Tipos e Classes

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Paradigmas de Programação na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

Tipos e Classes Padrões

Tipos de dados

UFABC

- Um tipo é uma coleção de valores relacionados entre si.

Exemplos

- `Int` compreende todos os valores de números inteiros.
- `Bool` contém apenas os valores `True` e `False`, representando valores lógicos

Tipos de dados

UFABC

- Em Haskell, os tipos são definidos pela notação

```
1  v :: T
```

- Significando que v define um valor do tipo T.

Tipos de dados

UFABC

```
1 False :: Bool
2 True :: Bool
3 10 :: Int
```

Tipos de funções

UFABC

- De forma similar uma função pode ser definida por

```haskell
f :: T0 -> T1
```

- Indicando que a função `f` recebe um valor do tipo `T0` e retorna um valor do tipo `T1`.

Tipos avaliados

UFABC

- O tipo da aplicação de uma função é o tipo do seu retorno:

```
1 False :: Bool
2 not :: Bool -> Bool
3 not False :: Bool
```

Inferência de tipo

UFABC

- Em Haskell, **toda** expressão tem um tipo calculado antes de avaliar o resultado da expressão.
- Os tipos podem ser definidos automaticamente pela inferência do tipo.

UFABC

Por exemplo, se eu tenho:

```
1 f :: A -> B
2 e :: A
```

então

```
1 f e :: B
```

Exemplo

UFABC

```haskell
impar x = x `rem` 2 == 1
```

Pergunta

Qual o tipo da função?

Exemplo

Abra o `ghci` e digite:

```
1 :t (`rem` 2)
```

Exemplo

UFABC

Abra o `ghci` e digite:

```
1 :t (`rem` 2)
2 (`rem` 2) :: Integral a => a -> a
```

Exemplo

UFABC

Logo `x` deve ser do tipo `Integral` e a função deve ser:

```
1 impar :: Integral a => a -> ???
2 impar x = x `rem` 2 == 1
```

Exemplo

UFABC

```
1 :t (== 1)
2 (== 1) :: (Eq a, Num a) => a -> Bool
```

Isso restringe ainda mais nosso tipo, como veremos mais a frente. Por ora, observemos `-> Bool`.

Exemplo

UFABC

A assinatura da função fica:

```haskell
impar :: (Eq a, Integral a) => a -> Bool
impar x = x `rem` 2 == 1
```

Exemplo

UFABC

Se eu fizer (ou tentar):

```
1 r1 = impar "3"
```

Isso vai gerar um erro de compilação!

Exemplo

UFABC

```
  • No instance for (Integral [Char]) arising from
a use of ‘impar’
  • In the expression: impar "3"
    In an equation for ‘r1’: r1 = impar "3"
```

Tipos Básicos

Tipos Básicos

UFABC

- O compilador GHC já vem com suporte nativo a diversos tipos básicos.
- Durante o curso veremos como definir e criar os nossos próprios tipos.

Tipos Básicos

Os tipos são:

- `Bool`: contém os valores `True` e `False`. Expressões booleanas podem ser executadas com os operadores `&&` (e), `||` (ou) e `not`.
- `Char`: contém todos os caracteres no sistema Unicode. Podemos representar a letra `'a'`, o número `'5'`, a seta tripla `'⇉'` e o *homem de terno levitando*[1] `'🕴'`.
- `String`: sequências de caracteres delimitados por aspas duplas: `"Olá Mundo"`.

[1] Este é o nome oficial do caracter na tabela Unicode v.7.0!

Tipos Básicos

- `Int`: inteiros com precisão fixa em 64 bits. Representa os valores numéricos de -2^{63} até $2^{63} - 1$.
- `Integer`: inteiros de precisão arbitrária. Representa valores inteiros de qualquer precisão, a memória é o limite. Mais lento do que operações com `Int`.
- `Float`: valores em ponto-flutuante de precisão simples. Permite representar números com um total de 7 dígitos, em média.
- `Double`: valores em ponto-flutuante de precisão dupla. Permite representar números com quase 16 dígitos, em média.

Note que ao escrever:

```
1 x = 3
```

O tipo de `x` pode ser `Int, Integer, Float` ou `Double`.

Pergunta

Qual tipo devemos atribuir a `x`?

Listas

Listas são sequência de elementos do mesmo tipo agrupados por colchetes e separados por vírgula:

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

UFABC

Uma lista de tipo `T` tem tipo `[T]`:

```haskell
[1,2,3,4] :: [Int]
[False, True, True] :: [Bool]
['o', 'l', 'a'] :: [Char]
```

UFABC

- O tamanho da lista (*length*) representa a quantidade de elementos que ela contém.
- Uma lista vazia é representada por `[]`
- Listas com um elemento, como `[1]`, `[False]` e `[[]]` são chamadas singleton.

UFABC

- Como podem ter percebido no slide anterior, podemos ter listas de listas:

```
1 [ [1,2,3], [4,5] ] :: [[Int]]
2 [ [ 'o','l','a'], ['m','u','n','d','o'] ] :: [[Char]]
```

Notem que:

- O tipo da lista não especifica seu tamanho
- Não existe limitação quanto ao tipo da lista
- Não existe limitação quanto ao tamanho da lista

Tuplas

UFABC

- Tuplas são sequências finitas de componentes, contendo zero ou mais tipos diferentes:

```
1 (True, False) :: (Bool, Bool)
2 (1.0, "Sim", False) :: (Double, String, Bool)
```

- O tipo da tupla é definido como `(T1, T2,...,Tn)`.

UFABC

- O número de componentes de uma tupla é chamado aridade (*arity*)
- Uma tupla de aridade zero, a tupla vazia, é representada por ( )
- Tuplas de tamanho dois são conhecidas como *duplas*, já as de tamanho três são *triplas*.

UFABC

Notem que:

- O tipo da tupla especifica seu tamanho
- Não existe limitações aos tipos associados à tupla (podemos ter tuplas de tuplas)
- Tuplas **devem** ter um tamanho finito (e fixo!)
- Tuplas de aridade 1 não são permitidas para manter compatibilidade do uso de parênteses para definir a ordem de avaliação

UFABC

- Funções são mapas de argumentos de um tipo para resultados em outro tipo. O tipo de uma função é escrita como `T1 -> T2`, ou seja, o mapa do tipo `T1` para o tipo `T2`:

```haskell
not  :: Bool -> Bool
even :: Int -> Bool
```

UFABC

- Como não existem restrições para os tipos, a noção de mapa de um tipo para outro é suficiente para escrever funções com 0 ou mais argumentos e que retornem 0 ou mais valores.

Exercício

Crie as seguintes funções em um arquivo **aula02.hs**, carregue no **ghci**, verifique seu tipo e teste com algumas entradas:

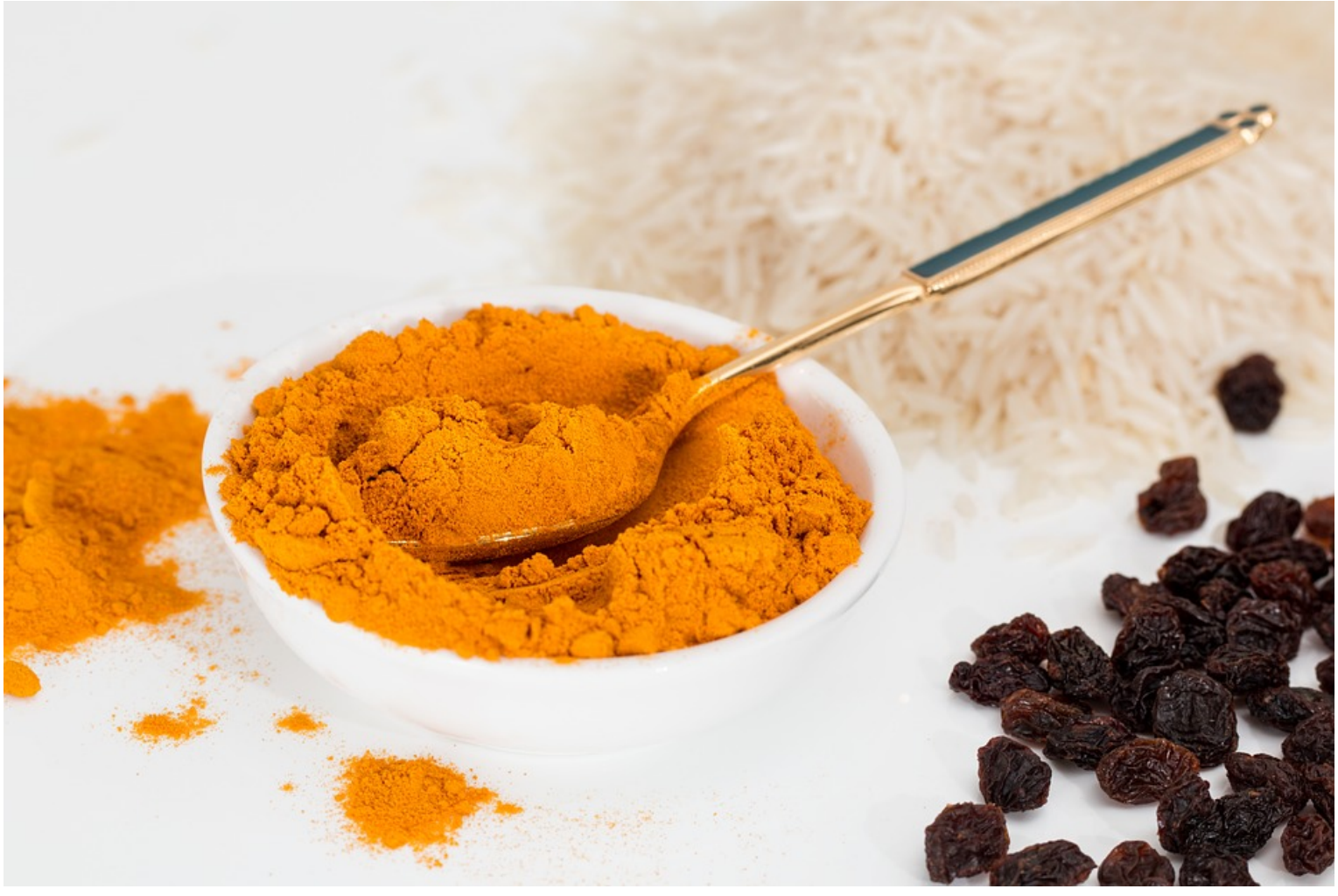
```
1 soma :: (Int, Int) -> Int
2 soma (x,y) = x+y
3
4 zeroAteN :: Int -> [Int]
5 zeroAteN n = [0..n]
```

UFABC

- Uma função pode ser total se ela for definida para qualquer valor do tipo de entrada ou parcial se existem algumas entradas para qual ela não tem valor de saída definido:

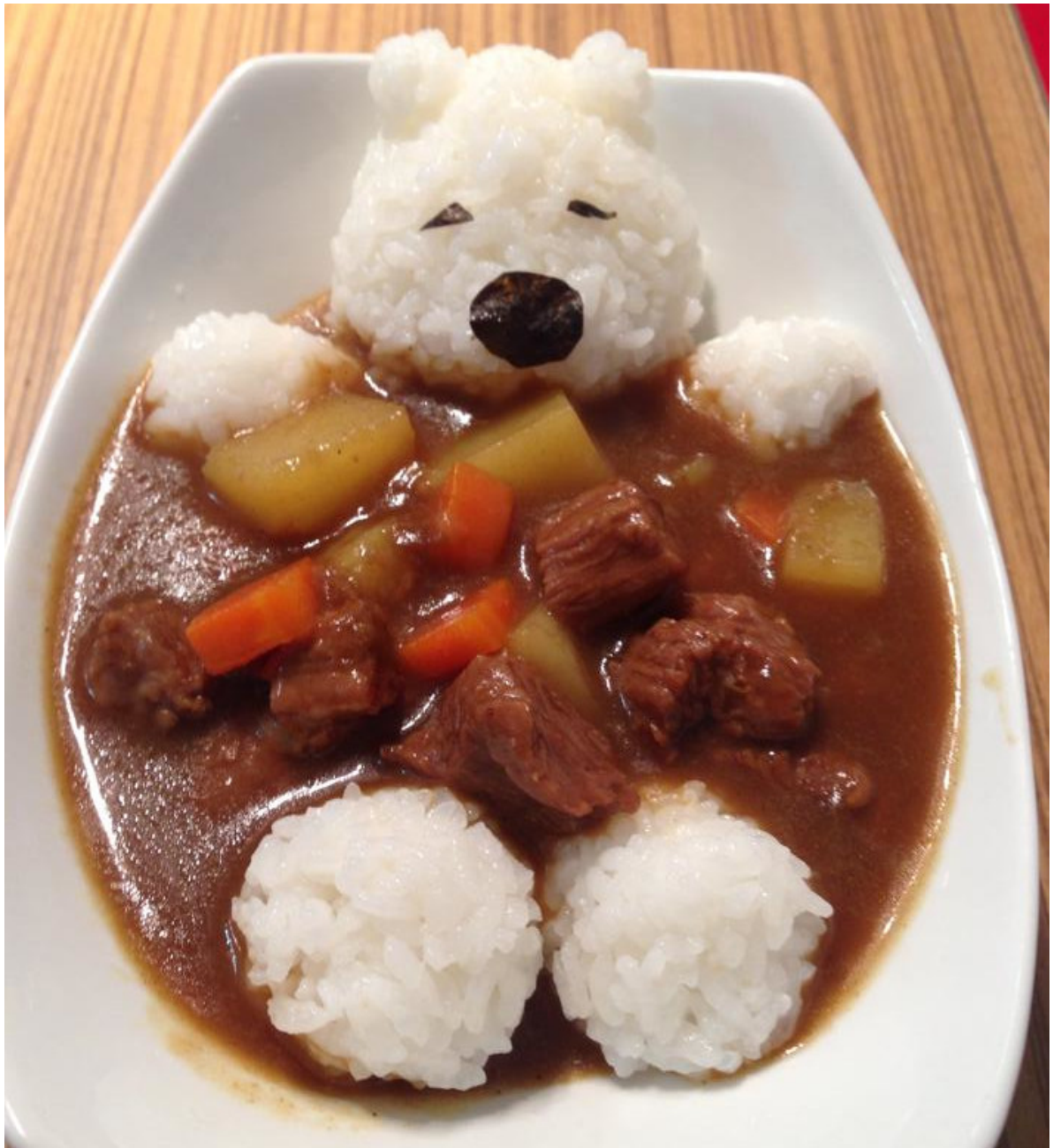
```
1 > head []
2 *** Exception: Prelude.head: empty list
```

Curry



Curry

UFABC



UFABC

- É óbvio que o pessoal do Unicode também criou um símbolo para o curry japonês (karê, カレー):  .

```
1 Prelude> :t '🍛'
2 '🍛' :: Char
3 Prelude>
```

- Funções com múltiplos argumentos podem ser definidas de uma outra forma, inicialmente não óbvia, mas que torna sua representação mais natural.

Como não existem restrições de tipos, uma função pode retornar uma outra função:

```haskell
soma' :: Int -> (Int -> Int)
soma' x = \y -> x + y
```

UFABC

- Ela recebe um valor `x` e retorna uma função que recebe um `y` e devolve `y + x` (aprenderemos sobre `\y` mais adiante).

```haskell
soma' :: Int -> (Int -> Int)
soma' x = \y -> x + y
```

UFABC

A seguinte definição ainda é válida:

```haskell
soma' :: Int -> (Int -> Int)
soma' x y = x + y
```

Ela indica que a função `soma'` recebe um valor `x`, cria uma função `\y -> x + y` e aplica com o valor `y`. Isso é conhecido como curried functions.

```haskell
soma' :: Int -> (Int -> Int)
soma' x y = x + y
```

UFABC

Da mesma forma podemos ter:

```haskell
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x*y*z
```

UFABC

Para evitar escrever um monte de parênteses (como no Lisp 🙂), a seguinte sintaxe é válida:

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

mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

Polimorfismo

UFABC

Considere a função `length` que retorna o tamanho de uma lista. Ela deve funcionar para qualquer uma dessas listas:

```
1 [1,2,3,4] :: [Int]
2 [False, True, True] :: [Bool]
3 ['o', 'l', 'a'] :: [Char]
```

Pergunta

Qual é então o tipo de `length`?

Tipos polimórficos

UFABC

- Qual o tipo de `length`?

```haskell
length :: [a] -> Int
```

- Quem é `a`?

Tipos polimórficos

UFABC

- Em Haskell, `a` é conhecida como variável de tipo e ela indica que a função deve funcionar para listas de qualquer tipo.
- As variáveis de tipo devem seguir a mesma convenção de nomes do Haskell, iniciar com letra minúscula. Como convenção utilizamos `a, b, c,...`.

Overloaded types

UFABC

- Considere agora a função `(+)`, diferente de `length` ela pode ter um comportamento diferente para tipos diferentes.
- Internamente somar dois `Int` pode ser diferente de somar dois `Integer` (e definitivamente é diferente de somar dois `Float`).
- Ainda assim, essa função deve ser aplicada a tipos numéricos.

Overloaded types

UFABC

- A ideia de que uma função pode ser aplicada a apenas uma classe de tipos é explicitada pela Restrição de classe (*class constraint*).
- Uma restrição é escrita na forma `C a`, onde `C` é o nome da classe e `a` uma variável de tipo.

```
1  (+) :: Num a => a -> a -> a
```

- A função `+` recebe dois tipos de uma classe numérica e retorna um valor desse mesmo tipo.

UFABC

- Note que nesse caso, ao especificar a entrada como `Int` para o primeiro argumento, todos os outros devem ser `Int` também.

```
1 (+) :: Num a => a -> a -> a
```

UFABC

- Uma vez que uma função contém uma restrição de classe, pode ser necessário definir instâncias dessa função para diferentes tipos pertencentes à classe.
- Os valores também podem ter restrição de classe:

```
1  3 :: Num a => a
```

O que resolve nosso problema anterior.

UFABC

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

UFABC

- Tipos que podem ser comparados em igualdade e desigualdade:

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

Eq - classe da igualdade

UFABC

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

`Ord` - classe de ordem

UFABC

- A classe `Eq` acrescida de operadores[fn::Quando a função é

comumente utilizada de maneira infixa, ela também é usualmente chamada de operador.] de ordem:

```
1 (<) :: a -> a -> Bool
2 (<=) :: a -> a -> Bool
3 (>) :: a -> a -> Bool
4 (>=) :: a -> a -> Bool
5 min :: a -> a -> a
6 max :: a -> a -> a
```

`Ord` - classe de ordem

UFABC

```
1  > 4 < 6
2  > min 5 0
3  > max 'c' 'h'
4  > "Ola" <= "Olaf"
```

Show - classe imprimíveis

UFABC

- A classe **Show** define como imprimir um valor de um tipo:

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

Show - classe imprimíveis

UFABC

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

Read - classe legíveis

UFABC

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

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

Read - classe legíveis

UFABC

- Precisamos especificar o tipo que queremos extrair da String:

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

Num - classe numérica

UFABC

- A classe `Num` define todos os tipos numéricos e deve suas instâncias devem responder à:

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

Num - classe numérica

UFABC

```
1  > 1 + 3
2  > 6 - 9
3  > 12.3 * 5.6
```

Num - classe numérica

UFABC

- 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

UFABC

- A classe `Integral` define todos os tipos numéricos inteiros e suas instâncias devem responder a:

```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
```

`Integral` - classe de números inteiros

UFABC

- O uso de crases transforma uma função em operador infixo.

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

`Integral` - classe de números inteiros

```
1  > 10 `quot` 3
2  > 10 `rem` 3
3  > 10 `div` 3
4  > 10 `mod` 3
```

Pergunta

Pra que 2 de cada?

`Integral` - classe de números inteiros

- As funções `quot` e `rem` arredondam para o 0, enquanto `div` e `mod` para $-\infty$

`Fractional` - classe de números inteiros

UFABC

- A classe `Fractional` define todos os tipos numéricos fracionários e suas instâncias devem responder:

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

`Fractional` - classe de números inteiros

UFABC

```
1  > 10 / 3
2  > recip 10
```

Outros operadores e funções úteis

UFABC

Qual a diferença entre esses dois operadores de exponenciação?

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

Floating - classe de números de ponto flutuante

UFABC

```haskell
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

UFABC

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

Floating - classe de números de ponto flutuante

UFABC

```
1 asin :: a -> a
2 acos :: a -> a
3 atan :: a -> a
```

UFABC

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

UFABC

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

```
1  > :info Integral
2  class (Real a, Enum a) => Integral a where
3    quot :: a -> a -> a
4    rem :: a -> a -> a
5    div :: a -> a -> a
6    mod :: a -> a -> a
7    quotRem :: a -> a -> (a, a)
8    divMod :: a -> a -> (a, a)
9    toInteger :: a -> Integer
10   {-# MINIMAL quotRem, toInteger #-}
```

UFABC

- 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’
```

UFABC

- Escreva as definições para os seguintes tipos em um arquivo `atividade02.hs` e carregue no `ghci`. Não importa o que ela faça, só não pode gerar erro:

```haskell
bools :: [Bool]
nums :: [[Int]]
soma :: Int -> Int -> Int -> Int
copia :: a -> (a, a)
f :: a -> a
g :: Eq a => a -> (a, a) -> Bool
h :: Num a => Int -> a -> a
```