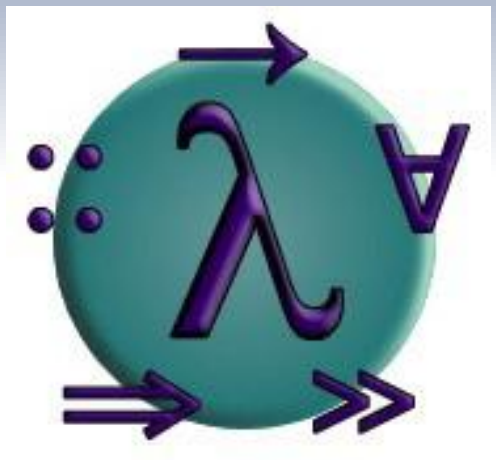

Programação Funcional
Capítulo 3
Tipos e Classes

José Romildo Malaquias

2012.1
Departamento de Computação

Universidade Federal de Ouro Preto

Tópicos

Valores e Expressões

- **Valores** são as entidades básicas de uma linguagem de programação que são manipuladas durante a execução do programa.
- Valores representam os dados que são processados pelo programa.
- **Expressões** são frases de uma linguagem de programação que especificam uma computação: uma expressão pode ser avaliada para produzir um valor.
- Uma expressão **literal** é uma expressão cujo valor é dado diretamente, sem necessidade de avaliação.
- **Literais** são as formas mais simples de expressões.
- Exemplos de literais:

booleanos	`True`, `False`
números	`862`, `78`, `87.0451`
caracteres	`'B'`, `'9'`, `'!'`, `'\n'`
strings	`"vila rica"`, `"java"`
listas	`[56,0,17]`, `["banana","ovo"]`
tuplas	`("Ana",'F',19)`, `(4.2,5.1)`

Tipos

- Um **tipo** é uma coleção de valores relacionados.
- Por exemplo, em Haskell o tipo `Bool` contém os dois valores lógicos `False` e `True`.
- Os tipos servem para classificar os valores de acordo com as suas características.

Tipos básicos

Haskell tem um número de tipos básicos, incluindo:

tipo	**descrição**
`Bool`	valores lógicos
`Char`	caracteres simples
`String`	seqüências de caracteres
`Int`	inteiros de precisão fixa
`Integer`	inteiros de precisão arbitrária
`Float`	núm. em ponto flutuante com precisão simples
`Double`	núm. em ponto flutuante com precisão dupla

Tipos básicos (cont.)

Exemplos:

tipos	valores
`Bool`	`True`, `False`
`Char`	`'B'`, `'5'`, `'!'`, `'\n'`,
`String`	`"vila rica"`, `"Ana Carolina Gomes"`
`Int`	876, 2012
`Integer`	10, 754732483894857274878284000021121231 20
`Float`	4.56, 0.201E10
`Double`	78643, 987.3201E60

Tipo de uma expressão

- Toda expressão tem um tipo associado: o tipo do valor da expressão.
- Se a avaliação de uma expressão e produz um valor do tipo t, então o tipo de e é t, e escrevemos:

```
e :: t
```

- Exemplos:

```
True                  :: Bool
'a'                   :: Char
"maria das dores"     :: String
58                    :: Int
58                    :: Integer
58                    :: Float
58                    :: Double
8.079E10              :: Float
8.079E10              :: Double
not False             :: Bool
(5 + 2.3) * 4         :: Double
2*(5 - 8) <= 6 + 1    :: Bool
```

Consulta do tipo de uma expressão no GHCi

- No GHCi, o comando `:type` calcula o tipo de uma expressão, sem avaliá-la.
- Exemplos:

```
Prelude> not False
True

Prelude> :type not False
not False :: Bool

Prelude> :type 'Z'
'Z' :: Char

Prelude> :t 2*(5 - 8) <= 6 + 1
2*(5 - 8) <= 6 + 1 :: Bool
```

Inferência de tipos

- Toda expressão bem formada tem um tipo mais geral, que pode ser calculado automaticamente em tempo de compilação usando um processo chamado **inferência de tipos**.
- Desta maneira não é necessário declarar os tipos das variáveis nos programas, em geral.

Erros de tipo

- A aplicação de uma função a um ou mais argumentos de tipo inadequado constitui um erro de tipo.
- Por exemplo:

```
> 71 + False

<interactive>:3:4:
    No instance for (Num Bool)
      arising from a use of ‘+’
    Possible fix: add an instance declaration for (Num Bool)
    In the expression: 71 + False
    In an equation for ‘it’: it = 71 + False
```

Problema:
A função (+) requer dois números, porém foi aplicada aos argumentos 71, que é um número, e **False**, que é um valor lógico.

Checagem de tipos

- Haskell é uma linguagem fortemente tipada, com um sistema de tipos muito avançado.
- Todos os erros de tipo são encontrados em tempo de compilação (tipagem estática), o que torna os programas mais seguros e mais rápidos, eliminando a necessidade de verificações de tipo em tempo de execução.

Tópicos

Tipos listas

- Uma **lista** é uma seqüência de valores do mesmo tipo.
- Uma lista pode ser escrita colocando os elementos da seqüência entre colchetes e separados por vírgula.
- Exemplos:

```
[False,True,False]      :: [Bool]
['a','b','c','d']       :: [Char]
["bom","dia","Brasil"]  :: [String]
[1,8,6,0,21]            :: [Int]
[3.4,5.6,3.213]         :: [Double]
```

- Em geral:

```
[ t ]
```

é o tipo das listas cujos elementos são do tipo t.

Tipos listas (cont.)

- A **lista vazia** é escrita como

```
[]
```

- O tipo de uma lista não diz nada sobre seu tamanho:

```
[False,True]       :: [Bool]
[False,True,False] :: [Bool]
```

- Não há restrição sobre o tipo dos elementos de uma lista. Por exemplo, podemos ter listas de listas:

```
[[False,True],[False,True,False]] :: [[Bool]]
[['a'],['b','c'],['%','*'],[]]    :: [[Char]]
[[[1,2,3],[10,20,30]],[[-1,-2]]]  :: [[[Int]]]
```

- O tipo `String` é idêntico ao tipo `[Char]`.

Tópicos

Tipos tupla

- Uma **tupla** é uma seqüência de valores de diferentes tipos.
- Uma tupla é escrita colocando os elementos da seqüência entre parênteses e separados por vírgula.
- Exemplos:

```
(False,True)           :: (Bool,Bool)
(False,'a',True)       :: (Bool,Char,Bool)
("Joel",16,5.34,True)  :: (String,Int,Float,Bool)
```

- Em geral:

$$(\ t_1 , \ t_2 , \ \ldots , \ t_n \ )$$

é o tipo das n-uplas cujos i-ésimo componentes são do tipo t_i, para qualquer $i \in [1..n]$.

Tipos tupla (cont.)

- A **tupla vazia** é escrita como `()`
- O tipo da tupla vazia também é escrito como `()`, e é chamado de **unit**.
- O tipo de uma tupla codifica o seu tamanho:

```
(False,True)       :: (Bool,Bool)
(False,True,False) :: (Bool,Bool,Bool)
```

- O tipo dos componentes de uma tupla é irrestrito. Por exemplo:

```
('a',(False,'b'))     ::(Char,(Bool,Char))
(True,['a','b'])      ::(Bool,[Char])
(5.6,("Ana",18),'F')  ::(Double,(String,Int),Char)
```

Tipos tupla (cont.)

- Não há tuplas de um único elemento por questão de sintaxe.
- Qualquer expressão pode ser escrita entre **parênteses** sem alterar o seu significado.
- Exemplo:

```
('a') :: Char
```

  `('a')` é um valor do tipo `Char`.

  Em Haskell não existe o tipo tupla de um caracter.

- Parênteses são usados para agrupar subexpressões em expressões mais complexas.

Tópicos

Tipos função

- Uma **função** é um mapeamento de valores de um tipo (domínio) em valores de outro tipo (contra-domínio).
- Exemplos de funções do prelúdio:

```
not      :: Bool -> Bool
and      :: [Bool] -> Bool
words    :: String -> [String]
unwords  :: [String] -> String
isDigit  :: Char -> Bool
```

- Em geral:

```
t1 -> t2
```

é o tipo das funções que mapeiam valores do tipo t_1 em valores do tipo t_2.

Tipos função (cont.)

- Os tipos do argumento e do resultado são irrestritos.
- Exemplo: funções com múltiplos argumentos ou resultados são possíveis usando listas ou tuplas:

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

```
zeroto     :: Int -> [Int]
zeroto n    = [0..n]
```

Funções de ordem superior

- **Função de ordem superior** é uma função que tem outra função como argumento, ou produz uma função como resultado.
- Exemplo: A função `filter` do prelúdio recebe uma função e uma lista como argumentos, e filtra os elementos da lista para os quais a função dada retorna verdadeiro.

```
filter even [1,8,10,48,5,-3] ⇝ [8,10,48]
filter odd [1,8,10,48,5,-3]  ⇝ [1,5,-3]
```

`filter` é uma função de ordem superior.

- Exemplo: A função `(.)` do prelúdio recebe duas funções como argumento e resulta em uma terceira função que é a composição das funções dadas.

```
sqrt . abs                ⇝ uma funcao
(sqrt . abs) 9            ⇝ 3
(sqrt . abs) (16 - 25)    ⇝ 3
```

`(.)` é uma função de ordem superior.

Tópicos

Funções polimórficas

- Algumas funções podem operar sobre vários tipos de dados.
- Por exemplo: a função `head` recebe uma lista e retorna o primeiro elemento da lista:

```
head ['b','a','n','a','n','a']    ⇝ 'b'
head ["maria","paula","peixoto"] ⇝ "maria"
head [True,False,True,True]       ⇝ True
head [("ana",2.8),("pedro",4.3)] ⇝ ("ana",2.8)
```

  Não importa qual é o tipo dos elementos da lista.

- Qual é o tipo de `head`?

Funções polimórficas (cont.)

- Quando um tipo pode ser qualquer tipo da linguagem, ele é representado por uma **variável de tipo**.
- No exemplo dado, sendo `a` o tipo dos elementos da lista que é passada como argumento para a função `head`, então

```
head :: [a] -> a
```

`a` é uma variável de tipo e pode ser substituída por qualquer tipo.

O tipo de `head` estabelece que `head` recebe uma lista com elementos de um tipo qualquer, e retorna um valor deste tipo.

Funções polimórficas (cont.)

- Uma função é chamada **polimórfica** (de muitas formas) se o seu tipo contém uma ou mais variáveis de tipo.
- Exemplo:

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

para qualquer tipo `a`, `length` recebe uma lista de valores do tipo `a` e retorna um inteiro.

Funções polimórficas (cont.)

- As variaveis de tipo podem ser instanciadas para diferentes tipos em diferentes circunstâncias:

```
length [False,True] ⇝ 2      - a = Bool
length [1,2,3,4]    ⇝ 4      - a = Int
```

- As variáveis de tipo devem começar com uma letra minúscula, e são geralmente denominadas `a`, `b`, `c`, etc.

Funções polimórficas (cont.)

- Muitas das funções definidas no prelúdio são polimórficas.
- Por exemplo:

```
fst  :: (a,b) -> a
head :: [a] -> a
take :: Int -> [a] -> [a]
zip  :: [a] -> [b] -> [(a,b)]
id   :: a -> a
```

Tópicos

Sobrecarga

- Uma função polimórfica é **sobrecarregada** se seu tipo contém uma ou mais restrições de classe.
- Exemplo:

```
sum :: Num a => [a] -> a
```

> *para qualquer tipo numérico* `a`, `sum` *recebe uma lista de valores do tipo* `a` *e retorna um valor do tipo* `a`.

Sobrecarga (cont.)

- Variáveis de tipo restritas podem ser instanciadas para quaisquer tipos que satisfazer as restrições.
- Exemplo:

```
sum [1,2,3]         ⇝ 6     - a = Int
sum [1.1,2.2,3.3]   ⇝ 6.6   - a = Float
sum ['a','b','c']   ⇝ ERRO DE TIPO: Char não é um tipo numérico
```

Classes básicas

- Uma **classe** é uma coleção de tipos que suportam certas operações sobrecarregadas chamadas **métodos**.
- Haskell tem várias **classes de tipo** pré-definidas.

Classes básicas: Eq

- **Pré-requisitos**: nenhum
- **Caracterização**: tipos com teste de igualdade
- **Métodos**:

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

- **São instâncias**:
  - todos os tipos básicos `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, `Double`
  - todos os tipos listas cujo tipo dos elementos é instância de `Eq`
  - todos os tipos tuplas cujos tipos dos componentes são todos instância de `Eq`
- **Não são instâncias**:
  - tipos função

Classes básicas: Eq (cont.)

- Exemplos:

```
58 == 58                        ⇝ True
'a' == 'b'                      ⇝ False
"abc" == "abc"                  ⇝ True
[1,2] == [1,2,3]                ⇝ False
('a',False) == ('a',False)      ⇝ True
(1,2,3) /= (3,2,1)              ⇝ True
even == odd                     ⇝ ERRO DE TIPO
[even,odd] /= [even . abs]      ⇝ ERRO DE TIPO
'a' == 65                       ⇝ ERRO DE TIPO
```

Classes básicas: Ord

- **Pré-requisitos**: Eq
- **Caracterização**: tipos com ordenação total
- **Métodos**:

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

- O método compare recebe dois valores e retorna um resultado do tipo Ordering: GT, LT ou EQ.
- **São instâncias**:
  - todos os tipos básicos Bool, Char, String, Int, Integer, Float e Double
  - todos os tipos listas cujo tipo dos elementos é instância de Ord
  - todos os tipos tuplas cujos tipos dos componentes são todos instância de Ord
- Listas e tuplas são ordenadas lexicograficamente (da mesma maneira que as palavras em um dicionário).

Classes básicas: Ord (cont.)

- Exemplos:

```
False < True                        ⇝ True
"elegante" < "elefante"             ⇝ False
[1,2,3] > [1,2]                     ⇝ True
('a',2) >= ('b',1)                  ⇝ False
compare "Abracadabra" "Zebra"       ⇝ LT
compare 5 3                         ⇝ GT
compare (10.1,'a') (10.1,'a')       ⇝ EQ
compare ['a','b'] [3,2,7]           ⇝ ERRO DE TIPO
min 'a' 'b'                         ⇝ 'a'
max "amaral" "ana"                  ⇝ "ana"
```

Classes básicas: Show

- **Pré-requisitos**: nenhum
- **Caracterização**: tipos que podem ser convertidos para `String`
- **Métodos**:

```
show :: Show a => a -> String
```

- **São instâncias**:
  - todos os tipos básicos `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` e `Double`
  - todos os tipos listas cujo tipo dos elementos é instância de `Show`
  - todos os tipos tuplas cujos tipos dos componentes são todos instância de `Show`
- **Não são instâncias**:
  - tipos função

Classes básicas: Show (cont.)

- Exemplos:

```
show False            ⇝ "False"
show 'a'              ⇝ "'a'"
show 9172             ⇝ "9172"
show [123,-764,0,18]  ⇝ "[123,-764,0,18]"
show ('a',True)       ⇝ "('a',True)"
show ['a','b','c']    ⇝ "\"abc\""
show ["hi","hello"]   ⇝ "[\"hi\",\"hello\"]"
show "adeus"          ⇝ "\"adeus\""
show even             ⇝ ERRO DE TIPO
```

Classes básicas: `Read`

- **Pré-requisitos**: nenhum
- **Caracterização**: tipos que podem ser convertidos a partir de `String`
- **Métodos**:

```
read :: Read a => String -> a
```

- `Read` e `Show` são duais.
- **São instâncias**:
  - todos os tipos básicos `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` e `Double`
  - todos os tipos listas cujo tipo dos elementos é instância de `Read`
  - todos os tipos tuplas cujos tipos dos componentes são todos instância de `Read`
- **Não são instâncias**:
  - tipos função

Classes básicas: Read (cont.)

- Exemplos:

```
not (read "False")                    ⇝ True
read "67" + 11                        ⇝ 78
read "False" :: Bool                  ⇝ False
read "'a'" :: Char                    ⇝ 'a'
read "123" :: Int                     ⇝ 123
read "[1,2,3]" :: [Int]               ⇝ [1,2,3]
read "[1,2,3]" :: [Double]            ⇝ [1.0,2.0,3.0]
read "('a',5.6)" :: (Char,Double)     ⇝ ('a',5.6)
read "\"False\"" :: String            ⇝ "False"
read "marcos" :: String               ⇝ ERRO: NO PARSE
read "45"                             ⇝ ERRO: AMBIGUIDADE
```

- `read` converte uma string para um valor de um determinado tipo, que deve ser especificado pelo contexto (pode ser inferido) ou por uma anotação explícita de tipo.
- Se não for possível determinar o tipo do resultado de `read`, ocorre um erro de variável de tipo ambígua.

Classes básicas: Enum

- **Pré-requisitos**: `Ord`
- **Caracterização**: tipos que podem ser enumerados
- **Métodos**:

```
succ           :: Enum a => a -> a
pred           :: Enum a => a -> a
toEnum         :: Enum a => Int -> a
fromEnum       :: Enum a => a -> Int
enumFrom       :: Enum a => a -> [a]
enumFromThen   :: Enum a => a -> a -> [a]
enumFromTo     :: Enum a => a -> a -> [a]
enumFromThenTo :: Enum a => a -> a -> a -> [a]
```

- **São instâncias**:
  - os tipos básicos `()`, `Bool`, `Char`, `Int`, `Integer`, `Float`, `Double` e `Ordering`

Classes básicas: Enum (cont.)

- Exemplos:

```
pred 'M'                          ⇝ 'L'
succ 56                           ⇝ 57
succ 'z'                          ⇝ '{'
succ "marcos"                     ⇝ ERRO DE TIPO
fromEnum 'A'                      ⇝ 65
toEnum 65 :: Char                 ⇝ 'A'
fromEnum True                     ⇝ 1
not (toEnum 0)                    ⇝ True
enumFromTo 4 10                   ⇝ [4,5,6,7,8,9,10]
enumFromThenTo 4 7 20             ⇝ [4,7,10,13,16,19]
take 7 (enumFrom 100)             ⇝ [100,101,102,103,104,105,106]
take 10 (enumFromThen 'A' 'C')    ⇝ "ACEGIKMOQS"
```

Classes básicas: Bounded

- **Pré-requisitos:** nenhum
- **Caracterização**: tipos que possuem um valor mínimo e um valor máximo
- **Métodos**:

```
minBound :: Bounded a => a
maxBound :: Bounded a => a
```

- **São instâncias**:
  - os tipos básicos `()`, `Bool`, `Char`, `Int`, `Ordering`
  - os tipos tuplas cujos componentes são de tipos que são instâncias de `Bounded`

Classes básicas: Bounded (cont.)

- Exemplos:

```
minBound :: Bool                    ⇝ False
minBound :: Char                    ⇝ '\NUL'
minBound :: Int                     ⇝ -9223372036854775808
maxBound :: (Bool,Int,Ordering)     ⇝ (True,9223372036854775807,GT)
```

Classes básicas: Num

- **Pré-requisitos**: Eq, Show
- **Caracterização**: tipos numéricos
- **Métodos**:

```
(+)    :: Num a => a -> a -> a
(-)    :: Num a => a -> a -> a
(*)    :: Num a => a -> a -> a
negate :: Num a => a -> a
abs    :: Num a => a -> a
signum :: Num a => a -> a
```

- **São instâncias**:
  - os tipos básicos Int, Integer, Float, Double
- Observe que a classe Num não oferece um método para divisão.

Classes básicas: Num (cont.)

- Exemplos:

```
1 + 2 * 3       ⇝ 7
1.1 + 2.2       ⇝ 3.3000000000000003
negate 3.3      ⇝ -3.3
negate (-3.3)   ⇝ 3.3
abs 3           ⇝ 3
abs (-3)        ⇝ 3
signum 13       ⇝ 1
signum 0        ⇝ 0
signum (-13)    ⇝ -1
```

Classes básicas: `Real`

- **Pré-requisitos**: `Ord`, `Num`
- **Caracterização**: tipos cujos valores podem ser expressos como uma razão de dois números inteiros de precisão arbitrária
- **Métodos**

```
toRational :: Real a => a -> Rational
```

- **São instâncias**:
  - os tipos básicos `Int`, `Integer`, `Float`, `Double` e `Rational`
- `Real` é formada pelos tipos numéricos cujos valores podem ser comparados com (<) e demais operações relacionais da classe `Ord`.
- Nem todos os números podem ser comparados com (<), como por exemplo os números complexos.
- Todo número real de precisão finita pode ser expresso como um número racional.

Classes básicas: `Real` (cont.)

- Exemplos:

```
toRational 45      ⇝ 45 % 1
toRational (-4.5)  ⇝ (-9) % 2
toRational 6.7     ⇝ 7543529375845581 % 1125899906842624
```

Classes básicas: Integral

- **Pré-requisitos**: `Real`, `Enum`
- **Caracterização**: tipos inteiros
- **Métodos**:

```
div       :: Integral a => a -> a -> a
mod       :: Integral a => a -> a -> a
divMod    :: Integral a => a -> a -> (a, a)
toInteger :: Integral a => a -> Integer
```

- **São instâncias**:
  - os tipos básicos `Int` e `Integer`

Classes básicas: `Integral` (cont.)

- Exemplos:

```
div 7 2      ⇝ 3
mod 7 2      ⇝ 1
divMod 20 3  ⇝ (6,2)
17 `div` 3   ⇝ 5
17 `mod` 3   ⇝ 2
```

Classes básicas: `Integral` (cont.)

- A função

```
fromIntegral :: (Integral a, Num b) => a -> b
```

recebe um número integral e retorna este número convertido para um tipo numérico.

- Exemplo:

```
fromIntegral 17 :: Double                ⇝ 17.0
fromIntegral (length [1,2,3,4]) + 3.2    ⇝ 7.2
```

Classes básicas: `Fractional`

- **Pré-requisitos**: `Num`
- **Caracterização**: tipos numéricos não inteiros
- **Métodos**:

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

- **São instâncias**:
  - os tipos básicos `Float`, `Double` e `Rational`

Classes básicas: `Fractional` (cont.)

- Exemplos:

```
7.0 / 2.0 ⇝ 1.625
5.2 / 3.2 ⇝ 3.5
recip 2.0 ⇝ 0.5
```

Classes básicas: `Floating`

- **Pré-requisitos**: `Fractional`
- **Caracterização**: tipos em ponto flutuante

Classes básicas: Floating (cont.)

- **Métodos**:

```
pi      :: Floating a => a
exp     :: Floating a => a -> a
sqrt    :: Floating a => a -> a
log     :: Floating a => a -> a
(**)    :: Floating a => a -> a -> a
logBase :: Floating a => a -> a -> a
sin     :: Floating a => a -> a
tan     :: Floating a => a -> a
cos     :: Floating a => a -> a
asin    :: Floating a => a -> a
atan    :: Floating a => a -> a
acos    :: Floating a => a -> a
sinh    :: Floating a => a -> a
tanh    :: Floating a => a -> a
cosh    :: Floating a => a -> a
asinh   :: Floating a => a -> a
atanh   :: Floating a => a -> a
acosh   :: Floating a => a -> a
```

Classes básicas: `Floating` (cont.)

- **São instâncias**:
  - os tipos básicos `Float` e `Double`

Classes básicas: `Floating` (cont.)

- Exemplos:

```
pi::Float          ⇝ 3.1415927
pi::Double         ⇝ 3.141592653589793
5 ** 2.3           ⇝ 40.51641491731905
exp 1              ⇝ 2.718281828459045
sin (pi/2)         ⇝ 1.0
atan (-1)          ⇝ -0.7853981633974483
logBase 10 1000 ⇝ 2.9999999999999996
```

Classes básicas: RealFrac

- **Pré-requisitos**: `Real`, `Fractional`
- **Caracterização**: tipos reais fracionários
- **Métodos**

```
properFraction :: (RealFrac a, Integral b) => a -> (b,a)
truncate       :: (RealFrac a, Integral b) => a -> b
round          :: (RealFrac a, Integral b) => a -> b
ceiling        :: (RealFrac a, Integral b) => a -> b
floor          :: (RealFrac a, Integral b) => a -> b
```

- **São instâncias**:
  - os tipos básicos `Float`, `Double` e `Rational`

Classes básicas: `RealFrac` (cont.)

- Exemplos:

```
truncate 2.756          ⇝ 2
round 2.756             ⇝ 3
ceiling 2.756           ⇝ 3
floor 2.756             ⇝ 2
properFraction 2.756 ⇝ (2,0.755999999999998)
properFraction (9%4) ⇝ (2,1 % 4)
```

Sobrecarga de literais

Literais também podem ser sobrecarregados:

```
187        :: Num a => a
-5348      :: Num a => a
3.4        :: Fractional a => a
-56.78E13  :: Fractional a => a
```

Dicas e Sugestões

- Ao definir uma nova função em Haskell, é útil começar por escrever o seu tipo.
- Dentro de um **script**, é uma boa prática indicar o tipo de cada nova função definida.
- Ao indicar os tipos de funções polimórficas que usam números, igualdade, ou ordenações (ou outras resstrições), tome o cuidado de incluir as restrições de classe necessárias.

Exercícios

Exercício 1

Defina uma função chamada `exOr` para calcular o **ou exclusivo** de dois valores lógicos. Determine o tipo desta função.

Exercícios (cont.)

Exercício 2

Considere a seguinte definição de função:

```
mystery :: Integer -> Integer -> Integer -> Bool
mystery m n p = not (m == n && n == p)
```

1. Explique o que é calculado pela função.
2. Mostre a avaliação passo a passo da expressão

   ```
   mystery (2+4) 5 (11 `div` 2)
   ```

3. Escreva uma definição alternativa para esta função.

Exercícios (cont.)

Exercício 3

Defina a função

```
averageThree :: Integer -> Integer -> Integer -> Float
```

para calcular a média aritmética de três números inteiros.

Exercícios (cont.)

Exercício 4

Quais são os tipos dos valores a seguir?

a) `['a','b','c']`

b) `('a','b','c')`

c) `[(False,'0'),(True,'1')]`

d) `([False,True],['0','1'])`

e) `[tail,init,reverse]`

f) `[]`

g) `[[]]`

h) `[[10,20,30],[],[5,6],[24]]`

i) `(10e-2,20e-2,30e-3)`

j) `[(2,3),(4,5.6),(6,4.55)]`

k) `reverse`

l) `(["bom","dia","brasil"],sum,drop 7 "Velho mundo")`

m) `[sum,length]`

Exercícios (cont.)

Exercício 5

Determine o tipo de cada uma das funções definidas a seguir, e explique o elas calculam.

a) `second xs = head (tail xs)`

b) `swap (x,y) = (y,x)`

c) `pair x y = (x,y)`

d) `double x = x*2`

e) `palindrome xs = reverse xs == xs`

f) `twice f x = f (f x)`

Fim