Algumas soluções

99 questions - Questão 1

Encontre o último elemento de uma lista:

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

99 questions - Questão 1

Vamos pensar primeiro na assinatura da função:

```
myLast :: ?? -> ??
```

99 questions - Questão 1

Vamos pensar primeiro na assinatura da função:

```haskell
myLast :: [a] -> a
```

99 questions - Questão 1

Agora os casos triviais:

```haskell
myLast :: [a] -> a
myLast []     = ??
myLast (x:[]) = ??
```

99 questions - Questão 1

Agora os casos triviais:

```haskell
myLast :: [a] -> a
myLast []     = error ”Não existe elementos na lista”
myLast (x:[]) = x
```

Finalmente o caso geral:

```
myLast :: [a] -> a
myLast []     = error "Não existe elementos na lista"
myLast (x:[]) = x
myLast (_:xs) = myLast xs
```

99 questions - Questão 2

E o penúltimo elemento?

```
myButLast :: [a] -> a
myButLast []      = error ”Não existe elementos na lista”
myButLast (x:[]) = x -- essa linha está errada
myButLast (_:xs) = myButLast xs
```

E o penúltimo elemento?

```
myButLast :: [a] -> a
myButLast []        = error ”Não existe elementos na lista”
myButLast (x:_:[]) = x
myButLast (_:xs)   = myButLast xs
```

Project Euler 04

Um número palíndrome é aquele que pode ser lido nas duas direções: 12321

O maior palíndrome construído pelo produto de dois números de dois dígitos é $9009 = 91 \times 99$.

Encontre o maior palíndrome feito pelo produto de dois números de três dígitos.

Basicamente queremos:

```
euler4 :: Integer
euler4 = maior
      $ [x*y | x <- tresDigitos, y <- tresDigitos
                ehPalindrome (x*y)]
```

Project Euler 04

Quais são os números de três dígitos?

```
tresDigitos :: [Integer]
tresDigitos = ??
```

Project Euler 04

Quais são os números de três dígitos?

```haskell
tresDigitos :: [Integer]
tresDigitos = [100..999]
```

Project Euler 04

Para ver se um número é palindrome devemos fazer:

```haskell
ehPalindrome :: Integer -> Bool
ehPalindrome n = n == reverso n
```

Como fazer o reverso?

Project Euler 04

Podemos transformar o número em uma lista de seus dígitos e reconstruir o número por essa lista!

```
reverso :: Integer -> Integer
reverso n = reconstruir $ digitos n
```

Só estamos criando mais problemas?

Podemos transformar o número em uma lista de seus dígitos e reconstruir o número por essa lista!

```
reverso :: Integer -> Integer
reverso n = reconstruir $ digitos n
```

Só estamos criando mais problemas? Não! Estamos quebrando nosso problema em problemas menores!

Para gerar uma lista de dígitos:

```haskell
digitos :: Integer -> [Integer]
digitos n = ??
```

Caso trivial primeiro!!

Project Euler 04

Para gerar uma lista de dígitos:

```
digitos :: Integer -> [Integer]
digitos 0 = []
digitos n = ??
```

Eu posso remover o dígito mais a direita com divisão e resto!

Para gerar uma lista de dígitos:

```
digitos :: Integer -> [Integer]
digitos 0 = []
digitos n = resto : digitos quociente
  where (quociente, resto) = divMod n 10
```

Project Euler 04

Carregue seu arquivo atual no `ghci` e verifique a função `digitos`

Project Euler 04

Carregue seu arquivo atual no `ghci` e verifique a função `digitos`

```
> digitos 1234
[4, 3, 2, 1]
```

Ele já retorna na ordem inversa! Que conveniente!

Agora precisamos reconstruir o número a partir da lista:

```haskell
reconstruir :: [Integer] -> Integer
reconstruir ns = ???
```

Caso base primeiro!

Agora precisamos reconstruir o número a partir da lista:

```
reconstruir :: [Integer] -> Integer
reconstruir []     = 0
reconstruir (x:[]) = x
reconstruir (x:xs) = ??
```

Project Euler 04

Agora precisamos reconstruir o número a partir da lista:

```haskell
reconstruir :: [Integer] -> Integer
reconstruir []     = 0
reconstruir (x:[]) = x
reconstruir (x:xs) = ??
```

Hmm, vamos acrescentar um acumulador nos parâmetros para facilitar!

Project Euler 04

Agora precisamos reconstruir o número a partir da lista:

```
reconstruir :: Integer -> [Integer] -> Integer
reconstruir acc []     = acc
reconstruir acc (x:[]) = 10*acc + x
reconstruir acc (x:xs) = ??
```

Project Euler 04

Agora precisamos reconstruir o número a partir da lista:

```haskell
reconstruir :: Integer -> [Integer] -> Integer
reconstruir acc []     = acc
reconstruir acc (x:[]) = 10*acc + x
reconstruir acc (x:xs) = reconstruir (10*acc + x) xs
```

Ufa! Vamos conferir!

Project Euler 04

```
> reconstruir 0 $ digitos 1234
4321
```

Project Euler 04

Agora temos que:

```haskell
reverso :: Integer -> Integer
reverso n = reconstruir 0 $ digitos n
```

Só precisamos agora criar a função `maior`:

```
maior :: [Integer] -> Integer
maior []     = error "Não existe maior em lista vazia!"
maior (x:[]) = x
maior (x:xs) = ??
```

Ou `x` é maior que o maior de `xs` ou o contrário!

Project Euler 04

```haskell
maior :: [Integer] -> Integer
maior []      = error "Não existe maior em lista vazia!"
maior (x:[]) = x
maior (x:xs) | x > resto = x
             | otherwise = resto
             where resto = maior xs
```

Agora é só testar a função! O resultado deve ser 906609.

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.

Criem a função `duasVezes` que recebe uma função `f` e um argumento qualquer `x` e aplica `f` em `x` duas vezes seguida:

```
duasVezes :: (a -> a) -> a -> a
duasVezes f x = f (f x)
```

Funções com funções

Essa função são aplicáveis em diversas situações:

```
> duasVezes (*2) 3
12

> duasVezes reverse [1,2,3]
[1,2,3]
```

Aplicação parcial

O Haskell permite que uma função de $n > 1$ argumentos seja aplicada **parcialmente** definindo uma função com $m < n$ argumentos:

```haskell
soma x y z = x + y + z

soma2 x y  = soma 2

soma23 x   = soma2 3

> soma2 1 2
5

> soma23 1
6
```

Com isso podemos criar novas funções a partir da nossa função `duasVezes`:

```
quadruplica = duasVezes (*2)
```

Considere o padrão comum:

```haskell
[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:

```
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 primo [1..10]
[2,3,5,7]
```

Filter

Podemos passar funções parciais também como argumento:

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

> filter (/= ' ') "abc def ghi"
"abcdefghi"
```

Map e Filter

As duas funções `map` e `filter` costumam serem utilizadas juntas, assim como na compreensão de listas:

```haskell
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

Podemos utilizar o operador $ para separar as aplicações das funções e remover os parênteses:

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

A execução é de baixo para cima.

Outras funções de alta ordem

Outras funções úteis durante o curso:

```
> all even [2,4,6,8]
True

> any odd [2,4,6,8]
False

> takeWhile even [2,4,6,7,8]
[2,4,6]

> dropWile even [2,4,6,7,8]
[7,8]
```

Project Euler 04

Dada a função `cartesiano`, reescreva a função `euler4` utilizando `map` e `filter`

```
euler4 :: Integer
euler4 = maior
       $ [x*y | x <- tresDigitos, y <- tresDigitos
              ehPalindrome (x*y)]

cartesiano :: [Integer] -> [Integer] -> [Integer]
cartesiano xs ys = [(x,y) | x <- xs, y <- ys]
```

Dada a função `cartesiano`, reescreva a função `euler4` utilizando `map` e `filter`

```
euler4 :: Integer
euler4 = maior
       $ filter ehPalindrome
       $ map (\(x,y) -> x*y)
       $ cartesiano tresDigitos tresDigitos

cartesiano :: [Integer] -> [Integer] -> [Integer]
cartesiano xs ys = [(x,y) | x <- xs, y <- ys]
```

Folding

Função foldr

Considerem as funções recursivas:

```
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:

```
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`

O nome dessa função significa `dobrar`, pois ela justamente dobra a lista aplicando a função `f` em cada elemento da lista e um resultado parcial.

```
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 = foldr (+) 0
```

Exercício

Defina `product` utilizando `foldr`.

Resposta

```
product = foldr (*) 1
```

Função length

Como podemos implementar `length` utilizando `foldr`?

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

Função `length`

Para a lista:

```
1 : (2 : (3 : []))
```

devemos obter:

```
1 + (1 + (1 + 0))
```

Função `length`

Da assinatura de `foldr`:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Percebemos que na função `f` o primeiro argumento é um elemento da lista e o segundo é o valor acumulado.

Dessa forma podemos utilizar a seguinte função anônima:

```
length = foldr (\_ n -> 1+n) 0
```

Exercício

Reescreva a função `reverse` utilizando `foldr`:

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Resposta

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

snoc x xs = xs ++ [x]
reverse = foldr snoc []
```

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
```

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
```

Função `length`

Como ficaria a função `length` utilizando `foldl`?

```
length = foldr (\_ n -> 1+n) 0
length = foldl (??) 0
```

Função `length`

Basta inverter a ordem dos parâmetros:

```haskell
length = foldr (\_ n -> 1+n) 0
length = foldl (\n _ -> n+1) 0
```

Função `reverse`

E a função `reverse`?

Resposta

```
1 : (2 : (3 : []))
  => (([] f 3) f 2) f 1

f xs x = ???
reverse = foldl f []
```

Resposta

```haskell
1 : (2 : (3 : []))
  => (([] f 3) f 2) f 1

f xs x = x:xs
reverse = foldl f []
```

`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.

Composição de funções

Composição de funções

Na matemática a composição de função $f \circ g$ define uma nova função z tal que $z(x) = f(g(x))$.

No Haskell temos o operador `(.)`:

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

Dada uma função que mapeia do tipo b para o tipo c, e outra que mapeia do tipo a para o tipo b, gere uma função que mapeie do tipo a para o tipo c.

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

Propriedades da composição

A composição de função é associativa:

```
(f . g) . h == f . (g . h)
```

Propriedades da composição

E tem um elemento nulo que é a função `id`:

```
f . id = id . f = f
```

Propriedades da composição

Essas duas propriedades são importantes durante a construção de programas, pois elas permitem o uso do `foldr` (e dentre outras funções de alta ordem):

```
-- cria uma função que é a composição de uma lista de funções
compose :: [a -> a] -> (a -> a)
compose = foldr (.) id
```

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]
```

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

Porém, não podemos definir tipos recursivos:

```haskell
type Tree = (Int, [Tree])
```

mas temos outras formas de definir tais tipos...

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 t = head [v | (k',v) <- t, k == k']
```

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

Exercício

Crie uma função `paraCima` do tipo `Trans` definido anteriormente que ande para cima dado uma coordenada (some +1 em y).

Resposta

```haskell
paraCima :: Trans
paraCima (x,y) = (x,y+1)
```

Declaração de tipo

Como esses tipos são apenas apelidos, eu posso fazer:

```
array = [(1,3), (5,4), (2,3), (1,1)] :: [(Int, Int)]
> find 2 array
3

array' = [(1,3), (5,4), (2,3), (1,1)] :: Assoc Int Int
> find 2 array
3
```

O compilador não distingue um do outro.

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 -> Trans
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)
```

Exemplo

E a função `caminhar`:

```
caminhar :: [Dir] -> Trans
caminhar []     coord = coord
caminhar (d:ds) coord = caminhar ds (para d coord)
```

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

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.

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 = Circulo Ponto Double
           | Retangulo Ponto Double Double

-- um quadrado é um retângulo com os dois lados iguais
quadrado :: Ponto -> Double
quadrado p n = Retangulo p n n
```

Tipos Produto

`Circulo` e `Retangulo` são funções construtoras:

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

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

Tipos parametrizados

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

```
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:

```
-- 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
```

Either

Outras tipo interessante é o `Either` definido como:

```haskell
data Either a b = Left a | Right b
```

Esse tipo permite que uma função retorne dois tipos diferentes, dependendo da situação.

Either

```haskell
-- ou retorna uma String ou um Int
safeDiv' :: Int -> Int -> Either String Int
safeDiv' _ 0 = Left "divisão por 0"
safeDiv' m n = Right (m `div` n)

> safeDiv' 2 2
1
> safeDiv' 2 0
"divisão por 0"
```

Exercício

Crie um tipo `Fuzzy` que pode ter os valores `Verdadeiro`, `Falso`, `Pertinencia Double`, que define um intermediário entre `Verdadeiro` e `Falso`.

Crie uma função `fuzzifica` que recebe um `Double` e retorna `Falso` caso o valor seja menor ou igual a 0, `Verdadeiro` se for maior ou igual a 1 e `Pertinencia v` caso contrário.

Resposta

```haskell
data Fuzzy = Falso | Pertinencia Double | Verdadeiro

fuzzifica :: Double -> Fuzzy
fuzzifica x | x <= 0     = Falso
            | x >= 1     = Verdadeiro
            | otherwise = Pertinencia x
```

Newtype

Uma terceira forma de criar um novo tipo é com a função `newtype`, que permite apenas um construtor:

```haskell
newtype Nat = N Int
```

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

A diferença entre `newtype` e `data` é que o primeiro define um novo tipo até ser compilado, depois ele é substituído como um sinônimo. Isso ajuda a garantir a checagem de tipo em tempo de compilação.

Tipos Recursivos

Árvore Binária

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

```
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 meior e uma árvore à direita.

Árvore Binária

Desenhe a seguinte árvore:

```
t :: Tree Int
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5
       (Node (Leaf 6) 7 (Leaf 9))
```

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
```

Exercício

Altere a função `contem` levando em conta que essa é uma árvore de busca, ou seja, os nós da esquerda são menores ao nó atual, e os nós da direita são maiores.

Resposta

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

Classes de Tipo

Clases de Tipo

Aprendemos em uma aula anterior sobre as classes de tipo, 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)
```

Clases de Tipo

Essa declaração diz: *para um tipo* a *pertencer a classe* Eq *deve ter 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 extender outra para forma uma nova classe.

Considere a classe `Ord`:

```
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`.

Instância de Ord

Seguindo nosso exemplo de Booleano, temos:

```haskell
instance Ord Bool where
    False < True = True
    _     < _    = False

    b <= c = (b < c) || (b == c)
    b > c  = c < b
    b >= c = c <= b
```

Lembrando:

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

Eq - classe da igualdade

Tipos que podem ser comparados em igualdade e desigualdade:

```haskell
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

Eq - classe da igualdade

```
> 1 == 2
False
> [1,2,3] == [1,2,3]
True
> "Ola" /= "Alo"
True
```

Ord - classe de ordem

A classe Eq acrescido de operadores de ordem:

```
(<) :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>) :: a -> a -> Bool
(>=) :: a -> a -> Bool
min :: a -> a -> a
max :: a -> a -> a
```

Ord - classe de ordem

```
> 4 < 6
> min 5 0
> max 'c' 'h'
> "Ola" <= "Olaf"
```

Show - classe imprimíveis

A classe `Show` define como imprimir um valor de um tipo:

```
show :: a -> String
```

Show - classe imprimíveis

```
> show 10.0
> show [1,2,3,4]
```

Read - classe legíveis

A classe Read define como ler um valor de uma String:

```
read :: String -> a
```

Read - classe legíveis

Precisamos especificar o tipo que queremos extrair da String:

```
> read "12.5" :: Double
> read "False" :: Bool
> read "[1,3,4]" :: [Int]
```

Num - classe numérica

A classe `Num` define todos os tipos numéricos e deve ter as instâncias de:

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs :: a -> a
signum :: a -> a
fromInteger :: Integer -> a
```

Num - classe numérica

```
> 1 + 3
> 6 - 9
> 12.3 * 5.6
```

Num - classe numérica

O que as seguintes funções fazem? (use o `:t` para ajudar)

```
> negate 2
> abs 6
> signum (-1)
> fromInteger 3
```

Num - classe numérica

- **negate:** inverte o sinal do argumento.
- **abs:** retorna o valor absoluto.
- **signum:** retorna o sinal do argumento.
- **fromInteger:** converte um argumento do tipo inteiro para numérico.

Note que os valores negativos devem ser escritos entre parênteses para não confundir com o operador de subtração.

Integral - classe de números inteiros

A classe `Integral` define todos os tipos numéricos inteiros e deve ter as instâncias de:

```haskell
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
```

O uso de crases transforma uma função em operador infixo.

```
> quot 10 3 == 10 `quot` 3
```

Integral - classe de números inteiros

```
> 10 `quot` 3
> 10 `rem` 3
> 10 `div` 3
> 10 `mod` 3
```

As funções `quot` e `rem` arredondam para o 0, enquanto `div` e `mod` para o infinito negativo.

Fractional - classe de números inteiros

A classe `Fractional` define todos os tipos numéricos fracionários e deve ter as instâncias de:

```
(/) :: a -> a -> a
recip :: a -> a
```

Fractional - classe de números inteiros

```
> 10 / 3
> recip 10
```

Outros operadores e funções úteis

Qual a diferença entre esses dois operadores de exponenciação?

```
(^) :: (Num a, Integral b) => a -> b -> a
(**) :: Floating a => a -> a -> a
```

Floating - classe de números de ponto flutuante

```
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
```

Floating - classe de números de ponto flutuante

```haskell
sin :: a -> a
cos :: a -> a
tan :: a -> a
```

Floating - classe de números de ponto flutuante

```
asin :: a -> a
acos :: a -> a
atan :: a -> a
```

Floating - classe de números de ponto flutuante

```
sinh :: a -> a
cosh :: a -> a
tanh :: a -> a
asinh :: a -> a
acosh :: a -> a
atanh :: a -> a
```

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)
```

Classe Enum

Implementa as funções:

`succ, pred, toEnum, fromEnum`

Classe Enum

```haskell
data Dias = Seg | Ter | Qua | Qui | Sex | Sab | Dom
                    deriving (Show, Enum)
```

Enum é enumerativo:

```haskell
succ Seg == Ter
pred Ter  == Seg
fromEnum Seg == 0
toEnum 1 :: Dias == Ter
```

Exercício

Defina um tipo para jogar o jogo Pedra, Papel e Tesoura e defina as funções `ganhaDe`, `perdeDe`.

Defina também uma função denominada `ganhadores` que recebe uma lista de jogadas e retorna uma lista dos índices das jogadas vencedoras.

Resposta

```haskell
data Jogada = Pedra | Papel | Tesoura
              deriving (Show, Enum, Eq, Ord)

ganhaDe :: Jogada -> Jogada -> Bool
Pedra   `ganhaDe` Tesoura = True
Papel   `ganhaDe` Pedra   = True
Tesoura `ganhaDe` Papel   = True
j1      `ganhaDe` j2      | j1 == j2  = True
                          | otherwise = False
```

Resposta

```haskell
perdeDe :: Jogada -> Jogada -> Bool
j1 `perdeDe` j2 = not $ j1 `ganhaDe` j2

ganhador :: Jogada -> [Jogada] -> Bool
ganhador j js = all (j `ganhaDe`) js
```

Resposta

```haskell
ganhadores :: [Jogada] -> [Int]
ganhadores js = map snd
              $ filter ((j,i) -> ganhador j js)
              $ zip js [0..]
```

Projeto para casa