



Lista, Tupla e Casamento de Padrões

Prof^a. Rachel Reis
rachel@inf.ufpr.br



Lista

- Coleção de elementos do mesmo tipo.
- Declaração da lista: o tipo dos elementos deve ser colocado entre colchetes.
 - Exemplo: `listanum :: [Int]`
- Definição dos elementos: devem ser colocados entre colchetes e separados por vírgulas.
 - `listanum = [1, 2, 3, 4, 5]` -- *lista de inteiros*
 - `listachar = ['a', 'b', 'c', 'd']` -- *lista de caracteres ("abcd")*
 - `listavazia = []` -- *lista sem nenhum elemento*



Lista

- Outras formas de inicializar uma lista:

- Listas de listas

```
listadelista = [[1,2],[3,4],[5,6]] --[[Int]]
```

- Listas preenchidas automaticamente, por meio de reconhecimento de padrões

```
lista1 = [1..10] -- [1,2,3,4,5,6,7,8,9,10]
```

```
lista2 = [1,3..10] -- [1,3,5,7,9]
```

```
lista3 = [10,8..0] -- [10,8,6,4,2,0]
```



Lista

- Exemplo 1: ambiente interativo GHCi

```
> lista4 = [1 .. 100]
```

```
> lista4
```

```
λ> lista = [1..100]
λ> lista
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100]
```



Lista

- Exemplo 2: ambiente interativo GHCi

```
> lista5 = [1, 3 .. 100]
```

```
> lista5
```

```
> lista = [1, 3 .. 100]
> lista
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,
41,43,45,47,49,51,53,55,57,59,61,63,65,67,69,71,73,75,77
,79,81,83,85,87,89,91,93,95,97,99]
>
```

- Atenção para a posição das vírgulas.



Lista

- Existem dois operadores básicos para se construir e manipular uma lista:

1) Operador interfixado “:” → usado para a construção de uma lista, elementos por elemento.

- Formato:

<elemento> : <lista>

O <elemento> à esquerda do **operador :** é adicionado na primeira posição da <lista> à direita



Lista

- Exemplo 1

> 8: [] -- lê-se, 8 constrói com uma lista vazia
[8]

- Exemplo 2

> 4: [6, 8] -- lê-se, 4 constrói com uma lista de 6 e 8
[4, 6, 8]

- Exemplo 3

> 6: 8: [] -- lê-se, 6 constrói uma lista com 8, que
[6, 8] constrói com uma lista vazia



Lista

- Existem dois operadores básicos para se construir e *manipular* uma lista:

2) Operador de concatenação “++” → usado para **concatenar** duas listas.

- Formato:

<lista> ++ <lista>



Lista

- Exemplo 1

```
> [1, 2, 3] ++ [4, 5, 6]  
[1, 2, 3, 4, 5, 6]
```

- Exemplo 2

```
> pares = [2, 4 .. 10]  
> impares = [1, 3 .. 11]  
> pares ++ impares  
[2, 4, 6, 8, 10 , 1, 3 , 5, 7 , 9, 11]
```



Lista

- No haskell, uma lista é dividida em cabeça (*head*) e cauda (*tail*).
 - cabeça (*head*): é o primeiro elemento da lista.
 - cauda (*tail*): todos os elementos da lista com exceção do primeiro.
- Funções do módulo Prelude para obter a cabeça e cauda:
 - > *head* [2, 4, 6, 8, 10] → 2
 - > *tail* [2, 4, 6, 8, 10] → [4, 6, 8, 10]



Lista – Função recursiva

- Lista e função recursiva, em geral, usa-se:
 - lista vazia
 - lista com cabeça (h) seguida de cauda (t)
- Para representar uma lista com cabeça e cauda utiliza-se o operador de construção “:”, ou seja:

[1, 2, 3] pode ser escrita como 1: [2, 3]

sendo: 1 → cabeça da lista

[2, 3] → cauda da lista



Função recursiva – Exemplo 1

- Escreva uma função recursiva em Haskell que calcule o comprimento de uma lista formada por números inteiros.



Função recursiva – Exemplo 1

```
-- calcula o comprimento de uma lista
comp :: [Int] -> Int
comp [] = 0
comp (h:t) = 1 + comp t
```

- Testando a função *comp* no ambiente interativo GHCi

```
> lista1 = [1 .. 20]
```

```
> comp lista1
```

```
20
```

```
> lista2 = [2, 4 .. 100]
```

```
> comp lista2
```

```
50
```



Função recursiva – Exemplo 2

- Escreva uma função recursiva em Haskell que receba uma lista de inteiros e devolva outra com os valores da primeira elevados ao cubo.

```
-- calcula o cubo de um valor
cubo :: Int -> Int
cubo x = x * x * x

-- calcula o cubo dos elementos da lista
aoCubo :: [Int] -> [Int]
aoCubo [] = []
aoCubo (h:t) = cubo h : aoCubo t
```

- Testando a função *aoCubo* no GHCi

```
> lista6 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
> aoCubo lista6
```

```
[1, 8, 27, 64, 125, 216, 343, 512, 729]
```



Para praticar em sala...

- Escreva uma função recursiva em Haskell que verifique se uma *string* possui o caractere informado (passado como parâmetro).


```
-- verifica se o caractere ch está na lista
possuiChar :: [Char] -> Char -> Bool
possuiChar [] ch = False
possuiChar (cabeca:cauda) ch
    | cabeca == ch = True
    | otherwise = possuiChar cauda ch
```

- Testando a função *possuiChar* no GHCi

```
> possuiChar "Joao" 'a'
True
```

```
> possuiChar "Joao" 'b'
False
```



Lista por Compreensão

- É uma forma de gerar listas de acordo com um critério, ou seja, é uma lista definida por um “gerador”:

```
numPares = [2*x | x <- [0..10]]
```

→ Lê-se: *numPares* é uma lista $2x$, tal que x representa cada um dos elementos de uma lista que vai de 0 a 10.



Lista por Compreensão

- É uma forma de gerar listas de acordo com um critério, ou seja, é uma lista definida por um “gerador”:

```
numPares = [2*x | x <- [0..10]]
```

- A lista numPares será gerada a partir da lista [0 .. 10].
- x representa cada um dos elementos da lista [0 .. 10].
- A nova lista numPares será formada pelo dobro de cada elemento **x** da lista [0 .. 10].



Lista por Compreensão

- Geração de lista infinita

```
listaInfinita = [2*x | x <- [0, 1..]]
```

→ Lê-se: *listaInfinita* é uma lista de $2x$, tal que x representa cada um dos elementos de uma lista que vai de 0 a infinito.

- Por causa da “**avaliação preguiçosa**”, a lista só realiza o cálculo até onde for necessário.



Lista por Compreensão

```
listaInfinita = [2*x | x <- [0, 1..]]
```

■ Exemplo

```
> head listaInfinita
```

```
0
```

```
> take 10 listaInfinita
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```



Lista por Compreensão

- É possível colocar **condições** no gerador de listas.

`[x | x <- [1..n], mod n x == 0]`

→ Lê-se: *gerar uma lista formada por elementos x, tal que x representa cada um dos elementos de uma lista que vai de 1 até n, onde o resto da divisão de n por x é zero.*



Lista por Compreensão

- É possível colocar condições no gerador de listas.

`[x | x <- [1..n], mod n x == 0]`

→ Indica que será criada uma nova lista com elementos x , onde:

- x é uma lista formada por elementos de 1 até n .
- e o resto da divisão de n por x tem que ser igual a zero.



Listas por Compreensão

- Exemplo, $n = 8$

```
> [x | x <- [1..8], mod 8 x == 0]
```

```
> [x | x <- [1,2,3,4,5,6,7,8], mod 8 x == 0]
```

```
> [x | x <- [1,2,3,4,5,6,7,8], mod 8 x == 0]
```

```
[1,2,4,8]
```




Exemplo

- Escreva uma função em haskell que retorne os 10 primeiros múltiplos de n . Utilize geradores de lista.



Exemplo

- Escreva uma função em haskell que retorne os 10 primeiros múltiplos de n. Utilize geradores de lista.

```
multiplos :: Int -> [Int]
multiplos n = [n*x | x <- [1 .. 10]]
```

- Testando no ambiente interativo GHCi

```
> multiplos 7
```

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```



Para praticar em casa

- Escreva uma função em Haskell que verifique se um determinado número é primo. Utilize geradores de lista.

→ Dica use a lista geradora abaixo:

```
[x | x <- [1..n], mod n x == 0]
```



Quicksort

- Para ilustrar o poder da lista por compreensão, vamos implementar em Haskell, o algoritmo de ordenação de vetores (ou listas) Quicksort.
- Passos do algoritmo
 - Escolhe um elemento do vetor (pivô)
 - Particiona o vetor de maneira que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores sejam maiores
 - Ordena recursivamente os vetores de elementos menores e maiores.



Quicksort

- **Caso base**, são os vetores de tamanho 0 ou 1, pois já se encontram ordenados.
- **Passo recursivo**: a cada chamada, um elemento é colocado na posição correta do vetor, e não será mais manipulado no passo seguinte.



Quicksort em Haskell

- Uso de lista por compreensão e concatenação (++)

```
-- Ordenação de listas Quicksort
qsort :: [Int] -> [Int]
qsort [] = []
qsort(h:t) = qsort[y | y <- t, y < h] -- < pivô
           ++[h] - o próprio pivô
           ++ qsort[y | y <- t, y >= h] -- > pivô
```



Tuplas em Haskell

- Coleção de valores que podem ou não ser de tipos diferentes.
- Os valores são colocados entre parênteses e separados por vírgulas.

("Maria", 22)

- A ordem dos elementos importa, de maneira que

$(27, \text{"José"}) \neq (\text{"José"}, 27)$



Tuplas em Haskell

- Quando a tupla possui apenas dois elementos, ela é chamada de “par” ou “2-upla”.
- Algumas funções do Prelude só podem ser usadas em tuplas com dois elementos:
 - **fst** : acessa o primeiro elemento de um par
 - > **fst** (“Maria”, 22) → “Maria”
 - **snd**: acessa o segundo elemento de um par
 - > **snd** (“Maria”, 22) → 22



Tuplas - Exemplo

- Vamos desenvolver um sistema acadêmico de notas para manipular informações sobre os alunos.
- Um aluno pode ser representado pela seguinte tupla:

```
(String, Int) -- nome, média das notas
```

Será que é possível melhorar
essa representação?



Tuplas - Exemplo

type: permite a criação de um “apelido” para um tipo já existente, não cria um tipo novo.

```
type NomeAluno = String
type MediaNota = Int
type Aluno = (NomeAluno, MediaNota)
type Turma = [Aluno]
```



Tuplas - Exemplo

- Escreva uma função que retorne uma lista com o nome dos alunos aprovados em uma turma.

```
type NomeAluno = String
type Nota = Int
type Aluno = (NomeAluno, Nota)
type Turma = [Aluno]
```

```
type NomeAluno = String
```

```
type Nota = Int
```

```
type Aluno = (NomeAluno, Nota)
```

```
type Turma = [Aluno]
```

```
aprovados :: Turma -> Int -> [NomeAluno]
```

```
aprovados tma nota = [nome | (nome, media) <- tma, media >= nota]
```

- Testando no ambiente interativo GHCi.
 - Crie um módulo e salve o código em um script.
 - Crie uma turma
 - > turma = [(“Joao”, 8), (“Pedro”, 6), (“Maria” , 9), (“Bia”, 5)]
 - Chame a função aprovados
 - > aprovados turma 7



Casamento de Padrões

- Para entender o casamento de padrões vamos ver um exemplo.

```
padrao :: Int -> String
```

```
padrao 1 = "um"
```

```
padrao 2 = "dois"
```

```
padrao 3 = "tres"
```

```
padrao x = "não está entre 1 e 3"
```

Casamento de Padrões

```
padrao :: Int -> String
padrao 1 = "um"
padrao 2 = "dois"
padrao 3 = "tres"
padrao x = "não está entre 1 e 3"
```

- Ao chamar a função padrão, a linguagem tenta definir em qual padrão ela se encaixa, testando de **cima para baixo** cada um dos padrões.
- O **primeiro** padrão encontrado, que casar com o valor passado como parâmetro, é executado.



Casamento de Padrões

- O que aconteceria se o programa fosse escrito da forma abaixo?

```
padrao :: Int -> String
padrao x = "não está entre 1 e 3"
padrao 1 = "um"
padrao 2 = "dois"
padrao 3 = "tres"
```

- O programa vai sempre executar o padrao x



Casamento de Padrões

- O que aconteceria se o programa fosse escrito da forma abaixo?

```
padrao :: Int -> String
padrao 1 = "um"
padrao 2 = "dois"
padrao 3 = "tres"
padrao _ = "não está entre 1 e 3"
```

- Como `x` não está sendo usado para nada, podemos substituí-lo por *underscore* (padrão curinga).



Casamento de Padrões

- Qual a finalidade da função abaixo?

```
padrao1 :: [Int] -> Int
padrao1 [] = 0
padrao1 (_:t) = 1 + padrao1 t
```

- A função calcula o comprimento de uma lista de inteiros.
- Qual a **saída** para a seguinte chamada?

```
padrao1 [x | x <- [1..100], mod x 2 == 0]
```



Casamento de Padrões

- Qual a finalidade da função abaixo?

```
type Tupla4 = (Int, Int, Int, Int)
padrao2 :: Tupla4 -> String
padrao2 (_,_,_,k)
    | k > 10 = "Maior que 10"
    | otherwise = "Não é maior que 10"
```

- A função verifica se o quarto elemento da tupla é > 10.
- Qual a **saída** para a seguinte chamada?

```
padrao2 (100,2,35,40)
```



Padrão curinga

- O padrão curinga (representado pelo caractere *underscore*) é usado para representar dados indefinidos que não estão sendo utilizados pelo programa.
- O uso do *underscore* facilita a implementação de funções que utilizam casamento de padrões.
- O uso do *underscore* é usado, muitas vezes como alternativa para estruturas if-then-else muito grandes.

- Exemplo usando guardas:

```
f1 :: Int -> Int -> Int -> Int
f1 x y z | x == 1 = 10
          | y == 2 = 20
          | z == 3 = 30
          | otherwise = 0
```

- Exemplo sem guardas:

```
f1 :: Int -> Int -> Int -> Int
f1 1 _ _ = 10
f1 _ 2 _ = 20
f1 _ _ 3 = 30
f1 _ _ _ = 0
```



Para praticar...

- Reescreva a função abaixo de maneira mais simples utilizando casamento de padrões e padrão curinga.

```
func :: (Int, (Int,Int)) -> Int
func z = if fst z == 1
         then fst (snd z) + snd (snd z)
         else if fst z == 2
              then fst (snd z) - snd (snd z)
         else 0
```