

Paradigmas de Programação

# Linguagem Haskell

Prof<sup>a</sup> Andréa Schwertner Charão  
DLSC/CT/UFSM

# C versus Haskell: Quicksort

**C:**  
**Linguagem**  
**Imperativa**

## Quicksort em C

```
// To sort array a[] of size n: qsort(a,0,n-1)
void qsort(int a[], int lo, int hi)
{
    int h, l, p, t;

    if (lo < hi) {
        l = lo;
        h = hi;
        p = a[hi];

        do {
            while ((l < h) && (a[l] <= p))
                l = l+1;
            while ((h > l) && (a[h] >= p))
                h = h-1;
            if (l < h) {
                t = a[l];
                a[l] = a[h];
                a[h] = t;
            }
        } while (l < h);


        a[hi] = a[l];
        a[l] = p;

        qsort( a, lo, l-1 );
        qsort( a, l+1, hi );
    }
}
```

# C versus Haskell: Quicksort

## Quicksort em Haskell

```
qsort []      = []  
qsort (x:xs) = qsort (filter (< x) xs) ++ [x] ++ qsort (filter (>= x) xs)
```



Haskell:  
Linguagem  
**Declarativa**

Fonte: [http://www.haskell.org/haskellwiki/Introduction#Quicksort\\_in\\_Haskell](http://www.haskell.org/haskellwiki/Introduction#Quicksort_in_Haskell)

# Condicionais: if-then-else

```
doubleSmall :: Int -> Int
doubleSmall x = if x > 100
    then x
    else x*2
```

Atenção!  
Indent obrigatório!

```
> doubleSmall 2
4
> doubleSmall 200
200
```

# Condicionais: guardas

Atenção!  
Indent obrigatório!

```
doubleSmall :: Int -> Int
doubleSmall x
    | x > 100      = x
    | otherwise    = x*2
```

Guardas geralmente deixam o código mais limpo do que com if-then-else.

# Condicionais: guardas

`<FunctionName> <Parameters>`

```
| <Test1>      = <Exp1>
| <Test2>      = <Exp2>
| ...          ...
| <Testn>      = <Expn>
| otherwise    = <Exp>
```

Indent  
obrigatório

otherwise é  
opcional

# Funções recursivas: fatorial

Com  
if-then-else:

```
fatorial :: Int -> Int
fatorial n = if n > 0
              then n * fatorial (n-1)
              else 1
```

Com  
guardas:

```
fatorial :: Int -> Int
fatorial n
  | n > 0      = n * fatorial (n-1)
  | n == 0    = 1
```

# Pattern matching

- Casamento de padrões
- Testes de condições ficam **implícitos**
- Legibilidade de código

```
fatorial :: Int -> Int
fatorial n = if n > 0
  then n * fatorial (n-1)
  else 1
```

Formas  
(quase)  
equivalentes

```
fatorial :: Int -> Int
fatorial 0 = 1
fatorial n = n * fatorial (n - 1)
```



# Listas: operador ':'

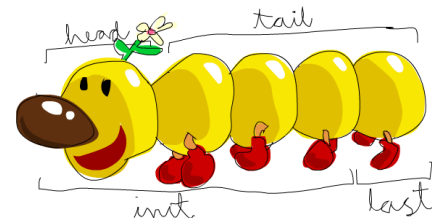
- **Construção** de listas: em LISP, Scheme, etc. há uma função chamada "cons"
- Em Haskell se usa o operador ':'. Ex.:

Operação: `1:[]`

Resultado: `[1]`

- Forma geral: *elem : lista*

Ou seja: constrói lista com *head=elem* e *tail=lista*



# Listas: operador ':'

```
> 1:[ ]
```

```
[1]
```

```
> 1:[2]
```

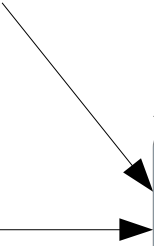
```
[1,2]
```

```
> 1:(2:[ ])
```

```
[1,2]
```

```
> 1:2:[ ]
```

```
[1,2]
```



Formas  
equivalentes

```
> 'a': "bc"
```

```
"abc"
```

```
> 'a':'b':[ ]
```

```
"ab"
```

```
> "ab": "cd"
```

```
ERRO!
```

```
> "ab": [ "cd" ]
```

```
[ "ab", "cd" ]
```

# Listas: operador ':'

- Função para colocar maiúscula no início da palavra

```
import Char  
  
inicialMaiusc :: String -> String  
inicialMaiusc x = toUpper (head x) : tail x
```

```
> inicialMaiusc "andrea"  
"Andrea"  
  
> inicialMaiusc ""
```

**ERRO!**

# Listas: operador ':'

- Função para colocar maiúscula no início da palavra - **corrigida com condicional**

```
import Char

inicialMaiusc :: String -> String
inicialMaiusc x = if x == ""
    then ""
    else toUpper (head x) : tail x
```

```
> inicialMaiusc ""
""
```

## Listas: operador ':'

- Função para colocar maiúscula no início da palavra - **corrigida usando pattern matching**

```
import Char

inicialMaiusc :: String -> String
inicialMaiusc "" = ""
inicialMaiusc x = toUpper (head x) : tail x
```

# Listas: operador ':'

- Função para colocar maiúscula no início da palavra - **mais pattern matching!**

```
import Char

inicialMaiusc :: String -> String
inicialMaiusc "" = ""
inicialMaiusc (x:xs) = toUpper x : xs
```

Padrão que equivale a uma lista com head x e tail xs

# Definindo funções com listas

- Função: somatório de elementos de uma lista

```
somaElem :: [Int] -> Int
```

```
somaElem [] = 0
```

```
somaElem lis = head lis + somaElem (tail lis)
```

```
> somaElem [1,2,3]
```

```
= 1 + somaElem [2,3]
```

```
= 1 + (2 + somaElem [3])
```

```
= 1 + (2 + (3 + somaElem []))
```

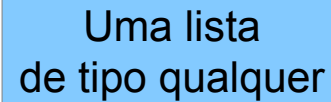
```
= 1 + (2 + (3 + 0))
```

```
= 6
```

# Definindo funções com listas

- Função: tamanho de uma lista

```
tamanho :: [a] -> Int  
-- complete-me!
```



Uma lista  
de tipo qualquer



# Definindo funções com listas

- Função: tamanho de uma lista

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

```
tamanho [] = 0
```

```
tamanho lis = 1 + tamanho (tail lis)
```

# Definindo funções com listas

- Função: gera uma string com n repetições de um caracter c

```
> repete 0 'a'
""
> repete 1 'a'
"a"
> repete 4 'a'
"aaaa"
```

# Definindo funções com listas

- Função: gera uma string com n repetições de um caracter c

```
repete :: Int -> Char -> [Char]  
repete 0 c = []  
repete n c = c : repete (n-1) c
```

# Definindo funções com listas

- Função: gera uma lista com as potências de 2, com expoente de **n** até **0** ( $2^n, 2^{(n-1)}, 2^{(n-2)}, \dots, 2^0$ )

```
> gerapot2 0
[1]
> gerapot2 1
[2, 1]
> gerapot2 2
[4, 2, 1]
> gerapot2 8
[256, 128, 64, 32, 16, 8, 4, 2, 1]
```

# Definindo funções com listas

- Função: gera uma lista com as potências de 2, com expoente de **n** até **0** ( $2^n, 2^{(n-1)}, 2^{(n-2)}, \dots, 2^0$ )

```
gerapot2 :: Int -> [Int]
gerapot2 0 = [1]
gerapot2 n = 2^n : gerapot2 (n-1)
```

# Definindo funções com listas

- Função geraPotencias: agora com expoente de 0 até n

```
> gerapot2 ' 0  
[1]  
> gerapot2 ' 1  
[1,2]  
> gerapot2 ' 2  
[1,2,4]  
> gerapot2 ' 8  
[1,2,4,8,16,32,64,128,256]
```

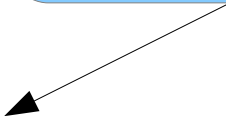
# Definindo funções com listas

- Função geraPotencias: agora com expoente de 0 até n

```
gerapot2' :: Int -> [Int]  
gerapot2' n = aux 0 n
```

```
aux :: Int -> Int -> [Int]  
aux e n = if (e <= n)  
           then 2^e : aux (e+1) n  
           else []
```

Função auxiliar recursiva  
com 2 argumentos:  
um que varia (e),  
outro fixo (n)



# Tupla

- É um tipo composto, delimitado por parênteses
- Pode conter dados heterogêneos
- Exemplos:
  - Definição do tipo: (Int,Int) (Int, Char) (Int, Char, Float)
  - Dado: (9,8) (0,'a') (0, 'a', 9.0)
- Funções que manipulam tuplas de 2 elementos:
  - fst (retorna primeiro elemento)
  - snd (retorna segundo elemento)
- Exemplos:

```
> fst ('a',2)
'a'
> snd (1,2)
2
> fst (1,2,3)  ERRO!
```



# Definindo funções com tuplas

- Função: retorna uma tupla com um número e seu quadrado

```
quadradoTupla :: Int -> (Int,Int)
```

```
quadradoTupla n = (n, n^2)
```

```
> quadradoTupla 8  
(8,64)
```

# Definindo funções com tuplas e listas

- Função: retorna uma tabela de números de **n** a **1** e seus quadrados

```
geraTabela :: Int -> [ (Int, Int) ]  
geraTabela 0 = []  
geraTabela n = (n, n^2) : geraTabela (n-1)
```

```
> geraTabela 5  
[(5,25),(4,16),(3,9),(2,4),(1,1)]
```

# Exercícios

1) Defina uma função **recursiva** que receba uma lista de números inteiros e produza uma lista com cada número elevado ao quadrado, conforme o exemplo abaixo

```
> eleva2 [1,2,3,4,5]  
[1,4,9,16,25]
```

# Exercícios

2) Defina uma função **recursiva** que verifique se um dado character está contido numa string, conforme os exemplos abaixo

```
> contido 'e' "andrea"
True
> contido 'x' "andrea"
False
> contido 'a' ""
False
```

# Exercícios

3) Defina uma função **recursiva** que receba uma string e retire suas vogais, conforme os exemplos abaixo

```
> semVogais "andrea"  
"ndr"  
> semVogais "xyz"  
"xyz"  
> semVogais "ae"  
""
```

# Exercícios

4) Defina uma função **recursiva** que receba uma lista de coordenadas de pontos 2D e desloque esses pontos em 2 unidades, conforme o exemplo abaixo

```
> translate [(0.1,0.2), (1.1,6), (2,3.1)]  
[(2.1,2.2),(3.1,8.0),(4.0,5.1)]
```

# Exercícios

5) Defina uma função que receba um número  $n$  e retorne uma tabela de números de **1** a **n** e seus quadrados, conforme o exemplo abaixo

```
> geraTabela' 5  
[(1,1),(2,4),(3,9),(4,16),(5,25)]
```