Universidade Federal de Pelotas

Curso de Haskell

André Rauber Du Bois
dubois@ufpel.edu.br

INTRODUÇÃO

- Em Haskell, programamos 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 "Andre Du Bois"
"sioB uD erdnA"
```

- 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
- Função são definidas dentro de *scripts*
- Segue um exemplo de script em Haskell:

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

maiorDeIdade :: Bool       -- Usa a definicao de
maiorDeIdade = (idade>=18) -- idade

quadrado :: Int -> Int   -- funcao que eleva num.
```

```
quadrado x = x * x          -- ao quadrado

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, (`-`) menos, (`*`) multiplicação e (`/`) divisão
- Lógicos: (&&) AND, (`||`) OR, (`not`) negação
- Relacionais: (==) igual, (`/=`) diferente, (`>, <, >=, <=`) multiplicação e (`/`) divisão
- O tipo `Bool` possui dois valores `True` ou `False`
- Exemplos:

```
hugs> not True
False
hugs> True || False
True
```

```
hugs> True  && True
True
hugs> 3 * 5
15
```

- Os operadores podem ser usados também na forma pré-fixa, bastando colocar parenteses entre os operadores:

```
hugs> (||) False False
False
hugs> (+) 33 22
55
```

- Os operadores podem ser usados na definição de novas funções:

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

```
tresIguais :: Int -> Int -> Int -> Bool
tresIguais x y z = (x == y) && (y == z)
```

- Usando as funções:

```
hugs> igual 2 3
False
hugs> 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 resultado da seguinte expressão??

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

CONSTRUINDO LISTAS

- Uma lista em Haskell, se parece com uma lista encadeada. Por exemplo a lista [1,2,3,4], é construida usando o operador **cons**(:), ou **construtor** de listas:

```
1 : 2 : 3 : 4 : []
```

- O operador : é uma função que serve para construir listas de qualquer tipo:

```
[True, False, False, True] ==  True : False : False : T
[1]  == 1 : []
[(1,2), (3,4)] == (1,2) : (3,4) : []
```

- Toda a vez que o **cons** é usado, ele assume o tipo da lista que está construindo:

```
(:) :: Int  -> [Int]  -> [Int]
(:) :: Bool -> [Bool] -> [Bool]
```

- Por isso, pode-se dizer que o `cons` possui um tipo **polimórfico**

  ```
  (:) :: t -> [t] -> [t]
  ```

  onde `t` é uma variável que pode assumir qualquer tipo

- Outro operador de listas visto anteriormente é o operador de concatenação (`++`)

  ```
  Hugs.Base> [1,2,3] ++ [4,5,6]
  [1,2,3,4,5,6]
  ```

FUNÇÕES QUE TRABALHAM COM LISTAS

- Supondo que queremos definir a função

  ```
  somaLista :: [Int] -> Int
  ```

  que soma os elementos de uma lista. Essa função deve trabalhar com listas de qualquer tamanho. Para implementar `somaLista` devemos usar recursão. A recursão sobre listas geralmente possui dois casos:

  - O caso da lista vazia `[]`
  - O caso da lista não vazia. Toda a lista não vazia possui um elemento **cabeça** (head), e um resto da lista chamado de **resto** (tail). Ex: A lista [1,2,3], possui cabeça 1, e rabo [2,3]. Uma lista com cabeça `c` e rabo `x`, é representada pelo padrão `(c:x)`

- A função `somaLista` pode ser definida então:

- A soma da lista vazia `[]` é `0`
- A soma de uma lista não vazia `(a:x)` é conseguida a somada a soma dos elementos de `x`. Então:

```
somaLista :: [Int] -> Int
somaLista []     = 0
somaLista (a:x)  =  a + somaLista x
```

- Exemplo:

```
Hugs> somaLista [1, 2, 3 ,4 ,5]
15
```

- A soma é calculada da seguinte maneira:

```
somaLista [1, 2, 3, 4, 5]
= 1 + somaLista [2, 3, 4, 5]
1 + ( 2 + somaLista [3, 4, 5])
= 1 + (2 + ( 3 + somaLista [4, 5]))
= 1 + (2 + ( 3 + ( 4 + somaLista [5])))
```

```
= 1 + (2 + ( 3 + ( 4 + ( 5 + somaLista [])))))
= 1 + (2 + ( 3 + ( 4 + ( 5 + 0))))
= 15
```

TIPOS ABSTRATOS

- Vimos vários tipos de dados: tipos básicos `Int`, `Float` `Char` etc... tipos compostos: `[Int]`, `(Int,Char)` `String`
- Mas existem tipos de dados comuns em computação que são de difícil modelagem usando esses tipos básicos. Ex: Meses do Ano (`Janeiro`, `Fevereiro`, etc...Árvores, etc
- Esses tipos são modelados usando os **tipos algébricos** em Haskell
- Os tipos Algébricos mais simples são modelados usando enumeração dos elementos:

```
data Temperatura = Frio | Calor
 deriving(Eq,Show)

data Estacao = Verao | Outono | Inverno | Primavera
```

```
deriving(Eq,Show)
```

- O tipo `Temperatura` possui dois valores `Frio` e `Calor` esses valores são chamados de **construtores** do tipo `Temperatura`
- Para definir funções sobre esses tipos usamos casamento de padrões:

```
tempo :: Estacao -> Temperatura
tempo Verao  = Calor
tempo _      = Frio
```

- O tipo `Bool` visto em aula também pode ser definido usando tipos algébricos:

```
data Bool = True | False
```

Tipos Produto

- Ao invés de usarmos uma tupla para definir um tipo com vários componentes, podemos usar tipos algébricos:

```
data Funcionario = Pessoa Nome Idade
 deriving(Eq,Show)

type Nome = String
type Idade = Int

andre :: Funcionario
andre = Pessoa  "Andre Du Bois" 28
```

- O construtor do tipo (`Pessoa`) pode ser visto como uma função:

```
Pessoa :: Nome -> Idade -> Funcionario
```

ALTERNATIVAS

- Uma forma Geométrica pode ser um círculo ou um Retangulo:

```
data Forma = Circulo Float |Retangulo Float Float
  deriving(Eq,Show)
```

- Podemos definir as seguintes funções:

```
redondo :: Forma -> Bool
redondo (Circulo x) = True
redondo (Retangulo x y) = False

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

TIPOS RECURSIVOS

- Tipos Algébricos podem ser Recursivos e Polimórficos.
  Exemplo:

```
-- Uma arvore binaria de inteiros:
data Arvore = Folha | Nodo Int Arvore Arvore

-- Uma arvore binaria polimorfica:
data ArvoreP a = Folha | Nodo a (Arvore a) (Arvore a)
```

- Exemplo:

```
minhaArvore :: Arvore
minhaArvore = Nodo 10 (Nodo 14 (Nodo 1 Folha Folha) Folha)
```

- Funcoes:

```
somaArvore :: Arvore -> Int
somaArvore Folha = 0
```

```
somaArvore (Nodo n a1 a2) = n +
      somaArvore a1 + somaArvore a2
```