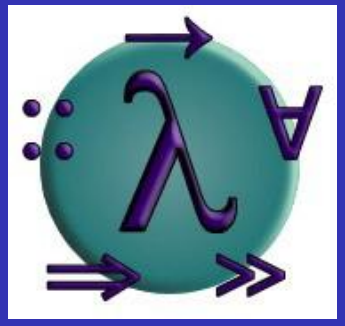
Programação Funcional



Capítulo 3

Tipos

José Romildo Malaquias

Departamento de Computação
Universidade Federal de Ouro Preto

2017.2

Tópicos

Tipo

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

Tópicos

Tipos numéricos inteiros

`Int`

- inteiros de precisão fixa
- limitado (tem um valor mínimo e um valor máximo)
- faixa determinada pelo tamanho da palavra
- Exemplos: `876`, `2012`

`Integer`

- inteiros de precisão arbitrária
- ilimitado
- menos eficiente que `Int`
- Exemplos: `10`, `754732483274878400304534523342312O`

Tipos numéricos fracionários

`Float`

- reais em ponto flutuante
- precisão simples
- Exemplos: `4.56`, `0.201E10`

`Double`

- reais em ponto flutuante
- precisão dupla
- Exemplos: `78643`, `987.3201E-60`

Tipos lógico e caracter

`Bool`

- valores lógicos: *verdadeiro* e *falso*
- Exemplos: `False`, `True`

`Char`

- caracteres unicode
- Exemplos: `'B'`, `'!'`, `'\n'`

Tipos lista

`[ t ]`

- listas: sequências de valores do mesmo tipo
- todos os elementos devem ser do mesmo tipo t
- o tamanho não é codificado no tipo
- Exemplos:

`['O','B','A']`	`[Char]`
`['B','A','N','A','N','A']`	`[Char]`
`[False,True,True]`	`[Bool]`
`[ [False,True], [], [True,False,True] ]`	`[[Bool]]`

`String`

- listas de caracteres
- é um sinônimo para `[Char]`
- Exemplos:

`"ufop"`	`['u','f','o','p']`
`"bom\ndia"`	`['b','o','m','\n','d','i','a']`
`""`	`[]`

Tipos tupla

$(t_1, \ldots, t_n)$

- tuplas: sequência de valores possivelmente de tipos diferentes
- $t_1, \ldots, t_n$ são os tipos dos componentes
- o tamanho é codificado no tipo
- `()` é a tupla vazia, do tipo `()`
- não existe tupla de um único componente
- Exemplos:

```
('A','t')
('A','t','o')
('A',True)
("Joel",'M',True,"COM")
(True,("Ana",'f'),['A','A','B'])
()
("nao e tupla")
```

```
(Char,Char)
(Char,Char,Char)
(Char,Bool)
(String,Char,Bool,String)
(Bool,(String,Char),[Char])
()
String
```

Tipos função

t_1 `->` `...` `->` t_n

- funções: mapeamentos definidos por um algoritmo
- $t_1, \ldots, t_{n-1}$ são os tipos dos argumentos
- t_n é o tipo do resultado
- Exemplos:

```
not       Bool -> Bool
(&&)      Bool -> Bool -> Bool
and       [Bool] -> Bool
words     String -> [String]
unwords   [String] -> String
```

Tópicos

Tipo de uma expressão

- Toda expressão tem um **tipo** associado.
- O tipo de uma expressão é o tipo do valor obtida pela avaliação da expressão.
- Exemplos:

expressão	valor da expressão	tipo da expressão
`True`	`True`	`Bool`
`'a'`	`'a'`	`Char`
`"ouro" ++ "preto"`	`"ouro preto"`	`String`
`not False`	`True`	`Bool`
`2*(5 - 8) >= 6 + 1`	`False`	`Bool`

Assinatura de tipo em expressões

- Qualquer expressão pode ter o seu tipo anotado.
- Se e é uma expressão e t é um tipo, então

  ```
  e :: t
  ```

  também é uma expressão.
- O tipo t especificado deve ser compatível com o tipo da expressão e.
- O tipo de `e::t` é t.
- `::` tem precedência menor do que todos os operadores de Haskell.

Exemplos de assinatura de tipo em expressões

```
True                  :: Bool
'a'                   :: Char
"maria das dores"     :: String
58                    :: Int
50 + 8                :: Double
2*(5 - 8) <= 6 + 1    :: Bool
```

```
abs (5+4::Double) * 8          -- 5+4 é do tipo Double
product [2::Integer,3,4,5]     -- 2 é do tipo Integer
```

```
7 :: Char          -- erro de tipo: 7 não pode ser do tipo Char
'F' :: Bool        -- erro de tipo: 'F' não pode ser do tipo Bool
not True :: Float  -- erro de tipo: (not True) não pode ser do
                   --                tipo Float
[4::Int,5::Float]  -- erro de tipo: Int e Float são incompatíveis
                   --                em uma lista
```

Consulta do tipo de uma expressão no GHCi

- No GHCi, o comando `:type` (ou de forma abreviada `:t`) calcula o tipo de uma expressão, sem avaliar a expressão.
- 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

Prelude> :t (lines,unlines)
(lines,unlines) :: (String -> [String], [String] -> String)

Prelude> :t 6::Float
6::Float :: Float
```

Assinatura de tipo em uma definição

- Ao fazer uma definição de variável ou função, o seu tipo pode ser anotado.
- Exemplos:

```haskell
alunos :: [String]
alunos = ["andré", "fábio", "paula", "rodrigo"]

notas :: [Double]
notas = [2.3, 6.9, 5.8, 9.5]

media :: [Double] -> Double
media ns = sum ns / length ns

resultado :: Double
resultado = media notas
```

Tópicos

Operação sobre vários tipos de dados

- 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`?

```
head :: [Char] -> Char
head :: [String] -> String
head :: [Bool] -> Bool
head :: [(String,Double)] -> (String,Double)
```

  `head` pode ter vários tipos.

Operação sobre vários tipos de dados

- 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`?

```
head :: [Char] -> Char
head :: [String] -> String
head :: [Bool] -> Bool
head :: [(String,Double)] -> (String,Double)
```

`head` pode ter vários tipos.

Operação sobre vários tipos de dados

- 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`?

```
head :: [Char] -> Char
head :: [String] -> String
head :: [Bool] -> Bool
head :: [(String,Double)] -> (String,Double)
```

`head` pode ter vários tipos.

Operação sobre vários tipos de dados

- 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`?

```
head :: [Char] -> Char
head :: [String] -> String
head :: [Bool] -> Bool
head :: [(String,Double)] -> (String,Double)
```

`head` pode ter vários tipos.

Operação sobre vários tipos de dados

- 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`?

```
head :: [Char] -> Char
head :: [String] -> String
head :: [Bool] -> Bool
head :: [(String,Double)] -> (String,Double)
```

`head` pode ter vários tipos.

Variáveis de tipo

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

Valor polimórfico

- Um valor é chamado **polimórfica** (*de muitas formas*) se o seu tipo contém uma ou mais variáveis de tipo.
- Exemplos:

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

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

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

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

```
fst :: (a,b) -> a
```

para quaisquer tipos `a` e `b`, `fst` recebe um par de valores do tipo `(a,b)` e retorna um valor do tipo `a`.

Instanciação de variáveis de tipo

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

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

expressão	valor	instanciação
`length [False,True]`	2	`a = Bool`
`length "54321"`	5	`a = Char`
`length ["ana","joel","mara"]`	3	`a = String`
`length [("ana",True)]`	1	`a = (String,Bool)`
`length [(&&),(||)]`	2	`a = Bool -> Bool -> Bool`

Funções polimórficas predefinidas

- Muitas das funções definidas no prelúdio são polimórficas.
- Exemplos:

```
fst  :: (a,b) -> a             -- seleciona o primeiro elemento de um par
snd  :: (a,b) -> b             -- seleciona o segundo elemento de um par
head :: [a] -> a               -- seleciona o primeiro el. de uma lista
tail :: [a] -> [a]             -- seleciona a cauda de uma lista
take :: Int -> [a] -> [a]      -- seleciona os primeiros el. de uma lista
zip  :: [a] -> [b] -> [(a,b)] -- combina duas listas, elemento a elemento
id   :: a -> a                 -- resulta no próprio argumento
```

- A lista vazia é polimórfica:

```
[] :: [a]
```

Tópicos

Classes de tipo

- Uma **classe de tipos** (ou simplesmente **classe)** é uma coleção de tipos para os quais é definido um conjunto de funções (aqui chamadas de **métodos**) que podem ter diferentes implementações.
- Uma classe especifica uma **interface** indicando o nome e a assinatura de tipo de cada função.
- Cada tipo que é **instância** (faz parte) da classe define (implementa) as funções especificadas pela classe.
- Desta maneira um mesmo nome de função pode estar associado a mais de uma implementação, caracterizando a **sobrecarga**, também chamada de **polimorfismo *ad hoc***.

Classes de tipo (cont.)

- Quando uma função sobrecarregada é usada, a escolha da implementação adequada baseia-se nos tipos dos argumentos ou do resultado da função.
- Classes de tipos podem ser parecidas com as classes das linguagens orientadas a objetos, mas elas são realmente muito diferentes. Elas são mais parecidas com interfaces.
- Haskell tem várias classes predefinidas.
- O programador pode definir suas próprias classes.

Exemplo de classe: `Eq`

- A classe predefinida **`Eq`** fornece uma interface para testar a igualdade de dois valores.
- **`Eq`** especifica as funções
  - `(==)`, que verifica se dois valores são iguais, e
  - `(/=)`, que verifica se dois valores são diferentes.
- Qualquer tipo em que o teste de igualdade entre dois valores desse tipo faz sentido pode ser uma instância de **`Eq`**.
- A maioria dos tipos predefinidos de Haskell são instâncias de **`Eq`**.
- Nenhum tipo função é instâncias de **`Eq`**.

Variáveis de tipo restritas

- Uma expressão de tipo pode conter um **contexto** que impõe restrições sobre algumas variáveis de tipo, restringindo-as a tipos que são instâncias das classes especificadas na restrição.
- Exemplos:

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

para qualquer tipo com igualdade `a`, `(==)` *recebe dois valores do tipo* `a` *e resulta em um valor do tipo* `Bool`.

```
elem :: Eq a => a -> [a] -> Bool
```

para qualquer tipo com igualdade `a`, `elem` *recebe um valor do tipo* `a` *e uma lista de valores do mesmo tipo* `a`, *e resulta em um valor do tipo* `Bool`.

Variáveis de tipo restritas (cont.)

- Uma **variável de tipo restrita** somente pode ser instanciada para tipos que satisfazem as restrições.
- Exemplos:

```
'A' == 'A'       ⇝ True              -- a = Char
"End" == "Fim"   ⇝ False             -- a = String
take == drop     ⇝ ERRO DE TIPO:
                     Int -> [a] -> [a] não é instância de Eq
```

Pré-requisitos

- Algumas vezes para que um tipo seja instância de uma classe, ele deve primeiro ser instância de outras classes (**pré-requisitos**).
- Motivo: alguns métodos da classe podem depender de métodos especificados nas outras classes.

Algumas classes predefinidas

classe	**instâncias**
Eq	tipos cujos valores podem ser comparados para igualdade
Ord	tipos com uma ordenação total de seus valores
Bounded	tipos com um valor mínimo e um valor máximo
Show	tipos cujos valores podem ser exibidos
Read	tipos cujos valores podem ser lidos
Enum	tipos cujos valores podem enumerados
Num	tipos numéricos
Integral	tipos numéricos inteiros
Fractional	tipos numéricos fracionários (exclui os inteiros)
Real	tipos numéricos reais
RealFrac	tipos numéricos reais fracionários (exclui os inteiros)
Floating	tipos numéricos com representação em ponto flutuante
RealFloating	tipos numéricos reais com representação em ponto flutuante

Classe `Eq`

- **Caracterização**: tipos com teste de igualdade
- **Métodos**:

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

- **Algumas 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

Classe **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
```

Classe `Ord`

- **Pré-requisitos**: `Eq`
- **Caracterização**: tipos com ordenação total de seus valores
- **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`.
- **Algumas 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`

Classe `Ord` (cont.)

  - 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).

Classe `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"
```

Classe `Show`

- **Caracterização**: tipos cujos valores podem ser representados como string
- **Alguns métodos**:

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

- **Algumas 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

Classe **Show** (cont.)

- Exemplos:

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

Classe `Read`

- **Caracterização**: tipos cujos valores podem ser obtidos de uma representação string
- **Alguns métodos**:

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

- `Read` e `Show` são duais: espera-se que

```
read . show = id
show . read = id
```

- **Algumas 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

Classe `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 de tipo.
- Se não for possível determinar o tipo do resultado de `read`, ocorre um erro de variável de tipo ambígua.

Classe `Enum`

- **Pré-requisitos**: `Ord`
- **Caracterização**: tipos cujos valores podem ser enumerados
- **Alguns 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]
```

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

Classe **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 6 (enumFrom 100)             ⇝ [100,101,102,103,104,105]
take 10 (enumFromThen 'A' 'C')    ⇝ "ACEGIKMOQS"
```

Classe **Bounded**

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

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

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

Classe **Bounded** (cont.)

- Exemplos:

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

Classe `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
fromInteger :: Num a => Integer -> a
```

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

Classe `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
fromInteger 99 :: Double     ⇝ 99.0
fromInteger 99 :: Rational   ⇝ 99 % 1
```

Classe `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
```

- **Algumas 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.

Classe `Real` (cont.)

- Exemplos:

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

Classe `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
```

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

Classe `Integral` (cont.)

- Exemplos:

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

Classe `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
```

- Observe que o argumento de `fromInteger` deve ser do tipo `Integer`, enquanto que o argumento de `fromIntegral` pode ser de qualquer tipo que seja instância da classe `Integral`.

Classe `Fractional`

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

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

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

Classe `Fractional` (cont.)

- Exemplos:

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

Classe `Floating`

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

Classe `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
```

Classe `Floating` (cont.)

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

Classe `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
```

Classe `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
```

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

Classe `RealFrac` (cont.)

- Exemplos:

```
truncate 2.756          ⇝ 2
round 2.756             ⇝ 3
ceiling 2.756           ⇝ 3
floor 2.756             ⇝ 2
properFraction 2.756 ⇝ (2,0.7559999999999998)
properFraction (9%4) ⇝ (2,1 % 4)
```

Sobrecarga de literais

- Algumas formas de literais são **sobrecarregadas**: um mesmo literal pode ser considerado de diferentes tipos.
- O tipo usado pode ser decidido pelo contexto ou por anotações de tipo.
- Se não for possível determinar o tipo, o compilador escolhe um tipo *dafault*.
- **Literais inteiros**:
  - podem ser de qualquer **tipo numérico** (como `Int`, `Integer`, `Float`, `Double` e `Rational`)
  - o tipo default é `Integer`
- **Literais em ponto flutuante**
  - Podem ser de qualquer **tipo numérico fracionário** (como `Float`, `Double`, `Rational` e `Complex Double`)
  - o tipo default é `Double`
- Exemplos:

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

Tópicos

Erros de tipo

- Toda expressão sintaticamente correta tem o seu tipo calculado em tempo de compilação.
- Se não for possível determinar o tipo de uma expressão ocorre um erro de tipo.
- A **aplicação** de uma função a um ou mais argumentos de *tipo inadequado* constitui um erro de tipo.
- Por exemplo:

```
Prelude> not 'A'

<interactive>:6:5:
    Couldn't match expected type 'Bool' with actual type 'Char'
    In the first argument of 'not', namely 'A'
    In the expression: not 'A'
    In an equation for 'it': it = not 'A'
```

Explicação:
A função `not` requer um valor booleano, porém foi aplicada ao argumento `'A'`, que é um caracter.

Checagem de tipos

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

Tópicos

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**.
- A capacidade de inferir tipos automaticamente facilita a programação, deixando o programador livre para omitir anotações de tipo ao mesmo tempo que permite a verificação de tipos.
- A inferência de tipo é feita usando as regras de tipagem de cada forma de expressão.

Lembretes sobre inferência de tipos

- O tipo mais geral dos literais inteiros é `Num a => a`
- O tipo mais geral dos literais em ponto flutuante é `Fractional a => a`
- Todos os elementos de uma lista devem ser do mesmo tipo
- Em uma aplicação de função
  - o tipo do argumento deve ser compatível com o domínio da função
  - o tipo do resultado deve ser compatível com o contra-domínio da função

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 restrições), tome o cuidado de incluir as restrições de classe necessárias.

Exercícios

Exercise 1

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

Exercise 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.)

Exercise 3

Defina a função

```
media3 :: Float -> Float -> Float -> Float
```

para calcular a média aritmética de três números.

Exercise 4

Defina a função

```
mediaPonderada :: (Double,Double) ->
                  (Double,Double) ->
                  (Double,Double) ->
                  Double
```

para calcular a média ponderada de três notas. Em cada argumento, o primeiro componente do par é a nota, e o segundo componente é o peso correspondente.

Exercícios (cont.)

Exercise 5

Defina a função

```
aumentoSalario :: Double -> Double -> Double
```

para calcular o valor do novo salário, sendo dados o valor do salário atual e a taxa percentual do aumento salarial.

Exercise 6

Defina uma função que recebe o salário base de um funcionário e resulta no salário a receber, sabendo-se que o funcionário tem gratificação de 5% sobre o salário base e paga imposto de 7% sobre este salário.
Use uma anotação de tipo para a função.

Exercícios (cont.)

Exercise 7

Defina uma função que recebe o valor de um depósito e o valor da taxa de juros, e resulta em um par contendo o valor do rendimento e o valor total incluindo o rendimento.
Use uma anotação de tipo para a função.

Exercise 8

Defina uma função que recebe a medida do raio r de um círculo e resulta na área A do círculo, dada por:

$$A = \pi \times r^2$$

Use uma anotação de tipo para a função.

Exercícios (cont.)

Exercise 9

Defina uma função que recebe a altura dos degraus de uma escada e a altura que o usuário deseja alcançar subindo a escada, e resulta na quantidade de degraus que ele deverá subir para atingir seu objetivo, sem se preocupar com a altura do usuário.
Faça uma anotação de tipo para a função.

Exercise 10

Determine o valor das expressões:

a) `show (show 42)`

b) `show 42 ++ show 42`

c) `show '\n'`

Exercícios (cont.)

Exercise 11

Verifique se as seguintes expressões são válidas e determine o seu tipo em caso afirmativo.

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

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

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

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

e) `[tail,init,reverse]`

f) `[[]]`

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

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

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

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

Exercícios (cont.)

k) `[sum,length]`

l) `div (read "45") 5`

Exercícios (cont.)

Exercise 12

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

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

b) `const x y = x`

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

d) `apply f x = f x`

e) `flip f x y = f y x`

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

g) `double x = x*2`

h) `palindrome xs = reverse xs == xs`

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

j) `aprovado nota = nota >= 6`

Exercícios (cont.)

k)

```haskell
mostra (nome,idade) = "Nome: " ++ nome ++ ", idade: " ++ show idade
```

l)

```haskell
mylog x b = log x / log b
```

m)

```haskell
f (x,y) = (show x, ['a' .. y])
```

Fim