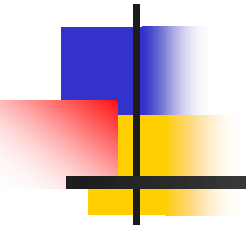
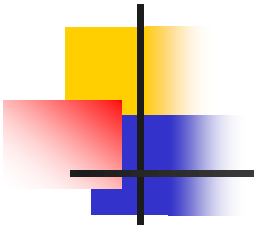


# Lista e Tupla

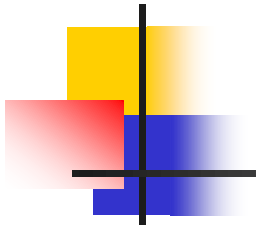
Profª. Rachel Reis

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

> lista = [1 .. 100]
> lista

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

# Lista

- Exemplo 2: ambiente interativo GHCi

> lista = [1, 3 .. 100]
> lista

```
> 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

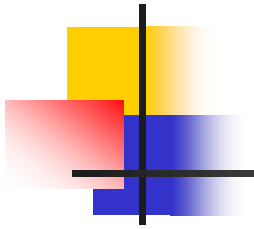
- Existem dois operadores básicos para se construir e manipular uma lista:

  1) Operador interfixado "**:**" → usado para a construção de uma lista, elemento por elemento.

     - Formato:

       <elemento> **:** <lista>

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



# Operador Interfixado

- Exemplo 1

  > 8: [ ]

  [8]

- Exemplo 2

  > 4: [6, 8]

  [4, 6, 8]

- Exemplo 3

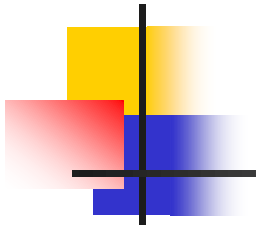
  > 6: 8: [ ]

  [6, 8]

# Lista

- Existem dois operadores básicos para se construir e *manipular* uma lista:
  2) Operador de concatenação "**++**" → usado para **concatenar** duas listas.
     - Formato:

       <lista> **++** <lista>



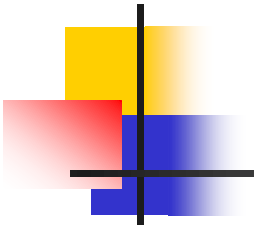
# Lista

- 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]
```

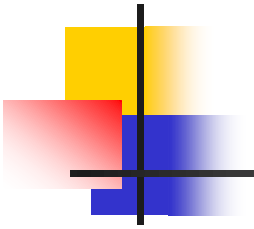


# Lista

- 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 cabeça e cauda:

  > *head* [2, 4, 6, 8, 10] → 2

  > *tail* [2, 4, 6, 8, 10] → [4, 6, 8, 10]

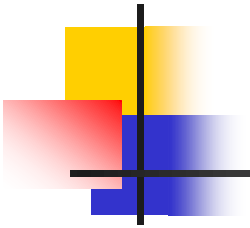


# Lista e Função recursiva

- 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]



# Função recursiva – Exemplo 1

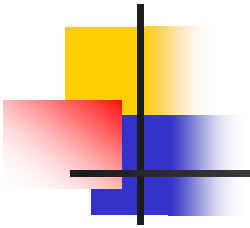
- Escreva uma função recursiva em Haskell que calcule o **comprimento** de uma lista formada por números inteiros.

# Solução – Exemplo 1

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

> lista1 = [1 .. 20]
> comp lista1
**20**

> lista2 = [2, 4 .. 100]
> comp lista2
**50**



# Função recursiva – Exemplo 2

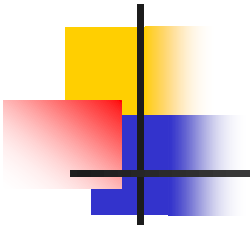
- Escreva uma função recursiva em Haskell que receba uma lista de inteiros e devolva outra com os valores da primeira elevados ao cubo.

# Solução – Exemplo 2

```
-- 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
```

> lista3 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
> aoCubo lista3
  [1, 8, 27, 64, 125, 216, 343, 512, 729]



# Para praticar em sala...

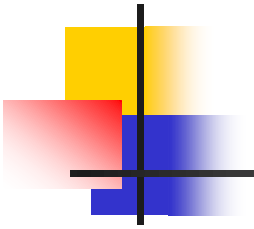
- Escreva uma função recursiva em Haskell que verifique se uma *string* possui o caractere informado (passado como parâmetro).

# Solução

```
-- 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
```

> possuiChar “Joao” ‘a’
True

> possuiChar “Joao” ‘b’
False

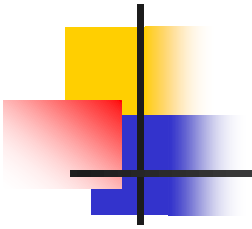


# Listas por Compreensão

- 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*.



# Listas 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].

# Listas por Compreensão

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

Quem são os elementos de numPares?

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

> numPares

[0,2,4,6,8,10,12,14,16,18,20]
```

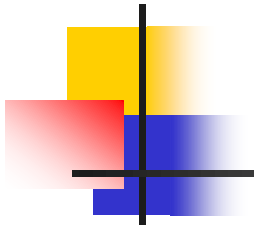
# Listas por Compreensão

- 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]
```

# Listas 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.*

# Listas por Compreensão

- É possível colocar condições no gerador de listas.

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

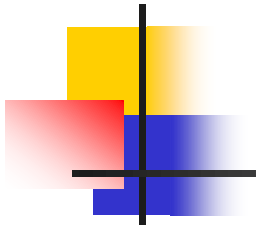
→ Indica que será criada uma nova lista com elementos x, onde:

  - 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]
```



# Listas por Compreensão

- Exemplo: escreva uma função em Haskell que retorne os **10 primeiros múltiplos de n**. Utilize geradores de lista.

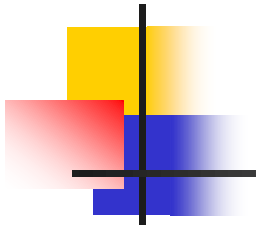
# Solução

```
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]

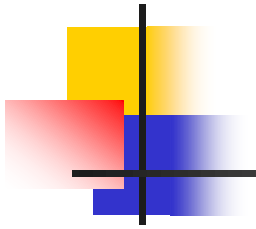


# Para praticar em casa

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

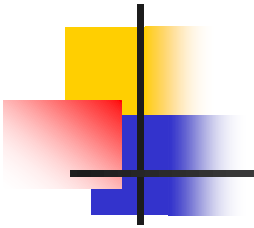
→ Dica use a lista geradora abaixo:

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



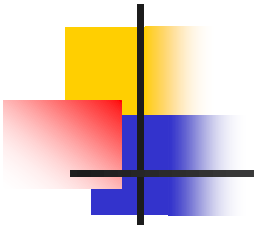
# Quicksort

- Para ilustrar o poder da lista por compreensão, vamos implementar em Haskell, o algoritmo de ordenação de vetores (ou listas) Quicksort.



# Quicksort

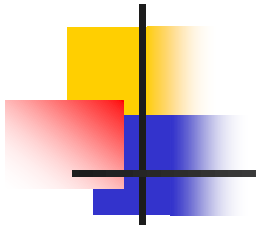
- Passos do algoritmo
  - Escolhe um elemento do vetor (pivô)
  - Particiona o vetor de maneira que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores sejam maiores
  - Ordena recursivamente os vetores com elementos menores e maiores.



# Quicksort

- **Caso base**, são os vetores de tamanho 0 ou 1, pois já se encontram ordenados.

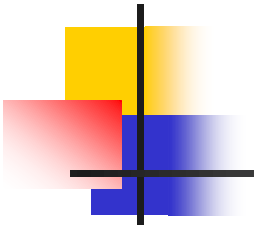
- **Passo recursivo**: a cada chamada, um elemento é colocado na posição correta do vetor, e não será mais manipulado no passo seguinte.



# 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

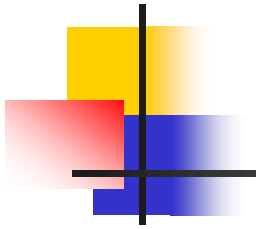
- Coleção de valores que podem ou não ser de tipos diferentes.
- Os valores são colocados entre parênteses e separados por vírgulas.

(“Maria”, 22)

- A ordem dos elementos importa, de maneira que

(27, “José”) ≠ (“José”, 27)

# Tuplas em Haskell

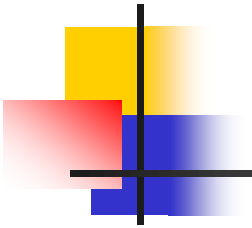


- Quando a tupla possui apenas dois elementos, ela é chamada de “par” ou “2-upla”.

- 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 1

- Escreva uma função recursiva que receba uma lista de números inteiros e retorne uma tupla contendo a quantidade de elementos pares e a quantidade de elementos ímpares presentes na lista.

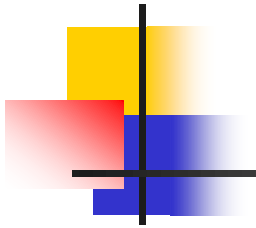
# Solução – Exemplo 1

```
-- contar a quantidade de pares e impares
contar :: [Int] -> (Int, Int)
contar [] = (0, 0)
contar (h:t)
      | mod h 2 == 0 = (pares + 1, impares)
      | otherwise = (pares, impares + 1)
      where
         (pares, impares) = contar t
```

> contar [1, 2, 3, 4, 5, 6, 7]

> (3, 4)

# Tuplas – Exemplo 2



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

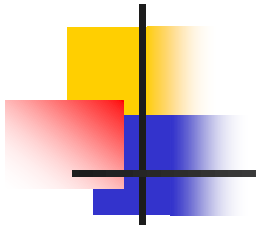
```
(String, Int) -- nome, média das notas
```

# Tuplas – Exemplo 2

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

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

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



# Tuplas – Exemplo 2

- Escreva uma função que, dada uma média de aprovação fornecida, retorne uma lista com o nome dos alunos **aprovados** em uma turma.

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

# Solução – Exemplo 2

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

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

- 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