Introdução a Programação Funcional com Haskell

Fabrício Olivetti de França, Emílio Francesquini

29 de Setembro de 2018

Compreensão de Listas

Definindo conjuntos na matemática

Na matemática, quando falamos em conjuntos, definimos da seguinte forma:

$$\{x^2 \mid x \in \{1..5\}\}$$

que é lido como *x ao quadrado para todo x do conjunto de um a cinco*.

Compreensão de Listas

No Haskell podemos utilizar uma sintaxe parecida:

```
> [x^2 | x <- [1..5]]
[1,4,9,16,25]
```

que é lido como *x ao quadrado tal que x vem da lista de valores de um a cinco*.

A expressão `x <- [1..5]` é chamada de **expressão geradora**, pois ela gera valores na sequência conforme eles forem requisitados.

Outros exemplos:

```
> [toLower c | c <- "OLA MUNDO"]
"ola mundo"
> [(x, even x) | x <- [1,2,3]]
[(1, False), (2, True), (3, False)]
```

Compreensão de Listas

Podemos combinar mais do que um gerador e, nesse caso, geramos uma lista da combinação dos valores deles:

```
>[(x,y) | x <- [1..4], y <- [4..5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5),(4,4),(4,5)]
```

Compreensão de Listas

Se invertermos a ordem dos geradores, geramos a mesma lista mas em ordem diferente:

```
> [(x,y) | y <- [4..5], x <- [1..4]]
[(1,4),(2,4),(3,4),(4,4),(1,5),(2,5),(3,5),(4,5)]
```

Isso é equivalente a um laço `for` encadeado!

Compreensão de Listas

Um gerador pode depender do valor gerado pelo gerador anterior:

```
> [(i,j) | i <- [1..5], j <- [i+1..5]]
[(1,2),(1,3),(1,4),(1,5),(2,3),(2,4),(2,5),
     (3,4),(3,5),(4,5)]
```

Equivalente a:

```
for (i=1; i<=5; i++) {
  for (j=i+1; j<=5; j++) {
    // faça algo
  }
}
```

Exemplo: concat

A função `concat` transforma uma lista de listas em uma lista única concatenada (conhecido em outras linguagens como `flatten`):

```
> concat [[1,2],[3,4]]
[1,2,3,4]
```

Exemplo: concat

Ela pode ser definida utilizando compreensão de listas:

```
meuConcat xss = [x | xs <- xss, x <- xs]
```

Exercício: length

Defina a função `meuLength` utilizando compreensão de listas! Dica, você pode somar uma lista de 1s do mesmo tamanho da sua lista.

Guards

Nas compreensões de lista podemos utilizar o conceito de **guardas** para filtrar o conteúdo dos geradores condicionalmente:

```
> [x | x <- [1..10], even x]
[2,4,6,8,10]
```

Divisores

Vamos criar uma função chamada `divisores` que retorna uma lista de todos os divisores de n.

```haskell
divisores :: Int -> [Int]
divisores n = [x | x <- [1..n], n `mod` x == 0]
```

Divisores

```
> divisores 15
[1,3,5,15]
```

Primos

Utilizando a função `divisores` podemos definir a função `primo` que retorna `True` se um certo número é primo:

```
primo :: Int -> Bool
primo n = divisores n == [1,n]
```

Primo

Note que para determinar se um número não é primo a função `primo` **não** vai gerar **todos** os divisores de `n`.

Por ser uma avaliação preguiçosa ela irá parar na primeira comparação que resultar em `False`:

```
primo 10 => 1 : _ == 1 : 10 : [] (1 == 1)
         => 1 : 2 : _ == 1 : 10 : [] (2 /= 10)
         False
```

Primos

Com a função `primo` podemos gerar a lista dos primos dentro de uma faixa de valores:

```haskell
primos :: Int -> [Int]
primos n = [x | x <- [1..n], primo x]
```

Primos

Podemos gerar também a lista com **TODOS** os números primos:

```haskell
todosPrimos :: [Int]
todosPrimos = [x | x <- [1..], primo x]
```

Exercício

Melhore o desempenho do código sabendo que todos os números primos (exceto 2 e 3) são da forma $6k+1$ ou $6k-1$.

A função `zip`

A função `zip` junta duas listas retornando uma lista de pares:

```
> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]

> zip [1,2,3] ['a', 'b', 'c']
[(1,'a'),(2,'b'),(3,'c')]

> zip [1,2,3] ['a', 'b', 'c', 'd']
[(1,'a'),(2,'b'),(3,'c')]
```

A função `zipWith`

A função `zipWith` junta duas listas. A maneira pela qual a combinação dos elementos é efetuada é dada pela função recebida como parâmetro.

```haskell
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

A função `zip` vista anteriormente poderia ser reescrita como:

```haskell
meuZip xs ys = zipWith (\x y -> (x, y)) xs ys
```

Ou mais simplesmente:

```haskell
meuZip = zipWith (\x y -> (x, y))
```

Exemplo: A sequência de Fibonacci

Para obter-se o n-ésimo elemento da sequência de Fibonacci (1, 1, 2, 3, 5, ...) podemos utilizar a seguinte fórmula $F_n = F_{n-2} + F_{n-1}$, com $n > 1$, $F_0 = 1$, $F_1 = 2$.

Podemos enxergar a sequência assim:

```
  1  1  2  3  5  8 ...
+    1  1  2  3  5 ...
  ----------------
  1  2  3  5  8 13 ...
```

Como utilizar essa característica para fazer uma implementação funcional elegante?

Exemplo: A sequência de Fibonacci

```
  1  1  2  3  5  8 ...
+    1  1  2  3  5 ...
  ----------------
  1  2  3  5  8 13 ...
```

A segunda parcela pode ser pensada como sendo própria lista com um 0 na frente.

1) Suponha que a lista já existe.
2) Utilize-a para definir a própria lista. (Definição recursiva)
3) Combine os elementos utilizando a soma.

```haskell
fibs :: [Integer]
fibs = 1:(zipWith (+) fibs (0:fibs))
```

Só funciona pois Haskell é preguiçoso!

Recursão

Recursão

A recursividade permite expressar ideias declarativas.

```haskell
fatorial :: Integer -> Integer
fatorial 0 = 1
fatorial 1 = 1
fatorial n = n * fatorial (n-1)
```

Fatorial

O Haskell avalia as expressões por substituição:

```
> fatorial 4
      => 4 * fatorial 3
      => 4 * (3 * fatorial 2)
      => 4 * (3 * (2 * fatorial 1))
      => 4 * (3 * (2 * 1))
      => 4 * (3 * 2)
      => 4 * 6
      => 24
```

Fatorial

Ao contrário de outras linguagens, ela não armazena o estado da chamada recursiva em uma pilha, o que evita o estouro da pilha.

A pilha recursiva do Haskell é a expressão armazenada, ele mantém uma pilha de expressão com a expressão atual. Essa pilha aumenta conforme a expressão expande, e diminui conforme uma operação é avaliada.

Fatorial Caudal

Mesmo no Haskell é importante utilizar sempre que possível a recursão caudal:

```haskell
fatorial :: Integer -> Integer
fatorial 0 = 1
fatorial 1 = 1
fatorial n = fatorial' n 1
  where fatorial' 0 r = r
        fatorial' n r = fatorial' (n-1) (r*n)
```

Fatorial Caudal

```
> fatorial 4
      => fatorial' 4 1
      => fatorial' 3 (1*4)
      => fatorial' 2 (1*4*3)
      => fatorial' 1 (1*4*3*2)
      => fatorial' 0 (1*4*3*2*1)
      => (1*4*3*2*1)
      => 24
```

Recusão Múltipla

Em alguns casos o retorno da função recursiva é a chamada dela mesma em múltiplas instâncias:

```haskell
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Recursão em Listas

Quais padrões podemos capturar em uma lista?

Pattern Matching

Quais padrões podemos capturar em uma lista?

- Lista vazia: `[]`
- Lista com um elemento: `(x : [])`
- Lista com um elemento seguido de vários outros: `(x : xs)`

E qualquer um deles pode ser substituído pelo *não importa* `_`.

Funções recursivas em listas

Podemos também fazer chamadas recursivas em listas, de tal forma a trabalhar com apenas parte dos elementos em cada chamada:

```
sum :: Num a => [a] -> a
sum [] = 0
sum ns = ???
```

Funções recursivas em listas

Podemos também fazer chamadas recursivas em listas, de tal forma a trabalhar com apenas parte dos elementos em cada chamada:

```haskell
sum :: Num a => [a] -> a
sum []     = 0
sum (n:ns) = n + sum ns
```

Produtória

Como ficaria a função `product` baseado na função `sum`:

```haskell
product :: Num a => [a] -> a
product []     = 0
product (n:ns) = n + sum ns
```

Produtória

Como ficaria a função `product` baseado na função `sum`:

```
product :: Num a => [a] -> a
product []     = 1
product (n:ns) = n * product ns
```

Tamanho

E a função `length`?

```
length :: Num a => [a] -> a
length []     = 0
length (n:ns) = n + sum ns
```

Tamanho

E a função `length`?

```
length :: [a] -> Int
length []       = 0
length (n:ns) = 1 + length ns
```

Exercício

Complete a função `qsort` que implementa o algoritmo Quicksort:

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort menores ++ [x] ++ qsort maiores
  where
    menores = [a | ???]
    maiores = [b | ???]
```

Funções de alta ordem

Funções de alta ordem

As funções que recebem uma ou mais funções como argumento, ou que retornam uma função são denominadas **Funções de alta ordem** (*high order functions*).

O uso de funções de alta ordem permitem aumentar a expressividade do Haskell quando confrontamos padrões recorrentes.

Considere o padrão de código:

```
[f x | x <- xs]
```

que utilizamos para gerar uma lista de números ao quadrado, somar um aos elementos de uma lista, etc.

Map

Podemos definir a função `map` como:

```haskell
map :: (a -> b) -> [a] -> [b]
map f xs = [f x | x <- xs]
```

Uma função que transforma uma lista do tipo a para o tipo b utilizando uma função `f :: a -> b`.

Map

Com isso temos uma visão mais clara das transformações feitas em listas:

```
> map (+1) [1,2,3]
[2,3,4]

> map even [1,2,3]
[False, True, False]

> map reverse ["ola", "mundo"]
["alo", "odnum"]
```

Observações sobre o map

- Ela é um tipo genérico, recebe qualquer tipo de lista
- Ela pode ser aplicada a ela mesma, ou seja, aplicável em listas de listas:

```
> map (map (+1)) [[1,2],[3,4]]
   => [ map (+1) xs | xs <- [[1,2],[3,4]] ]
   => [ [x+1 | x <- xs] | xs <- [[1,2],[3,4]] ]
```

Filter

Outro padrão recorrente observado é a filtragem de elementos utilizando guards nas listas:

```
> [x | x <- [1..10], even x]
[2,4,6,8,10]

> [x | x <- [1..10], primo x]
[2,3,5,7]
```

Filter

Podemos definir a função de alta ordem `filter` da seguinte forma:

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [x | x <- xs, p x]
```

`filter` retorna uma lista de todos os valores cujo o predicado `p` de `x` retorna `True`.

Filter

Reescrevendo os exemplos anteriores:

```
> filter even [1..10]
[2,4,6,8,10]

> filter (>5) [1..10]
[6,7,8,9,10]
```

Map e Filter

Podemos usar as funções `map` e `filter` na sequência:

```
somaQuadPares :: [Int] -> Int
somaQuadPares ns = sum [n^2 | n <- ns, even n]

somaQuadPares :: [Int] -> Int
somaQuadPares ns = sum (map (^2) (filter even ns))
```

Operador pipe

Para aumentar a legibilidade utilizamos o operador $ para separar as aplicações das funções e remover os parênteses:

```haskell
somaQuadPares :: [Int] -> Int
somaQuadPares ns = sum
                 $ map (^2)
                 $ filter even ns
```

A execução é de baixo para cima.

Função foldr

Considerem as funções recursivas:

```haskell
sum []     = 0
sum (x:xs) = x + sum xs

product []     = 1
product (x:xs) = x * product xs

length []     = 0
length (_:xs) = 1 + length xs
```

Função `foldr`

Podemos generalizar essas funções da seguinte forma:

```haskell
f [] = v
f (x:xs) = g x (f xs)
```

Função `foldr`

Essa funções é chamada de `foldr`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Função `foldr`

Pense nessa lista não-recursivamente a partir da definição de listas:

```
a1 : (a2 : (a3 : []))
```

Função `foldr`

Trocando `:` pela função `f` e `[]` pelo valor `v`:

```
a1 `f` (a2 `f` (a3 `f` v))
```

Função `foldr`

Ou seja:

```
foldr (+) 0 [1,2,3]
```

se torna:

```
1 + (2 + (3 + 0))
```

Função `foldr`

Que é nossa função `sum`:

```
sum xs = foldr (+) 0 xs
```

Exercício

Defina `product` utilizando `foldr`.

Função `foldl`

Um outro padrão de dobra é dado pela função `foldl`:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f v []     = v
foldl f v (x:xs) = foldl f (f v x) xs
```

Função `foldl`

Da mesma forma podemos pensar em `foldl` não recursivamente invertendo a lista:

```
1 : (2 : (3 : []))
  => (([] : 1) : 2) : 3
  => ((0 + 1) + 2) + 3
```

Função `foldl`

Quando `f` é associativo, ou seja, os parênteses não fazem diferença, a aplicação de `foldr` e `foldl` não se altera:

```haskell
sum = foldl (+) 0

product = foldl (*) 1
```

`foldr` vs `foldl`

Uma regra do *dedão* para trabalharmos por enquanto é:

- Se a lista passada como argumento é infinita, use `foldr`
- Se o operador utilizado pode gerar curto-circuito, use `foldr`
- Se a lista é finita e o operador não irá gerar curto-circuito, use `foldl`
- Se faz sentido trabalhar com a lista invertida, use `foldl`

(na verdade, ao invés de `foldl` devemos utilizar `foldl'` que é a versão não preguiçosa).

Exercício

Dadas as funções `dobra` e `somaUm` aplique-as em sequência na lista `[1..10]` utilizando `map`:

```haskell
dobra :: Num a => a -> a
dobra x = 2*x

somaUm :: Num a => a -> a
somaUm x = x + 1
```

Composição de funções

Podemos criar a composição de funções utilizando o operador `(.)`:

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

Composição de funções

Com isso evitamos sequências de `map` ou `filter`:

```
map (somaUm . dobra) [1..10]
```

Definindo novos tipos

Novos tipos de dados

A definição de novos tipos de dados, além dos tipos primitivos, permite manter a legibilidade do código e facilita a organização de seu programa.

Declaração de tipo

A forma mais simples de definir um novo tipo é criando *apelidos* para tipos existentes:

```haskell
type String = [Char]
```

(equivalente ao `#typedef` do C)

Declaração de tipo

Todo nome de tipo deve começar com uma letra maiúscula. As definições de tipo podem ser encadeadas!

Suponha a definição de um tipo que armazena uma coordenada e queremos definir um tipo de função que transforma uma coordenada em outra:

```haskell
type Coord = (Int, Int)
type Trans = Coord -> Coord
```

Declaração de tipo

A declaração de tipos pode conter variáveis de tipo:

```haskell
type Pair a = (a, a)

type Assoc k v = [(k,v)]
```

Declaração de tipo

Com isso podemos definir funções utilizando esses tipos:

```haskell
find :: Eq k => k -> Assoc k v -> v
find k dict = head [v | (k',v) <- dict, k == k']
```

```
> find 2 [(1,3), (5,4), (2,3), (1,1)]
3
```

Tipos de Dados Algébricos

Tipos de Dados Algébricos

- Tipos completamente novos.
- Pode conter tipos primitivos.
- Permite expressividade.
- Permite checagem em tempo de compilação

Tipos Soma

Tipo soma:

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

- data: declara que é um novo tipo
- Bool: nome do tipo
- True | False: poder assumir ou True ou False

Exemplo

Vamos criar um tipo que define a direção que quero andar:

```haskell
data Dir = Norte | Sul | Leste | Oeste
```

Exemplo

Com isso podemos criar a função `para`:

```
data Dir = Norte | Sul | Leste | Oeste

para :: Dir -> Coord -> Coord
para Norte (x,y) = (x,y+1)
para Sul   (x,y) = (x,y-1)
para Leste (x,y) = (x+1,y)
para Oeste (x,y) = (x-1,y)
```

Tipos Produto

Tipo produto:

```haskell
data Ponto = Ponto Double Double
```

- data: declara que é um novo tipo
- Ponto: nome do tipo
- Ponto: construtor (ou envelope)
- Double Double: tipos que ele encapsula

Tipos Produto

Para ser possível imprimir esse tipo:

```haskell
data Ponto = Ponto Double Double
                     deriving (Show)
```

- deriving: derivado de outra classe
- Show: tipo imprimível

Isso faz com que o Haskell crie automaticamente uma instância da função *show* para esse tipo de dado.

Tipos Produto

Para usá-lo em uma função devemos sempre envelopar a variável com o construtor.

```haskell
dist :: Ponto -> Ponto -> Double
dist (Ponto x y) (Ponto x' y') = sqrt
                               $ (x-x')^2 + (y-y')^2
```

```
> dist (Ponto 1 2) (Ponto 1 1)
1.0
```

Tipos Produto

Podemos misturar os tipos soma e produto:

```haskell
data Forma = Circunferencia Ponto Double
           | Retangulo Ponto Double Double
```

Tipos Produto

`Circunferencia` e `Retangulo` são funções construtoras:

```
> :t Circunferencia
Circunferencia :: Ponto -> Double -> Forma

> :t Retangulo
Retangulo :: Ponto -> Double -> Double -> Forma
```

Tipos Produto

Uma possível função área seria:

```
area :: Forma -> Double
area (Circunferencia p r) = pi*r^2
area (Retangulo p l a)    = l*a
```

Tipos Registros

Também podemos declarar os tipos produtos em um formato de registros, ou **record types**:

```haskell
data Contato = Contato { nome :: String, telefone :: String }
  deriving Show
```

Tipos Registros

Com isso ganhamos de brinde funções do tipo *getter* e *setter*:

```haskell
formataContato :: Contato -> String
formataContato c = (nome c) ++ " - " ++ (telefone c)

atualizaContato :: Contato -> String -> Contato
atualizaContato c t = c {telefone = t}

contato = Contato "Maria" "9999-9999"

main = do
  print (formataContato contato)
  print (formataContato $ atualizaContato contato "8888-8888")
```

Tipos parametrizados

As declarações de tipos também podem ser parametrizados, considere o tipo `Maybe`:

```haskell
data Maybe a = Nothing | Just a
```

A declaração indica que um tipo `Maybe a` pode não ser nada ou pode ser apenas o valor de um tipo `a`.

Maybe

Esse tipo pode ser utilizado para ter um melhor controle sobre erros e exceções:

```haskell
-- talvez a divisão retorne um Int
safeDiv :: Int -> Int -> Maybe Int
safeDiv _ 0 = Nothing
safeDiv m n = Just (m `div` n)

safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Just (head xs)
```

Maybe

Eses erros podem ser capturados com a expressão `case`:

```
divComErro :: Int -> Int -> Int
divComErro m n = case (safeDiv m n) of
                   Nothing -> error "divisão por 0"
                   Just x  -> x
```

Newtype

Uma terceira forma de criar um novo tipo é com a função `newtype`, que serve de intermediário entre `type` e `data`:

```
newtype MinhaString = S [Char]
```

A diferença entre `type` e `newtype` é que o primeiro é um sinônimo, enquanto o segundo define efetivamente um novo tipo:

```haskell
f1 :: String -> Int
f1 s = length s

f2 :: MinhaString -> Int
f2 (S s) = length s
```

Newtype

A diferença entre `type` e `newtype` é que o primeiro é um sinônimo, enquanto o segundo define efetivamente um novo tipo:

```
ghci> let x = "abc" :: [Char]
ghci> f1 x
3
ghci> f2 x
Error!
ghci> f2 (S "abc")
3
```

Tipos Recursivos

Árvore Binária

Um exemplo de tipo recursivo é a árvore binária, que pode ser definida como:

```haskell
data Tree a = Leaf a | Node (Tree a) a (Tree a)
```

ou seja, ou é um nó folha contendo um valor do tipo *a*, ou é um nó contendo uma árvore à esquerda, um valor do tipo *a* no meio e uma árvore à direita.

Árvore Binária

Podemos definir uma função `contem` que indica se um elemento `x` está contidado em uma árvore `t`:

```haskell
contem :: Eq a => Tree a -> a -> Bool
contem (Leaf y) x     = x == y
contem (Node l y r) x = x == y || l `contem` x
                               || r `contem` x
```

```
> t `contem` 5
True
> t `contem` 0
False
```

Classes de Tipo

Clases de Tipo

Classes de tipo são classes que definem grupos de tipos que devem conter algumas funções especificadas.

Para criar um novo tipo utilizamos a função `class`:

```
class Eq a where
   (==), (/=) :: a -> a -> Bool

   x /= y = not (x == y)
```

Essa declaração diz: *para um tipo* `a` *pertencer a classe* `Eq` *deve existir uma implementação das funções* `(==)` *e* `(/=)`.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  x /= y = not (x == y)
```

Além disso, ela já define uma definição padrão da função `(/=)`, então basta definir `(==)`.

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

  x /= y = not (x == y)
```

Para definirmos uma nova **instância** de uma classe basta declarar:

```
instance Eq Bool where
   False == False = True
   True  == True  = True
   _     == _     = False
```

Instâncias da Classe

Apenas tipos definidos por `data` e `newtype` podem ser instâncias de alguma classe.

Classes de Tipo

Uma classe pode estender outra para formar uma nova classe.

Considere a classe `Ord`:

```haskell
class Eq a => Ord a where
   (<), (<=), (>), (>=) :: a -> a -> Bool
   min, max             :: a -> a -> a

   min x y | x <= y    = x
           | otherwise = y

   max x y | x <= y    = y
           | otherwise = x
```

Ou seja, antes de ser uma instância de `Ord`, o tipo deve ser **também** instância de `Eq`.

Classes de tipos

Lembrando:

- **Tipo:** coleção de valores relacionados.
- **Classe:** coleção de tipos que suportam certas funções ou operadores.
- **Métodos:** funções requisitos de uma classe.

Classes interessantes

- **Eq:** relação de igualdade.
- **Ord:** relação de ordem.
- **Show:** transformar um tipo em String.
- **Read:** transformar uma String em outro tipo (parsing).
- **Enum:** deriva um tipo enumerativo que tem sucessor e predecessor.

Classes interessantes

- **Num:** classe numérica.
- **Integral:** classe dos inteiros.
- **Floating:** classe dos números em ponto flutuante.

Info

No ghci, o comando `:info` mostra informações sobre os tipos e as classes de tipo:

```
> :info Integral
class (Real a, Enum a) => Integral a where
  quot :: a -> a -> a
  rem :: a -> a -> a
  div :: a -> a -> a
  mod :: a -> a -> a
  quotRem :: a -> a -> (a, a)
  divMod :: a -> a -> (a, a)
  toInteger :: a -> Integer
  {-# MINIMAL quotRem, toInteger #-}
```

Info

No ghci, o comando `:info` mostra informações sobre os tipos e as classes de tipo:

```
> :info Bool
data Bool = False | True    -- Defined in ‘GHC.Types’
instance Eq Bool -- Defined in ‘GHC.Classes’
instance Ord Bool -- Defined in ‘GHC.Classes’
instance Show Bool -- Defined in ‘GHC.Show’
instance Read Bool -- Defined in ‘GHC.Read’
instance Enum Bool -- Defined in ‘GHC.Enum’
instance Bounded Bool -- Defined in ‘GHC.Enum’
```

Derivação de instâncias

Em muitos casos o Haskell consegue inferir as instâncias das classes mais comuns, nesses casos basta utilizar a palavra-chave `deriving` ao definir um novo tipo:

```haskell
data Bool = False | True
            deriving (Eq, Ord, Show, Read)
```

Exemplo

Vamos definir uma instância de `Ord` para o seguinte tipo:

```
data Dia = Dom | Seg | Ter | Qua | Qui | Sex | Sab
                   deriving (Show, Eq)
```

Instância de Enum

```
> :info Enum
class Enum a where
  succ :: a -> a
  pred :: a -> a
  toEnum :: Int -> a
  fromEnum :: a -> Int
  enumFrom :: a -> [a]
  enumFromThen :: a -> a -> [a]
  enumFromTo :: a -> a -> [a]
  enumFromThenTo :: a -> a -> a -> [a]
  {-# MINIMAL toEnum, fromEnum #-}
        -- Defined in ‘GHC.Enum’
 ...
```

Instância de Enum

```
...
  -- Defined at /tmp/teste/teste/src/Main.hs:21:10
instance Enum Word -- Defined in ‘GHC.Enum’
instance Enum Ordering -- Defined in ‘GHC.Enum’
instance Enum Integer -- Defined in ‘GHC.Enum’
instance Enum Int -- Defined in ‘GHC.Enum’
instance Enum Char -- Defined in ‘GHC.Enum’
instance Enum Bool -- Defined in ‘GHC.Enum’
instance Enum () -- Defined in ‘GHC.Enum’
instance Enum Float -- Defined in ‘GHC.Float’
instance Enum Double -- Defined in ‘GHC.Float’
```

Instância de Enum para Dia

Precisamos definir `toEnum` e `fromEnum`:

```
indicesDias :: [(Dia, Int)]
indicesDias = [
    (Dom, 0), (Seg, 1), (Ter, 2), (Qua, 3)
  , (Qui, 4), (Sex, 5), (Sab, 6)]

instance Enum Dia where

  fromEnum d = head [i |(d', i) <- indicesDias, d' == d]

  toEnum i = dia
    where
      (dia, _) = indicesDias !! i
```

Instância de Enum para Dia

Agora podemos fazer coisas como:

```
> [Seg .. Sex]
[Seg,Ter,Qua,Qui,Sex]

> succ Sex
Sab
```

Instância de Enum para Dia

Também podemos gerar a lista dos dias da semana com:

```
> enumFrom Seg
[Seg,Ter,Qua,Qui,Sex,Sab,*** Exception: Prelude.!!: index too l
```

Ops!

Exercício

Torne o tipo Dia que criamos membro da classe de tipos Bounded. Veja a definição desta classe abaixo:

```
> :info Bounded
class Bounded a where
  minBound :: a
  maxBound :: a
  {-# MINIMAL minBound, maxBound #-}
  ...
```

```haskell
instance Enum Dia where
  ...
  enumFrom d =
    map toEnum [fromEnum d .. fromEnum(maxBound ::Dia)]
```

E então:

```haskell
> [Seg ..]
[Seg,Ter,Qua,Qui,Sex,Sab]
```

Instância de Ord

Com o nosso tipo `Dia` sendo parte da classe de tipos Enum, fica fácil criar uma instância Ord (que só necessita da definição de `<=`):

```haskell
instance Ord Dia where
  (<=) a b = fromEnum a <= fromEnum b
```