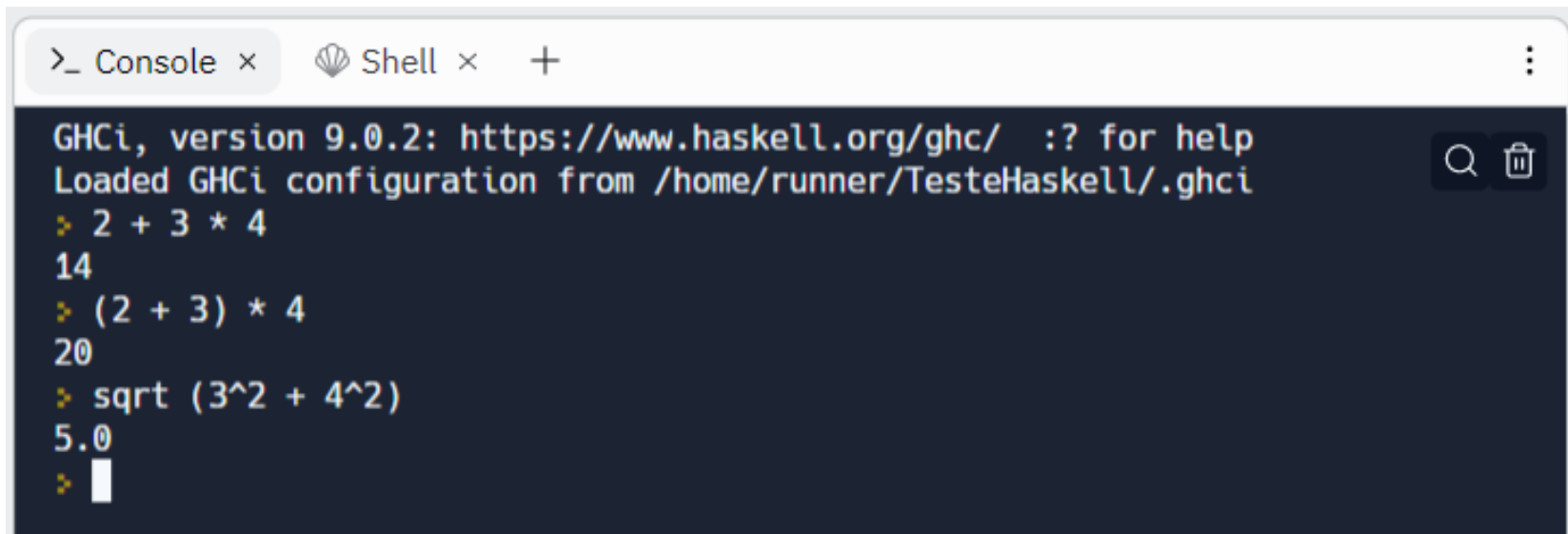
---

- 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.
- Tem-se o GHCi 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  
λ 2 + 3 * 4  
14  
λ (2 + 3) * 4  
20  
λ sqrt (3^2 + 4^2)  
5.0  
λ
```



# 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

---

$$f \ a \ b \ + \ c \ * \ d$$

Interpretando: aplica a função **f** aos argumentos **a** e **b**, e adiciona o resultado ao produto de **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

---

```
module Main where
```

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

```
soma x y = x + y
```

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

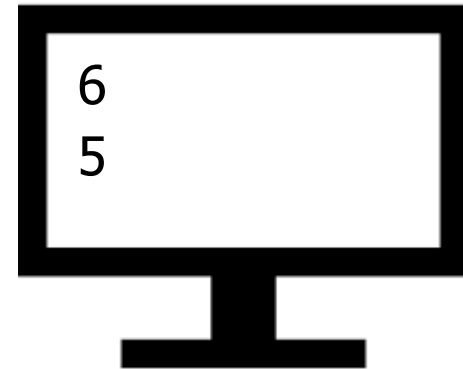
```
main = do
```

```
    let z = multiplica 2 3
```

```
    print (z)
```

```
    let s = soma 2 3
```

```
    print(s)
```





# 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

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

```
main :: IO()  
main = do
```

```
let z = multiplica 2 3  
print (z)
```

```
let s = soma 2 3  
print(s)
```



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

```
main :: IO()  
main = do
```

```
    let z = multiplica 2 3  
    print (z)
```

```
        let s = soma 2 3  
        print(s)
```





# 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

---

*script.* Operacoes.hs

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

*script.* Main.hs

```
module Main where
import Operacoes ←
main = do
    let m = multiplica 2 3
    print (m)

    let s = soma 2 3
    print(s)
```



# Funções em Scripts

---

- Podemos 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.* Teste.hs

```
module Teste where
import Operacoes ←
testeS x y = soma x y
testeM x y = multiplica x y
```



# Funções em Scripts

---

- Vamos usar o ambiente interativo GHCi
  - Carregar o dois *scripts* ou somente o Teste.hs

```
:l Operacoes.hs Teste.hs
```

 ou 

```
:l Teste.hs
```

- Executar as funções:

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



# Funções em Scripts

---

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

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



# 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