# Paradigmas de Programação

Prof. Maicon R. Zatelli

Haskell - Programação Funcional
Tipos de Dados

Universidade Federal de Santa Catarina
Florianópolis - Brasil

# Haskell - Introdução

O Haskell permite introduzir novos tipos de dados...

- type
- newtype
- data

# Haskell - type

- Simplesmente é um sinônimo para um tipo já existente, que utiliza os mesmos construtores do tipo original.

```
type Nome = String
type Idade = Int
type Linguagem = String
type Pessoa = (Nome, Idade, Linguagem)
```

# Haskell - type

```
type Nome = String
type Idade = Int
type Linguagem = String
type Pessoa = (Nome, Idade, Linguagem)

pessoa :: Int -> Pessoa
pessoa 1 = ("Bob", 25, "Haskell")
pessoa 2 = ("Tom", 22, "LISP")

getNome :: Pessoa -> Nome
getNome (n, _, _) = n

main = do putStrLn ( show (pessoa 2))
          putStrLn ( show (getNome (pessoa 1)) )
          --Construindo uma pessoa
          putStrLn ( show (getNome ("Bin", 26, "Scheme")))
```

# Haskell - newtype

Será atividade ...

# Haskell - data

- Pode-se construir os próprios tipos.

```
data Forma = Circulo Float | Retangulo Float Float
```

- *Forma* é o tipo;
- Depois de "=" temos os diversos valores que este tipo pode ter;
  - Ex: uma forma pode ser algo com o valor *Retangulo* seguido por dois números float;

# Haskell - data

```
data Forma = Circulo Float | Retangulo Float Float
area :: Forma -> Float
area (Circulo r) = pi * r * r
area (Retangulo b a) = b * a

minhaForma :: Forma
minhaForma = (Retangulo 4 6)

main = do putStrLn ( show (area (Circulo 6.42)) )
          putStrLn ( show (area (Retangulo 4 5)) )
          putStrLn ( show (area minhaForma) )
```

# Haskell - data

```
data Resposta a = Sim a | Nao a
```

- *Resposta* é o tipo;
- a é uma variável de tipo, ou seja, podemos decidir que tipo de *Resposta Sim* ou *Nao* queremos (ex: *Sim Int*, ou *Sim String*);
- Depois de "=" temos os diversos valores que este tipo *Resposta* pode ter.

# Haskell - data

```
data Resposta a = Sim a | Nao a

getRespostaInt :: Resposta Int -> String
getRespostaInt (Sim x) | x >= 80 = "MUITO BOM"
                       | otherwise = "BOM"
getRespostaInt (Nao x) | x >= 80 = "MUITO RUIM"
                       | otherwise = "RUIM"

getRespostaStr :: Resposta String -> String
getRespostaStr (Sim x) = x
getRespostaStr (Nao x) = x

main = do putStrLn (show (getRespostaInt (Sim 2)))
          putStrLn (show (getRespostaInt (Sim 89)))
          putStrLn (show (getRespostaInt (Nao 58)))
          putStrLn (show (getRespostaInt (Nao 83)))
          putStrLn (show (getRespostaStr (Sim "OK")))
          putStrLn (show (getRespostaStr (Nao "Nao")))
```

* Haskell não suporta sobrecarga de funções, mas usando classes podemos resolver este problema.

# Haskell - data

## Árvore Binária

```
data Arvore = Null | No Int Arvore Arvore
```

- Uma árvore pode ser Null ou um No com valor Int e uma Arvore esquerda e outra Arvore direita.

# Haskell - data

## Árvore Binária

```
minhaArvore :: Arvore
minhaArvore = No 5 (No 3 Null Null) (No 8 (No 7 Null Null) (No 9 Null Null))
```

# Haskell - data

## Árvore Binária

```
somaElementos :: Arvore -> Int
somaElementos Null = 0
somaElementos (No n esq dir) = n + (somaElementos esq) + (somaElementos dir)
```

- Aqui fazemos a soma de todos os elementos da árvore

# Haskell - data

## Árvore Binária

```
buscaElemento :: Arvore -> Int -> Bool
buscaElemento Null _ = False
buscaElemento (No n esq dir) x
    | (n == x) = True
    | otherwise = (buscaElemento esq x) || (buscaElemento dir x)
```

- Aqui fazemos a busca por um determinado elemento na árvore, caso ele seja encontrado, retorna True, caso contrário retorna False

# Haskell - data

## Árvore Binária

```
limiteSup :: Int
limiteSup = 1000 --Define um limite superior para o maior número

minimo :: Int -> Int -> Int
minimo x y | (x < y) = x
           | otherwise = y

minimoElemento :: Arvore -> Int
minimoElemento Null = limiteSup
minimoElemento (No n esq dir) =
    minimo n (minimo (minimoElemento esq) (minimoElemento dir))
```

- Aqui procuramos o menor número armazenado na árvore. Para facilitar, criamos uma função minimo que retorna o menor dentre dois números. Assim, o menor número na árvore será o mínimo dentre o número do nó atual e o mínimo dentre os mínimos obtidos na sub-árvore esquerda e direita do nó.

# Haskell - data

## Árvore Binária

```
main = do putStrLn (show (somaElementos minhaArvore))
          putStrLn (show (buscaElemento minhaArvore 30))
          putStrLn (show (buscaElemento minhaArvore 55))
          putStrLn (show (minimoElemento minhaArvore))
```

# Haskell - Alguns Links Úteis

- `https://wiki.haskell.org/Type`
- `http://learnyouahaskell.com/making-our-own-types-and-typeclasses`

# Haskell

Ver atividade no Moodle