Haskell & Erlang

Aula 02

Haskell
Definindo tipos personalizados

- Uma definição de tipo começa sempre com a palavra **data,** depois vem o nome do tipo (Cor), que deve começar com letra maiúscula (note que todos os tipos em Haskell começam com letra maiúscula), e seus construtores (que também começam com letra maiúscula).
- O tipo Cor é um enumerated type. Ele é formado por quatro valores que são chamados de construtores (constructors) de tipo.
- **Exemplo**
  - data Bool = True | False
  - data Cor = Azul | Verde | Vermelho | Amarelo
- Declaração de um tipo Point que contém uma tupla e é um tipo polimórfico.
  - data Point = Pt a a

Haskell
Definindo tipos personalizados

- Pode-se criar tipos com vários componentes por exemplo:
- Figuras geométricas
  - data Forma = Circulo Float | Retangulo Float Float

- Agora vamos calcular a área, através de *pattern matching (correspondência de padrões)*:
  - area :: Forma -> Float
  - area (Circulo r) = pi * r * r
  - area (Retangulo b a) = b * a

Haskell
Definindo tipos personalizados

- Quando um tipo é definido, algumas classes podem se instanciadas diretamente através da palavra reservada deriving:

```
data Meses = Jan | Fev | Mar | Abr | Mai | Jun | Jul | Ago | Set | Out | Nov | Dez
    deriving (Eq, Show, Enum)
```

- Desta maneira, pode-se fazer coisas do tipo:

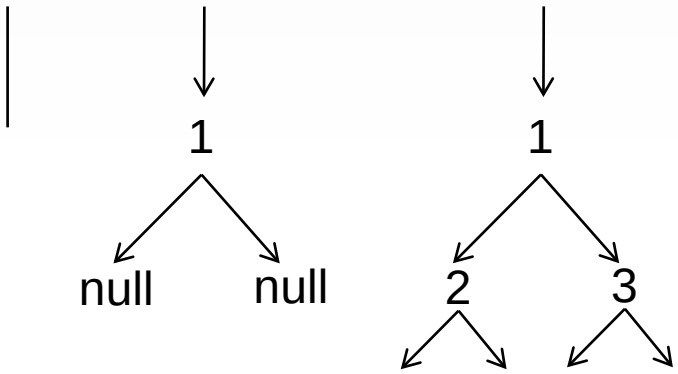
```
Haskell > Jan
Jan
Haskell > Mar == Mar
True
Haskell > [Jan .. Set]
[Jan,Fev,Mar,Abr,Mai,Jun,Jul,Ago,Set]
```

Haskell
Tipos Recursivos

- Definição de tipos algébricos recursivos
- Exemplo:
  - Construção de Árvores

```
data Arvore = Null | Node Int Arvore Arvore
```

  - Nesta definição, a árvore é definida ou por um valor nulo ou por um Node.
    - O Node é composto por um valor inteiro, uma árvore a esquerda e outra árvore a direita.



Haskell
Funções recursivas sobre árvores

- somaArvore :: Arvore -> Int
- somaArvore Null = 0
- somaArvore (Node valor esq dir) = valor + somaArvore (esq) + somaArvore (dir)

- Exercicios:
  - Função tamanho da árvore
  - Função ocorrências de um inteiro em uma árvore de inteiros
  - Contar a quantidade de folhas de uma árvore binária

Haskell
Funções de alta ordem

- Funções como argumentos ou como resultado de outras funções

- Permite
  - definições polimórficas
  - funções aplicadas sobre uma coleção de tipos
  - padrões de recursão usados por várias funções

- Facilita entendimento das funções
- Facilita modificações (mudança na função de transformação)
- Aumenta reuso de definições/código

Haskell
Função foldr1

- Folding é uma maneira de ver um loop com acumulador sobre uma lista, onde folding significa inserir um operador infixado entre os elementos da lista.
  - Considere a lista list *l = [x1 x2 x3 ... xn].*
  - *Aplicando o operador f fica:*

  x1 f x2 f x3 f ... f xn
- *O folding no Haskell é avaliado à direita (foldr1):*
  - *(x1 f (x2 f (x3 f ... (xn−1 f xn)...)))*
- foldr1:
  - Adiciona um operador entre os elementos de uma lista:
    - Main> foldr1 (+) [1..10]

      55

Haskell
Exercícios

- Defina a função **concat**, usando fold.
- Defina a função **and**, usando fold.
- Implemente uma função que inverte uma lista usando fold.

Haskell
Função filter

- filter:
  - Filtra uma lista através de uma propriedade ou predicado:
    - Exemplo:

```
Main> filter (>5) [1..10]
    [6,7,8,9,10]
Main> filter (>5) [1,2,3,4,5,6,7,8]
[6,7,8]
Main> filter odd [3,6,7,9,12,14]
[3,7,9]
Main> filter (\x -> length x > 4) ["aaaa","bbbbbbbbbbbb","cc"]
["bbbbbbbbbbbb"]
```

Haskell
Exercícios

- Use filter e a função par para definir uma função que retorne os número pares de uma lista.
- par::Int->Bool
- par n = (mod n 2 == 0)

Haskell
Função map

- map
  - Aplica uma função a cada elemento da lista:
    - Exemplo:
      - duplica :: Int -> Int
      - duplica x = 2 * x

```
Main> map duplica [1..10]
[2,4,6,8,10,12,14,16,18,20]
Main> map abs [-1,-3,4,-12]
[1,3,4,12]
Main> map reverse ["abc","cda","1234"]
["cba","adc","4321"]
Main> map (3*) [1,2,3,4]
[3,6,9,12]
```

Haskell
Exercícios

- Defina uma função que calcule o tamanho total dos elementos de uma lista de listas (usando map e fold)

- Para casa!

Haskell
Função zip

- Outra função muito utilizada é a função zip, que transforma duas listas em uma lista
- de tuplas.
- Haskell > zip [1, 3, 5] [2, 4, 6]
- [(1,2), (3, 4), (5, 6)]
- Haskell > zip [1, 3, 5, 7, 9, 11] [2, 4, 6]
- [(1,2), (3, 4), (5, 6)]

- A lista gerada sempre será limitada pela lista de menor tamanho utilizada com o argumento

Haskell
Lazy evaluation

- **Avaliação preguiçosa ou também chamada de avaliação atrasada. A ideia é que a estrutura de dados não seja feita de uma vez, e sim sob demanda.**
- Na avaliação preguiçosa, uma lista por exemplo, é construída incrementalmente.
- O consumidor da lista pede por um novo elemento quando é necessário.
- Por exemplo:

```
somaOsDoisPrimeiros :: [Integer] -> Integer

somaOsDoisPrimeiros (a:b:x) = a+b

Main> [1,2..]

[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21, {Interrupted!}

Main>somaOsDoisPrimeiros [1,2..]

3
```

Haskell
Currying

- Currying é uma técnica de simplificar programas que utilizam programação de alta ordem, a ideia é escrever funções de n argumentos em n funções aninhadas.
- *Por exemplo:*

```
soma :: Int -> Int -> Int
soma x y = x + y
```

- Esta função pega dois números inteiros como argumento e os soma:

```
Haskell > soma 2 8
10
```

- Se aplicarmos a função soma a apenas um argumento (soma 1), teremos uma função que aplicada a um argumento b, incrementa este valor (1+b).
- Pode-se definir então uma função incrementa da seguinte maneira:

```
incrementa :: Int -> Int
incrementa = soma 1
```

- Ex:

```
Haskell > incrementa 4
5
```

Haskell
Funções Lambda

- Ao invés de usar equações para definir funções, pode-se utilizar uma notação lambda, em que a função não precisa ter um nome. Por exemplo a função:

  sucessor :: Int -> Int

  sucessor x = x+1

- Na notação lambda em Haskell temos:

  Main > (\x -> x + 1) 10

  11

- Com 2 argumentos, a soma de inteiros ficaria assim:

  Main> (\ x y -> x + y) 10 20

Haskell
Definições locais

- Uma definição associa um nome a uma expressão.
- Todas as definições feitas até aqui podem ser vistas como globais, uma vez que elas são visíveis no módulo do programa aonde estão. Mas, muitas vezes é útil reduzir o âmbito de uma declaração.
- Em Haskell há duas formas de fazer definições locais: utilizando expressões **let ... In** ou através de cláusulas **where** junto da definição equacional de funções.

```
a = b + c
    where
          b = 1
          c = 2
d = a * 2
```

Referências

- [1] **http://www.macs.hw.ac.uk/~dubois/ProgramacaoHaskell.pdf**
- [2] http://www.marcosrodrigues6.hpg.ig.com.br/cap1.htm
- [3] http://www.cin.ufpe.br/~alms/pdf/JogosEducativosLinguagensFuncionais.pdf
- [4] http://caioariede.com/2009/aprendendo-erlang-parte-1
- [5] http://www.haskell.org/haskellwiki/Introduction
- [6] http://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_funcional#
- [7] http://www.erlang.se/publications/bjarnelic.pdf
- [8] http://www.haskell.org/tutorial/intro.html
- [9] Concepts, Techniques, and Models of Computar Programming, Peter Van Ray and Seif Haridi. 2003.