Programação Funcional e Lógica

Introdução à Programação Funcional em Haskell

Prof. Ricardo Couto A. da Rocha
rcarocha@ufg.br
UFG – Regional de Catalão

Declarações e Variáveis

- Exemplo de declaração de variável

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

- **x** "tem tipo" **Int** e declara o seu valor como **3**.
- **x** não é uma variável no sentido tradicional de linguagens imperativas
  - Valor de **x** não pode mudar
  - **=** não tem significado de atribuição (armazenamento de valor em local da memória)
  - **=** associa **x** ao valor **3**, ou **x** é um apelido para o valor **3**.
  - Código é uma definição de **x**.

Declarações e Variáveis

- Código abaixo produz um erro

```
x :: Int
x = 3
-- linha abaixo gera erro
x = 4
```

- Código abaixo está correto (*sintaticamente*). Qual é o significado dele?

```
y :: Int
y = y + 1
```

Tipos Básicos

- Exemplos de tipos básicos em Haskell:
  - **Bool** - True ou False
  - **Char** - um caractere
  - **Int** - inteiro de tamanho fixo
  - **Integer** - um inteiro de tamanho arbitrário
  - **Float** - ponto flutuante de precisão simples
  - **Double** - ponto flutuante de dupla precisão
- Integer é limitado pela quantidade de memória disponível na máquina

```
i :: Int
i = -78
```

Tipos Básicos

```
biggestInt, smallestInt :: Int
biggestInt  = maxBound
smallestInt = minBound
-- Inteiros de precisao sem limite
n :: Integer
n = 12345678909876543219873409823349873498987453 4
```

Tipos Básicos

```
-- Ponto flutuante
d1, d2 :: Double
d1 = 4.5387
d2 = 6.2831e-4
```

```
-- Booleanos
b1, b2 :: Bool
b1 = True
b2 = False
```

Tipos Básicos

```
-- Characters
c1, c2, c3 :: Char
c1 = 'x'
c2 = 'Ø'
c3 = 'ç'
```

```
-- Strings
s :: String
s = "Hello, Haskell!"
```

Olhar cartão de referência

Estrutura de Programa

- Um código Haskell deve ser definido em um arquivo **.hs** que especifica um módulo:

```
module Programa where
-- importações
import Data.List
-- definicoes
main = 2 + 3
```

- Todas as definições dentro do módulo são limitadas ao escopo especificado pelo **where** (o módulo)
- O programa é a avaliação especificada por **main.**

Estrutura de um Programa

- Haskell diferencia maiúsculas e minúsculas
- Nomes de funções devem usar o camel case, sempre.
- A sintaxe da linguagem **exige o uso de espaços para indentação** e indentações são usualmente necessárias.
  - Os espaços jamais devem ser substituídos por tabulações

Aritmética

- Exemplos de operações aritméticas em Haskell
  - Teste a avaliação no interpretador Haskell

```
ex01 = 3 + 2
ex02 = 19 - 27
ex03 = 2.35 * 8.6
ex04 = 8.7 / 3.1
ex05 = mod 19 3
ex06 = 19 `mod` 3
ex07 = 7 ^ 222
ex08 = (-3) * (-7)
```

Aritmética e Conversões

- Expressão abaixo causa erro
  - Diferentemente de muitas linguagens, Haskell não realiza a conversão implícita dos valores em uma avaliação

```
i :: Int
i = -78
n :: Integer
n = 12345678909876543219873443 47
erroSoma = i + n
```

- Necessário usar funções explícitas para conversão dos dados, como **`fromIntegral`**, que converte um tipo integral para outro tipo numérico.
- Deve-se usar **`round`**, **`floor`** e **`ceiling`** para conversão de números ponto flutuante para inteiros.
- Divisão entre inteiros sempre deve ser feita com `` `div` ``.

```
erroDivisao = i / i
```

Operadores Infixos

- Os operadores (como +, -) nada mais são do que funções que permitem o uso com notação infixa.

Olhar cartão de referência

- A invocação infixa de operadores que usam nomes, exige o uso entre símbolos ``, como `mod`.

```
-- notacao infixa
resto1 = i `mod` 10
-- invocacao normal como funcao
resto2 = mod i 10
-- operador - invocado como função
subtracao1 = (-) i 10
```

Operadores

- Expressão matemática: **3+4/4*6-1-1**
  - Qual é o resultado dessa expressão?
- **Precedência**

Olhar cartão de referência

  - Dados que dois operadores diferentes usados em uma expressão, a predecência define qual deverá ser aplicado primeiro.
  - Multiplicação e divisão tem precedência à soma
- **Associatividade**
  - Dados dois operadores com mesma precedência, a associatividade define em qual ordem eles são avaliados → à esquerda ou à direita.
- Parênteses garantem que a avaliação será aquela que pretendemos.

Invocações de Funções

- Funções são chamadas com os parâmetros separados por espaços
  - Não há uso de parênteses para delimitar os parâmetros
  - Não há uso da vírgula como separador dos parâmetros
  - Exemplo: `fun 10 d 4`
- Funções **têm precedência mais alta** que **qualquer** operador infixo

```
duplica :: Integer -> Integer
duplica n = n * 2
duplica 15+1
```

Qual valor é retornado?

Lógica Booleana

- Valores booleanos podem ser combinados com **&&** (E lógico), **||** (OU lógico) e **not.**

```
ex11 = True && False
ex12 = not (False || True)
```

- "*Coisas*" podem ser comparadas quanto à igualdade com **==** e **/=** (diferença) e quanto à ordem com **<**, **>** **<=** e **>=**

```
ex13 = ('a' == 'a')
ex14 = (16 /= 3)
ex15 = (5 > 3) && ('p' <= 'q')
ex16 = "Haskell" > "C++"
```

Definição de Funções

- Escrevendo funções

```
-- Computa a soma dos inteiros de 1 a n
sumtorial :: Integer -> Integer
sumtorial 0 = 0
sumtorial n = n + sumtorial (n-1)
```

- Cada cláusula é verificada, da primeira à última, encontrando aquelas que casam com a avaliação a ser feita.

```
sumtorial 3 → sumtorial n
sumtorial 3 = 3 + sumtorial (3-1)
3 + sumtorial 2
3 + 2 + sumtorial (2-1)
3 + 2 + sumtorial (1)
3 + 2 + 1 + sumtorial (1-1)
3 + 2 + 1 + sumtorial (0)
3 + 2 + 1 + 0
6
```

Exercício no Laboratório

Guardas

- Haskell possui expressões `if-then-else`

```
hailstone :: Integer -> Integer
hailstone n = if n `mod` 2 == 0 then
                n `div` 2
              else 3*n + 1
```

- **Guardas** → permitem descrever condições que determinam a definição da função

```
hailstone :: Integer -> Integer
hailstone n
  | n `mod` 2 == 0 = n `div` 2
  | otherwise  = 3*n + 1
```

- Avaliação de hailstone 3
  - se `(n `mod` 2 = 0)` então `hailstone n` assume especificação `n `div` 2`.
  - É possível acrescentar quantos guardas quanto necessário, sempre iniciando com `|`, com cuidado especial com a indentação.
  - `otherwise` é definido em uma biblioteca básica (Prelude) de Haskell como `True`.
  - Código Haskell é intensivamente baseado em uso de guardas (ao invés de ifs)

Exemplo

- Para a função abaixo, quais são os valores retornados para:
- **foo (-3)**
- **foo 0**
- **foo 1**
- **foo 36**
- **foo 38**

```
foo :: Integer -> Integer
foo 0 = 16
foo 1
  | "Haskell" > "C++" = 3
  | otherwise = 4
foo n
  | n < 0 = 0
  | n `mod` 17 == 2  = -43
  | otherwise = n + 3
```

Invocações de Funções

- Declarando uma função com mais de um argumento

```
f :: Int -> Int -> Int -> Int
f x y z = x + y + z
ex17 = f 3 17 8
```

- Interpretação:
  - Função recebe um inteiro e retorna uma função que recebe um inteiro, que recebe uma função que retorna um inteiro.
- Técnica chamada de currying que sempre deve ser usada em Haskell e é derivada no Lambda Calculus.
  - Discutiremos em detalhes mais à frente.

Exercício no Laboratório

Tuplas

- Estabelecem pares ou agrupamentos maiores de dados (triplas, quádruplas, ...).

```
p :: (Int, Char)
p = (3, 'x')
```

- Exemplo de função

```
sumPair :: (Int,Int) -> Int
sumPair (x,y) = x + y
```

- Olhar no cartão de referência as funções principais de manipulação de tuplas.

Olhar cartão de referência

Listas

- Tipo elementar de estrutura de dados em Haskell e usadas intensivamente

```
nums, range, range2 :: [Integer]
nums   = [1,2,3,19]
range  = [1..100]
range2 = [2,4..100]
```

- Strings são listas de caracteres que podem ser definidas literalmente como listas ou usando aspas

```
aluno1 :: [Char]
aluno1 = ['j', 'o', 'a', 'o']
aluno2 :: String
aluno2 = "joao"
```

Construção de Listas

- Lista vazia

```
listaVazia = []
```

- Operador : constrói uma lista, onde a esquerda é o primeiro elemento e à direita o restante da lista

```
ex18 = 1 : []
ex19 = 3 : (1 : [])
ex20 = 2 : 3 : 4 : []
-- ex20 equivalente a (usando açúcar sintático)
ex21 = [2,3,4]
```

Funções em Listas

- Gerador de listas de inteiros até n.

```
gListInt :: Int -> [Int]
gListInt n = gListIntIaN 1 n
gListIntIaN i n
          | i == n = [n]
          | otherwise = i : gListInt (i+1) n
```

- Funções usualmente aproveitam casamento de padrões.
  - Exemplo: computar o tamanho de uma lista de inteiros

```
tamListInt :: [Integer] -> Integer
tamListInt [] = 0
tamListInt (x:xs) = 1 + tamListInt xs
```

  - Observe que o valor de **x** é irrelevante e por isso ele pode ser substituído por _.

Exercício no Laboratório

Funções sobre listas

- Implementar uma função para somar pares de elementos e retornar a lista resultante

```
sumEveryTwo [1,2,9,9,5] => [3,18,5]
```

- Para resolver, podemos usar padrões aninhados (padrões dentro de padrões)

```
sumEveryTwo :: [Integer] -> [Integer]
sumEveryTwo [] = []
sumEveryTwo (x:[]) = [x]
sumEveryTwo (x:(y:zs)) = (x + y) :
                         sumEveryTwo zs
```

Currying

Introduction to Funcional Programming in Haskell. Richard Bird. Capítulo 1

- Currying
  - Técnica criada pelo matemático Haskell Curry para reduzir os parênteses na especificação da função, trocando argumentos estruturados por sequências mais simples.
- Considere a especificação

```
menor :: (Integer, Integer) -> Integer
menor (x,y) = if x <= y then x else y
```

- Usando currying

```
menor :: Integer -> (Integer -> Integer)
menor x y = if x <= y then x else y
```

- **Integer -> (Integer -> Integer)** é equivalente a **Integer -> Integer -> Integer**

Currying

- Outro exemplo

```
soma :: (Integer, Integer) -> Integer
soma (x,y) = x+y
```

- Com currying

```
somac :: Integer -> (Integer -> Integer)
somac x y = x + y
```

- `somac 4 3` significa `(somac 4) 3`

Currying

- Vantagens do Currying
  - Ajuda a reduzir o número de parênteses
  - Permite aplicar a função um único argumento, gerando uma função que pode ser útil por si mesma
- **somac 4** é equivalente a

```
somacQuatro :: Integer -> Integer
somacQuatro y = 4 + y
```

Currying

- Considere a função

```
duasvezes :: (Integer -> Integer) -> (Integer -> Integer)
-- ou duasvezes :: (Integer -> Integer) -> Integer -> Integer
duasvezes f x = f (f x)
```

- Considere uma função **`quadrado x = x * x`**

```
elevaQuarta :: Integer -> Integer
elevaQuarta = duasvezes quadrado
```

- **`elevaQuarta 2`** retorna **`16`**
- Se a função fosse especificada sem usar currying não seria possível explorar essa flexibilidade. Em particular, como Haskell - e linguagens funcionais - exploram intensivamente o uso de funções como parâmetros, o uso de currying é essencial.
- É possível transformar uma função sem currying e o contrário, usando as funções **`curry`** e **`uncurry`**. Elas podem ser aplicadas em tuplas.
  - Veja o cartão de referência (coluna do meio da página 2).

Operadores e Sections

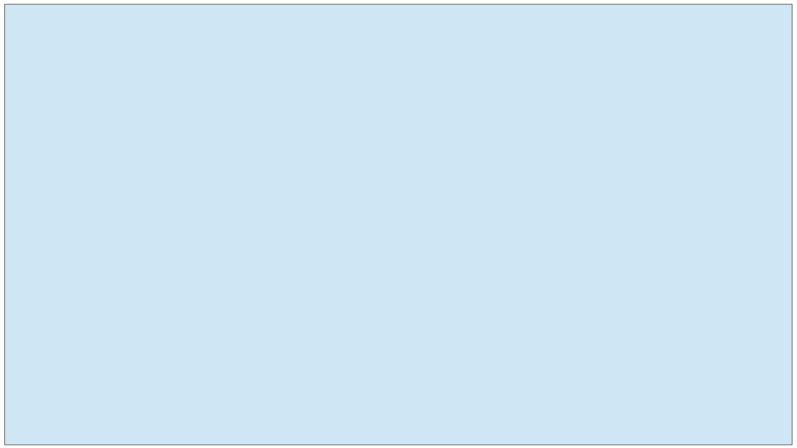
- Usar um operador no modo prefixo produz uma versão do operador que usa currying, como (+) no exemplo

```
(+) 1 2 = 1 + 2
```

- Seções: O uso com parênteses de um operador transforma-o na sua notação prefixa (que usa currying)

```
( +) 2
```

- Descreva o que faz cada uma dessas funções
  - `(*2)` : função duplica um valor
  - `(>0)` :
  - `(1/)` :
  - `(/2)` :
  - `(+1)` :



Composição funcional

- Composição funcional entre duas funções f e g é realizada pelo operador (.)

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```

ver cartão de referência ("Operadores", página 1, coluna do meio)

- Podemos refazer a definição de **elevaQuarta** usando o operador de composição

```
elevaQuarta :: Integer -> Integer
elevaQuarta = quadrado . quadrado
```