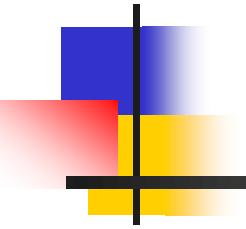
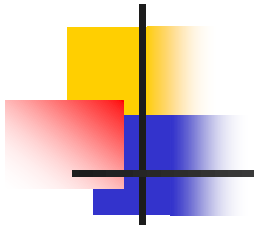


Casamento de Padrões

Profª. Rachel Reis

rachel@inf.ufpr.br



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"
```

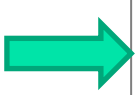
Casamento de Padrões

```
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]
```

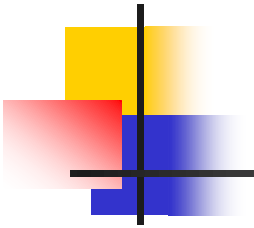
Casamento de Padrões

- 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 condicionais 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
```

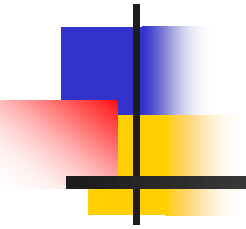
- Exemplo sem guardas:

```
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**.

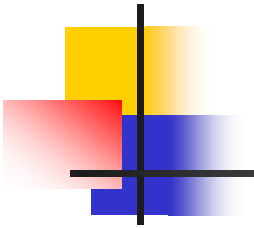
```
func :: (Int, (Int,Int)) -> Int
func z = if fst z == 1
         then fst (snd z) + snd (snd z)
         else if fst z == 2
            then fst (snd z) - snd (snd z)
            else 0
```



Funções de Alta Ordem

Profª. Rachel Reis

rachel@inf.ufpr.br



Funções de Alta Ordem

- Funções que podem receber outra função como argumento ou retornar uma função como resultado.
- Para ilustrar esse mecanismo, veremos duas funções utilizadas no processamento de listas:
  1) Mapeamento: recebe uma lista e aplica uma operação sobre todos os elementos daquela lista.
  2) Filtro: recebe uma função de teste e seleciona os elementos da lista que satisfazem a condição desejada.

Exemplo 1 - Função mapeamento

- Função recursiva que recebe 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
```

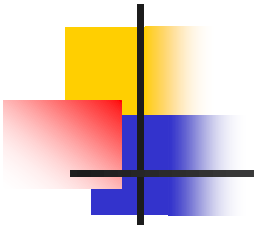
- A função aoCubo é uma função mapeamento.

Exemplo 2 - Função mapeamento

- Função recursiva que recebe uma lista de inteiros e devolva outra com os valores da primeira elevados ao quadrado.

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

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

  - A função aoQuadrado é uma função mapeamento.



Funções de Alta Ordem

- É possível implementar uma função de alta ordem que recebe uma função como argumento e aplica essa função em todos os elementos da lista, retornando uma lista de inteiros mapeada.
  - Nome da função: mapLista
  - Argumento:
    - função f do tipo (**`Int -> Int`**)
    - lista de inteiros

Exemplo 3

```
cubo :: Int -> Int
cubo x = x * x * x

quadrado :: Int -> Int
quadrado x = x * x

-- Função de Alta Ordem
mapLista :: (Int -> Int) -> [Int] -> [Int]
mapLista _ [] = []
mapLista f (h:t) = (f h) : (mapLista f t)
```

- mapLista cubo [1, 2, 3, 4]

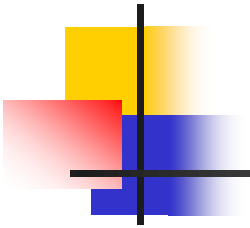
- mapLista quadrado [1, 2, 3, 4]

Exemplo 1 - Função filtro

```
par :: Int -> Bool
par x = (mod x 2 == 0)

impar :: Int -> Bool
impar x = (mod x 2 == 1)

-- Função de Alta Ordem
filtro :: (Int -> Bool) -> [Int] -> [Int]
filtro _ [] = []
filtro f (h:t) =
      | (f h) == True = h : (filtro f t)
      | otherwise = filtro f t
```

- filtro par [1, 2, 3, 4]

- filtro impar [1, 2, 3, 4]



Funções de Alta Ordem

- Existem funções próprias do Haskell para mapeamento e filtragem:
  - ***map***
  - ***filter***

Função Map

- Tipo:

```
map :: (a -> b)  -> [a] -> [b]
```

- Exemplos - Map

```
map (+7) [1, 2, 3]
map (True &&) [True, False]
```

Função Filter

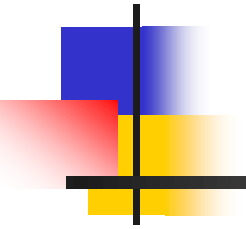
- Tipo:

```
filter :: (a -> Bool) -> [a] -> [a]
```

- Exemplos

```
filter isDigit "123-ab4"
```

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



Módulo Principal

Profª. Rachel Reis

rachel@inf.ufpr.br

Módulo Principal

- *script*: Main.hs

```
module Main where

main :: IO ()
main = do
    putStrLn "Hello world"
```

- *script*: Main.hs

```
module Main where
multiplica x y = x * y

main :: IO ()
main = do
    let z = multiplica 2  3
    print(z)
```

Módulo Principal

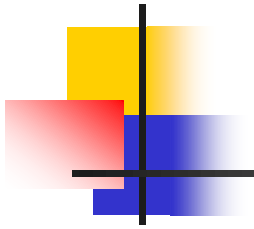
- O módulo principal pode importar outros módulos.

script: Operacoes.hs

```
module Operacoes where

multiplica  x y = x * y
```

script: Main.hs

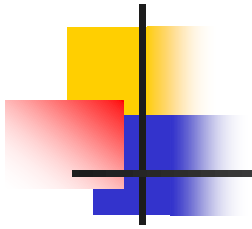
```
module Main where

import Operacoes

main :: IO ()
main = do
    let z = multiplica 2 3
    print (z)
```



Função – Saída de Dados

1) **putStrln**: exibe uma string na saída padrão, seguida de uma nova linha.

```
module Main where

main :: IO ()

main = do
    putStrLn "Hello world"
```



Função – Saída de Dados

2) **print**: converte um valor para uma string e exibe na saída padrão, seguida de uma nova linha.

```
module Main where

main :: IO ()

main = do
    let numero = 42
    print numero
```

Função – Saída de Dados

3) **putStr**: exibe uma string na saída padrão, sem adicionar uma nova linha no final.

```
module Main where

main :: IO ()

main = do
    putStr "Hello world"
```

Função – Entrada de Dados

1) **getLine**: lê a entrada digitada pelo usuário como uma string.

```
main :: IO ()
main = do
    putStrLn “Digite seu nome: ”
    nome ← getLine
    putStrLn (“Ola,  ” ++ nome ++ “!”)
```

Função – Entrada de Dados

2) **readLn**: lê a entrada digitada pelo usuário e converte automaticamente para o tipo especificado. Útil quando a entrada é de um tipo específico (ex.: Int, Double).

```
main :: IO ()
main = do
    putStrLn “Digite um numero: ”
    numero ← readLn :: IO Int
    print (numero)
```

Função – Entrada de Dados

3) **getChar**: lê um caractere digitado pelo usuário. Útil quando os caracteres precisam ser lidos individualmente.

```
main :: IO ()
main = do
    putStrLn “Digite uma letra: ”
    letra ← getChar
    print (letra)
```