Módulos

João Marcelo Uchôa de Alencar

Universidade Federal do Ceará - Quixadá

7 de Junho de 2017

Módulos

- Um módulo em Haskell é uma coleção de funções, tipos e classes.
- A modularização fomenta o reuso de *software*.
- O módulo *Prelude* é carregado por *default*.
- Sintaxe para carregamento: import Data.List
- Alternativa: :m + Data.List
- Se você carregou um *script* com :load que já tem um *import* para um módulo, não precisa carregar novamente.

Opções de Carregamento

```haskell
-- Carregar apenas funções específicas
import Data.List (nub, sort)

-- Carregar o módulo exceto uma função
import Data.List hiding (nub)

-- Exigir o nome do módulo antes da função
import qualified Data.Map

-- Renomeando o módulo
import qualified Data.Map as M
```

https://downloads.haskell.org/~ghc/latest/docs/html/libraries/

https://www.haskell.org/hoogle/

Data.List

- Módulo com funções para trabalhar com listas.
- O *Prelude* já importa algumas para você.

intersperse

```
ghci> intersperse '.' "DINHEIRO"
"D.I.N.H.E.I.R.O"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

intercalate

```
ghci> intercalate "  " ["bom","final","de", "semana"]
"bom  final  de  semana"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

transpose

```
ghci> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
transpose ["bom", "final", "de", "semana"]
["bfds","oiee","mnm","aa","ln","a"]
```

concat

```
ghci> concat ["bom","final","de", "semana"]
"bomfinaldesemana"
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

concatMap

```
ghci> concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]
```

and

```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

or

```
ghci> or $ map (==4) [2,3,4,5,6,1]
True
ghci> or $ map (>4) [1,2,3]
False
```

any e all

```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
ghci> all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
ghci> any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
```

iterate

```
ghci> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
ghci> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

splitAt

```
ghci> splitAt 3 "neymar"
("ney","mar")
ghci> splitAt 100 "neymar"
("neymar","")
ghci> splitAt (-3) "neymar"
("","neymar")
```

takeWhile

```
ghci> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
ghci> takeWhile (/=' ') "Bom final de semana."
"Bom"
ghci> sum $ takeWhile (<10000) $ map (^3) [1..]
53361
```

dropWhile

```
ghci> dropWhile (/=' ') "Bom final de semana"
" final de semana"
ghci> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
```

span e break

```
ghci> span (/=4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
break (==4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

sort

```
ghci> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
ghci> sort "Bom final de semana"
"   Baaadeefilmmnnos"
```

group

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
ghci> map (\l@(x:xs) -> (x,length l)) . group
      . sort $ [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
```

inits e tails

```
ghci>inits "quixada"
["","q","qu","qui","quix","quixa","quixad","quixada"]
ghci>tails "quixada"
["quixada","uixada","ixada","xada","ada","da","a",""]

search :: (Eq a) => [a] -> [a] -> Bool
search agulha palheiro =
   let nlen = length agulha
   in foldl (\acc x -> if take nlen x == agulha
      then True else acc) False (tails palheiro)
```

isInfixOf, isPrefixOf e isSuffixOf

```
ghci> "ufc" `isInfixOf` "a ufc em quixada"
True
ghci> "UFC" `isInfixOf` "a ufc em quixada"
False
ghci> "Ufc" `isInfixOf` "a Ufc em quixada"
False
ghci> "hey" `isPrefixOf` "hey ho!"
True
ghci> "hey" `isPrefixOf` "oh hey ho!"
False
ghci> "quixada" `isSuffixOf` "a ufc em quixada"
True
ghci> "quixada" `isSuffixOf` "a ufc em quixada."
False
```

partition

```
partition (`elem` ['A'..'Z']) "aqui é a UFC quixada."
("UFC","aqui \233 a  quixada.")
ghci> partition (>3) [1,3,5,6,3,2,1,0,3,7]
([5,6,7],[1,3,3,2,1,0,3])
```

find

```
ghci> find (>4) [1,2,3,4,5,6]
Just 5
ghci> find (>9) [1,2,3,4,5,6]
Nothing
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

elemIndex e elemIndices

```
ghci> :t elemIndex
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
ghci> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
ghci> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
ghci>' ' `elemIndices` "Onde estão os espaços?"
[4,10,13]
```

findIndex e findIndices

```
ghci> findIndex (==4) [5,3,2,1,6,4]
Just 5
ghci> findIndex (==7) [5,3,2,1,6,4]
Nothing
ghci>findIndices (`elem` ['A'..'Z'])
                        "Onde há Maiúsculas?"
[0,8]
```

zip ... zip7 e zipWith ... zipWith7

```
ghci> zipWith3 (\x y z -> x + y + z)
                    [1,2,3] [4,5,2,2] [2,2,3]
[7,9,8]
ghci> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
[(2,2,5,2),(3,2,5,2),(3,2,3,2)]
```

lines e unlines

```
ghci>lines "primeira\nsegunda\nterceira"
["primeira","segunda","terceira"]
ghci>unlines ["primeira","segunda","terceira"]
"primeira\nsegunda\nterceira\n"
```

words e unwords

```
ghci>words "Campus UFC Quixadá"
["Campus","UFC","Quixad\225"]
ghci>unwords ["Programação", "Funcional"]
"Programa\231\227o Funcional
```

nub e delete

```
ghci> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
[1,2,3,4]
ghci>nub "A vida continua"
"A vidacontu"
ghci>Data.List.delete 't' "tres pratos de trigo"
"res pratos de trigo"
ghci>Data.List.delete 't'
       . Data.List.delete 't' $ "tres pratos de trigo"
"res praos de trigo"
ghci>Data.List.delete 't'
       . Data.List.delete 't'
          . Data.List.delete 't' $ "tres pratos de trigo"
"res praos de rigo"
```

\\, union e intersect

```
ghci> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
ghci>"Sou um bom aluno" \\ "bom"
"Su u om aluno"
ghci> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
ghci>"Garantia" `union` "Ilimitada"
"GarantiaIlmd"
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

insert

```
ghci> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
ghci> insert 'g' $ ['a'..'f'] ++ ['h'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> insert 3 [1,2,4,3,2,1]
[1,2,3,4,3,2,1]
```

Data.Char

- São funções para lidar com caracteres, úteis na filtragem.
- Também existem vários predicados que ajudam na aplicação de funções como *takeWhile*.

isAlphaNum

```
ghci> all isAlphaNum "ufcquixada33"
True
ghci> all isAlphaNum "cuidado, cachorro bravo!"
False
```

isSpace

```
ghci> import Data.Function
ghci> words "bom dia jovens alunos"
["bom","dia","jovens","alunos"]
ghci> groupBy ((==) `on` isSpace) "bom dia jovens alunos"
["bom","dia","jovens","alunos"]
ghci> filter (not . any isSpace) .
      groupBy ((==) `on` isSpace) $ "bom dia jovens alunos"
["bom","dia","jovens","alunos"]
```

generalCategory

```
ghci> generalCategory ' '
Space
ghci> generalCategory 'A'
UppercaseLetter
ghci> generalCategory 'a'
LowercaseLetter
ghci> generalCategory '.'
OtherPunctuation
ghci> generalCategory '9'
DecimalNumber
ghci> map generalCategory " \t\nA9?|"
[Space,Control,Control,UppercaseLetter,
DecimalNumber,OtherPunctuation,MathSymbol]
```

Conversões entre Caracteres e Inteiros

```
ghci> map digitToInt "34538"
[3,4,5,3,8]
ghci> map digitToInt "FF85AB"
[15,15,8,5,10,11]
ghci> intToDigit 15
'f'
ghci> intToDigit 5
'5'
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```

Cifragem de Mensagens

```haskell
import Data.Char
cifrar :: Int -> String -> String
cifrar deslocamento mensagem = map chr deslocar
       where ords = map ord mensagem
       deslocar = map (+deslocamento) ords

decifrar :: Int -> String -> String
decifrar deslocamento mensagem =
       cifrar (negate deslocamento) mensagem
```

Data.Map

- Um *map* é semelhante a um dicionário em Python.
- Para uma *chave* retorna um *valor*.

Implementando um Mapeamento com Listas

```haskell
-- A maneira óbvia é uma lista de tuplas.
agenda = [("Joao","555-2938")
         ,("Jose","452-2928")
         ,("Maria","493-2928")
         ,("Luis","853-2492")]
-- Para achar uma Chave
findKey :: (Eq k) => k -> [(k,v)] -> v
findKey key xs = snd . head
                 . filter (\(k,v) -> key == k) $ xs
-- Versão "Foldada" com Maybe
findKey :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey key = foldr
              (\(k,v) acc -> if key == k
                             then Just v
                             else acc) Nothing
```

Criando Mapeamentos com Data.Map

```
-- Para evitar conflitos com Prelude
import qualified Data.Map as Map
-- Criando Mapeamentos
ghci> Map.fromList [("Joao","555-2938"),
                    ("Maria","452-2928"),
                    ("Jose","205-2928")]
fromList [("Joao","555-2938"),
          ("Maria","452-2928"),
          ("Jose","205-2928")]
ghci> Map.fromList [(1,2),(3,4),(3,2),(5,5)]
fromList [(1,2),(3,2),(5,5)]
```

Criando Mapeamentos com Data.Map

```
ghci> Map.empty
fromList []
ghci> Map.insert 3 100 Map.empty
fromList [(3,100)]
ghci> Map.insert 5 600
      (Map.insert 4 200 ( Map.insert 3 100  Map.empty))
fromList [(3,100),(4,200),(5,600)]
ghci> Map.insert 5 600 .
      Map.insert 4 200 . Map.insert 3 100 $ Map.empty
fromList [(3,100),(4,200),(5,600)]
```

null e *size*

```
ghci> Map.null Map.empty
True
ghci> Map.null $ Map.fromList [(2,3),(5,5)]
False
ghci> Map.size Map.empty
0
ghci> Map.size $ Map.fromList [(2,4),(3,3),(4,2),
                               (5,4),(6,4)]
5
```

lookup

```
ghci> let agenda = Data.Map.insert "Jose" "555-5556" $
                   Data.Map.insert "Maria" "555-5557" $
                   Data.Map.insert "Joao" "555-5558"
                   Data.Map.empty
ghci> Data.Map.lookup "Joao"  agenda
Just "555-5555"
ghci> Data.Map.lookup "Jonas"  agenda
Nothing
-- keys e elems retornam listas de chaves
-- e valores
ghci> Data.Map.keys agenda
["Joao","Jose","Maria"]
ghci> Data.Map.elems agenda
["555-5555","555-5557","555-5556"]
```

map e *filter*

```
-- No caso de mapeamentos, map e filter
-- aplicam automaticamente apenas nos valores
ghci> Map.map (*100) $ Map.fromList [(1,1),(2,4),(3,9)]
fromList [(1,100),(2,400),(3,900)]
ghci> Map.filter isUpper $
                  Map.fromList
                  [(1,'a'),(2,'A'),(3,'b'),(4,'B')]
fromList [(2,'A'),(4,'B')]
```

Lidando com Duplicatas

```
agenda = [("joao","555-2938")
    ,("joao","342-2492")
    ,("maria","452-2928")
    ,("jose","493-2928")
    ,("jose","943-2929")
    ,("jose","827-9162")
    ,("carlos","205-2928")
    ,("jonas","939-8282")
    ,("beatriz","853-2492")
    ,("beatriz","555-2111")]
-- O que acontece?
ghci> Data.Map.fromList agenda
```

Lidando com Duplicatas

```
-- Solução
ghci> let doisNumeros a b = a ++ "," ++ b
ghci> Data.Map.fromListWith (doisNumeros) agenda
ghci> Data.Map.insertWith (twoNumber) "joao" "555-5555" $
      Data.Map.fromListWith (twoNumber) agenda
```

Data.Set

- Imagine um *set* como o filho da lista com o *map*.
- Ao contrário da lista, são implementados internamentes com um árvore.
- São mais rápidos, porém perdem um pouco de flexibilidade.

Criando *sets*

```haskell
import qualified Data.Set as Set

texto1 = "A vida, como tudo na vida, acaba."
texto2 = "We are up all night to Get Lucky."
set1 = Set.fromList texto1
set2 = Set.fromList texto2
in = Set.intersection set1 set2
un = Set.unin set1 set2
di = Set.difference set1 set2
```

Eliminando Repetições

```
ghci> let a = Set.fromList [1,1,1,4,5,6,1,1,4,1,5]
ghci> a
fromList [1,4,5,6]
ghci> let b = Set.toList a
ghci> b
[1,4,5,6]
```

Criando seus Próprios Módulos

- Criar módulos permite reutilizar funções e tipos.
- Dizemos que um módulo exporta funções.
- Podemos definir funções que são internas ao módulo e não exportadas.

Módulo Geometria

```
module Geometria
( volumeEsfera,
  areaEsfera,
  volumeCubo,
  areaCubo,
  volumeOrtoedro,
  areaOrtoedro
) where
```

Importando o Módulo

- Para importar o módulo diretamente, teríamos que compilá-lo.
- Vamos ver isso nas próximas aulas.
- Por enquanto, temos que carregá-lo como qualquer outro arquivo.
- Mas ele já pode ser importado dentro de outros arquivos.

Importando o Módulo

```
GHCi, version 7.10.3
Prelude> :load Geometria.hs
[1 of 1] Compiling Geometria
Ok, modules loaded: Geometria.
*Geometria> Geometria.areaEsfera 5
314.15927
```

Dentro de um Arquivo

```
import Geometria

areas :: [Float] -> [Float]
areas l = map (Geometria.areaEsfera) l
```

Organizando Módulos em uma Hierarquia

- Vamos criar um diretório chamado Geometria.
- Dentro deles vamos criar os arquivos:
  - Esfera.hs
  - Ortoedro.hs
  - Cubo.hs

Esfera.hs

```haskell
module Geometria.Esfera
( volume,
  area
) where

volume :: Float -> Float
volume raio = (4.0 / 3.0) * pi * (raio ^ 3)

area :: Float -> Float
area raio = 4 * pi * (raio ^ 2)
```

Ortoedro.hs

```haskell
module Geometria.Ortoedro
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c

area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 +
             rectangleArea a c * 2 +
             rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

Cubo.hs

```haskell
module Geometria.Cubo
( volume
, area
) where

import qualified Geometria.Ortoedro as Ortoedro

volume :: Float -> Float
volume lado = Ortoedro.volume lado lado lado

area :: Float -> Float
area lado = Ortoedro.area lado lado lado
```

Exercício

Questão 14 da Lista 04!!!

- A função `duplicar :: String -> String` repete duas vezes cada vogal (letras 'a', 'e', 'i', 'o', 'u' minúsculas ou maiúsculas) numa cadeia de carateres; os outros carateres devem ficar inalterados.
- Exemplo: `duplicar "Ola, mundo!"== "OOlaa, muundoo!"`
- Dica: Crie uma lista com as vogais minúsculas e maiúsculas.
  1. Escreva uma definição recursiva para a função duplicar.
  2. Escreva uma definição usando o foldr para a função duplicarFold que faz o mesmo que duplicar.