# Introdução ao Haskell

**Profa. Juliana Kaizer Vizzotto**

juvizzotto@inf.ufsm.br

Disciplina de Paradigmas de Programação

## Introdução

- Em Haskell, porgramamos definindo e avaliando expressões. Por exemplo, podemos digitar no interpretador Hugs:

```
hugs> 27+3
30
```

- Além dos operadores numéricos, podemos usar funções pré-definidas da linguagem:

```
hugs> reverse "Juliana Kaizer Vizzotto"
"ottozziV reziaK anailuJ"
```

- A função `reverse` é uma função que inverte os caracteres em uma `string`

## Introdução

- Mas a grande vantagem da programação funcional, está em permitir que os programadores definam suas próprias funções.
- Funções são definidas dentro de *scripts*

```
--   exemplo.hs
--   comentario
--
idade :: Int -- Um valor inteiro constante
idade = 17

maiorDeIdade :: Bool
maiorDeIdade = (idade >= 18)

quadrado :: Int -> Int    -- funcao que eleva num
quadrado x = x * x        -- ao quadrado
```

## Continuação

```
mini :: Int -> Int -> Int -- funcao que mostra
mini a b                  -- o menor entre
    | a <= b    = a       -- dois valores
    | otherwise = b
```

## Operadores

- Os operadores lógicos e aritméticos são pré-definidos na linguagem
- Aritméticos: $(+)$ mais, $(-)$, $(*)$ multiplicação e $(/)$ divisão
- Lógicos: (&&) AND, (||) OR, (*not*) negação
- Relacionais: $(==)$ igual, $(/=)$ diferente, $(>, <, >=, <=)$
- O tipo `Bool` possui dois valores `True` ou `False`
- Exemplos:

```
Hugs> not True
False
Hugs> True || False
True
```

## Continuação

- 
```
Hugs> True && True
True
Hugs> 3 * 5
15
```
- Os operadores podem ser usados também na forma pré-fixa, bastano colocar parênteses entre os operadores:

```
Hugs> (||) False False
False
Hugs> (+) 33 22
55
```
- Os operadores podem ser utilizados na definição de novas funções:

## Continuação

- 
```
-- script2
igual :: Int -> Int -> Bool
igual x y =  x == y

tresIguais :: Int -> Int -> Int -> Bool
tresIguais x y z = ( x == z) && (z == y)
```
- Usando funções
  - Salvar arquivo exemplo2.hs
  - Carregá-lo no interpretador: `:load exemplo2.hs`
- 
```
*Main> igual 2 4
False
*Main> tresIguais 4 4 4
True
```

## Calculando Programas

- Em Haskell, os programas são calculados internamente como na simplificação de expressões matemáticas, usando substituição: Ex:

```
tresIguais 3 3 2 =
   (3 == 3) && (3 == 2)
    True    && False
    False
```
- Você consegue calcular o resultados da seguinte expressão?

```
tresIguais (quadrado 2) idade (quadrado 3)
```

## Recursão

- Repetições que geralmente são implementadas usando loops (for, while, etc) em linguagens imperativas, são implementadas em linguagens funcionais através de recursão.
- Ex: Se possuímos uma função `vendas :: Int -> Int` que devolve como resultado as vendas semanais de uma loja, sendo que as semanas são numeradas de forma seqüencial: 0, 1, 2, 3, .... Como podemos calcular a venda total de um período de *n* semanas?
- A venda total do período da semana 0 até a semana 3 é calculada da seguinte forma:

  `vendas 0 + vendas 1 + vendas 2 + vendas 3`
- Precisamos implementar uma função que receba como argumento uma semana *n* e calcule as vendas do período de

## Recursão

- 0 até *n*

  `vendaTotal :: Int -> Int`
- O cálculo seria:

  `vendas 0 + vendas 1 + vendas 2 + ...+ vendas n`
- A função `vendaTotal` pode ser implementada da seguinte maneira em Haskell:

```
vendaTotal :: Int -> Int
vendaTotal n
   | n == 0     = vendas 0
   | otherwise  = vendas n + vendaTotal (n - 1)
```

## Trabalhando com Strings: Introdução

- `++` é um operador que concatena duas Strings

  `Main> "Alo " ++ "Mundo"`
- `show` transforma um número em uma `string`

```
Main> "Vendas na Semana 3 = " ++ show (vendas 3)
"Vendas na semana 3 = 4"
```

## Exemplo: Tabela de Vendas

- Programa que gera uma tabela com todas as vendas de várias semanas:

```
Main> putStr (tabela 4)
Semana          Vendas


Semana 0        0
Semana 1        3
Semana 2        2
Semana 3        4
Semana 4        2
Total:         11
```

## Exemplo: Tabela de Vendas

- Programa que gera uma tabela com todas as vendas de várias semanas:

```
tabela :: Int -> String
tabela n = cabecalho ++ geraVendas  n ++ total n

cabecalho :: String
cabecalho = "Semana    Vendas\n"

geraString n :: Int -> String
geraString n = "\nSemana " ++
               show n ++ "      " ++
               ++ show (vendas n)
```

## Exemplo: Tabela de Vendas

```
geraVendas :: Int -> String
geraVendas 0 = geraString 0
geraVendas n = geraVendas (n -1)
               ++ geraString n

vendas :: Int -> Int
vendas 0 = 0
vendas 1 = 3
vendas 2 = 2
vendas 3 = 4
vendas 4 = 2

total :: Int -> String
total n = "\n Total:     "
          ++ show (vendaTotal n)
          ++ "\n"
```

## Exemplo: Tabela de Vendas

```
vendaTotal :: Int -> Int
vendaTotal 0 = vendas 0
vendaTotal n = vendas n + vendaTotal (n -1)
```

## Guards e Casamento de Padrões

- Até agora, nas definições de funções usamos *guards* (guardas) para definir os possíveis casos:

```
vendaTotal :: Int -> Int
vendaTotal n
  | n == 0      = vendas 0
  | otherwise   = vendas n + vendaTotal (n - 1)
```

- Outra maneira de se tratar os diferentes casos, é usando o casamento de padrões (*pattern matching*):

```
vendaTotal :: Int -> Int
vendaTotal 0 = vendas 0
vendaTotal n =  vendas n + vendaTotal (n -1)
```

- O casamento de padrões muitas vezes faz com que o código fique muito mais fácil de se entender.

## Tuplas

- Quando programamos, modelando itens do mundo real como tipos de dados
- Até agora vimos tipos que contém apenas um valor. Ex: `Int`, `Char`, `Bool`
- Várias coisas do mundo real podem ser modeladas como coleções de itens. Ex: uma pessoa possui um nome, um telefone, uma idade

## Tuplas

- Haskell possui um tipo de composição chamado de *tupla*
- Uma tupla pode conter vários itens de tipos diferentes, o tipo:

  `(t1,t2, ..., tn)`

  é o tipo de tupla com valores:

  `(v1,v2, ..., vn)`

  sendo que $v1 :: t1, v2 :: t2 ... vn :: tn$
- Exemplo:

```
joao :: (String, String, Int)
joao = ("Joao Silva", "222-2222", 23)
```

- Para mostrar que queremos usar essa tupla para representar pessoas, podemos definir um tipo *Pessoa* usando sinônimos de tipo

## Tuplas

- 
```
type Pessoa = (String, String, Int)

joao :: Pessoa
joao = ("Joao Silva", "222-2222", 23)
```

- Podemos então definir funções que trabalham com o tipo definido:

```
nome :: Pessoa -> String
nome (n,t,i) = n

telefone :: Pessoa -> String
telefone (n,t,i) = t

idade :: Pessoa -> Int
idade (n, t,i) = i
```