

PROGRAMAÇÃO FUNCIONAL COM HASKELL

Gustavo Santos

`gfdsantos@inf.ufpel.edu.br`

ÍNDICE

1. Paradigma de Programação Funcional
2. O Básico
3. Listas
4. Lambda, Currying e Composições
5. IO
6. Compilando Programas

PARADIGMA DE PROGRAMAÇÃO FUNCIONAL

1930



Alonzo Church apresenta o Cálculo Lambda.

1958

LISP

~~List of Stupid Parentheses~~
LISt Processing

LISP

```
(defun factorial (n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

1988

Requisitos

- Ser viável para o ensino, pesquisa e aplicações, como sistemas de larga escala
- Ter a sintaxe e semântica completamente descritiva
- Ser livre e de código aberto
- Basear-se em ideias que envolvessem o senso comum;
- Formar um padrão para o desenvolvimento de linguagens funcionais

1990

HASKELL

Versão 1.0

Programação Funcional

Alto nível de abstração.

Programação Funcional

Paradigma de programação focado em avaliação de funções.

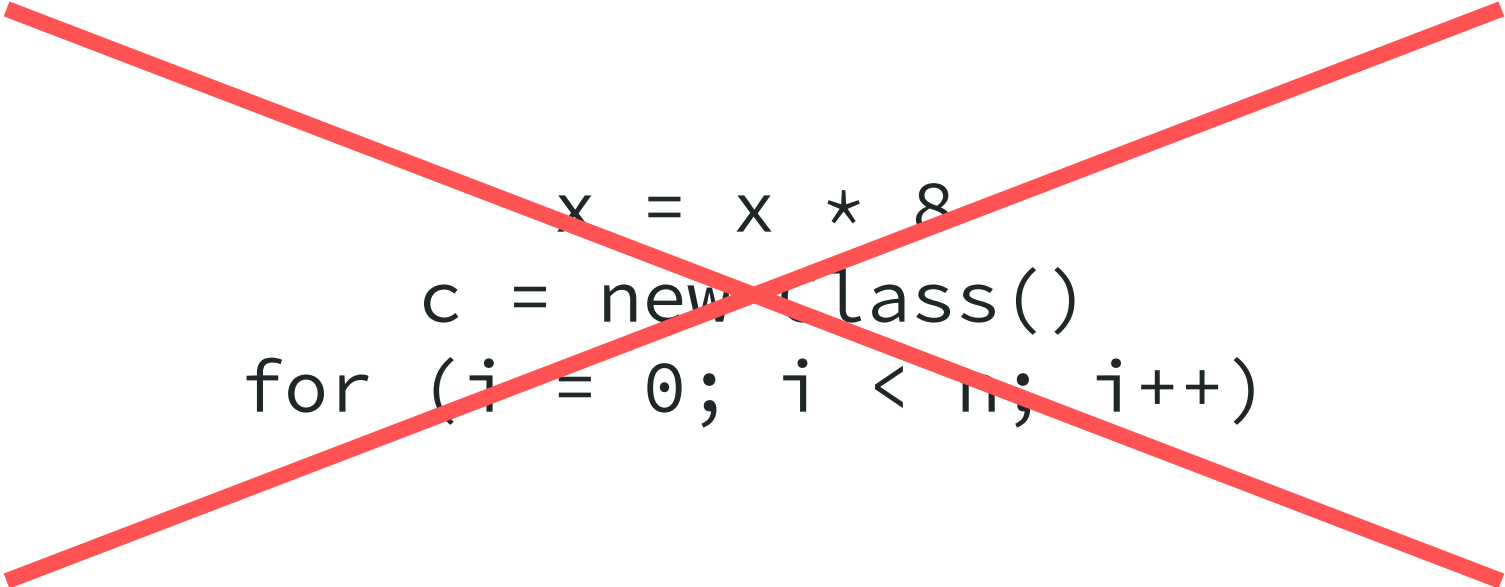
Programação Funcional

Programas elegantes e concisos.

Programação Funcional

```
x = x * 8  
c = new Class()  
for (i = 0; i < n; i++)
```

Programação Funcional



```
x = x * 8  
c = new class()  
for (i = 0; i < n; i++)
```

Programação Funcional

Facilidade em paralelizar programas!

Cinto de utilidades

- Recursão
- Estrutura de Decisão
- Estruturas de Dados Complexos
- Polimorfismo
- Funções de Alta Ordem
- Avaliação Preguiçosa
- Monadas

Linguagens Funcionais

Puras

- Haskell
- Miranda

Impuras

- Erlang
- LISP
- Scala
- D
- C#
- F#
- Python
- Ruby

O BÁSICO

Conhecendo o Ambiente

ghc: Compilador

ghci: Interpretador

gsantos ~ ghci

Prelude

Não gostou?

```
Prelude> :set prompt "haskell: "
```

Prelude

Precisa de ajuda?

`Prelude> :?`

Prelude

Quer sair?

```
Prelude> :quit
```

Prelude

Carregar um 'script':

```
Prelude> :load <nome>.hs
```

Operadores Aritméticos

Soma: +

Subtração: -

Multiplicação: *

Divisão: /

Potência: ^

Exponenciação: **

Operadores Lógicos

E:	&&
Ou:	
Negação:	not
Igualdade (comparador):	==
Diferença (comparador):	/=
Maior (comparador):	>
Maior ou igual (comp):	>=
Menor (comparador):	<
Menor ou igual (comp):	<=

Construindo Definições

let

Construindo Definições

```
Prelude> let cinco = 5
```


Funções Básicas (succ)

```
Prelude> succ 9  
10
```

Funções Básicas (min)

```
Prelude> min 3 9  
3
```

Funções Básicas (max)

```
Prelude> max 3 9  
9
```

Funções Básicas (min)

```
Prelude> 5 `min` 7  
5
```

Definindo Funções

```
Prelude> let um = 1
```

Definindo Funções

```
Prelude> let maisUm n = n + 1
```

Definindo Funções

```
Prelude> let ehPar x = mod x 2 == 0
```

Definindo Funções

```
Prelude> let t (x1, x2) = x1 * 2 - x2
```


Faça você mesmo!

Defina uma função que calcule a distância entre dois pontos. Dica: A função recebe duas tuplas, cada uma representando um ponto. (Raiz quadrada: sqrt)

Faça você mesmo!

```
let dist (x1, y1) (x2, y2) =  
    sqrt ( (x1 - x2)^2 + (y1 - y2)^2 )
```

Criando um Script

<nome>.hs

```
1  maisUm x = x + 1
2
3  soma x y = x + y
4
5  dist (x1, y1) (x2, y2) = sqrt ( (x1 - x2)^2 + (y1 - y2)^2 )
6  |
```

Carregando um Script para o GHCi

```
Prelude> :load <nome>.hs
```

Casamento de Padrões

diga 0 = "zero!"

diga 1 = "um!"

diga 2 = "dois!"

diga 5 = "cinco!"

diga 9 = "nove!"

diga _ = "Nao conheco este numero!"

Casamento de Padrões: Guards!

```
sorte 7 = "Sortudo!"
```

```
sorte n
```

```
  | n < 7 = "Uh, tente um numero maior"
```

```
  | n > 7 = "Uh, tente um numero menor"
```

Recursão

fatorial 0 = 1

fatorial n | n > 1 = n * fatorial (n - 1)

Faça você mesmo!

Defina uma função recursiva que calcule a multiplicação de dois números utilizando somas sucessivas.

Faça você mesmo!

$$\text{mult } n \ 0 = 0$$

$$\text{mult } 0 \ m = 0$$

$$\text{mult } n \ m = n + \text{mult } n \ (m-1)$$

LISTAS

Criando uma Lista

```
[1, 2, 3, 4, 5]  
['a', 'e', 'i', 'o', 'u']
```

Criando um Range

```
[limite inferior .. limite superior]
```

Criando um Range

`[0..9]`

Criando um Range

`[primeiro, segundo .. lim. superior]`

Criando um Range

```
[0, 2 .. 10]
```


Funções sobre listas: concatenador

```
Prelude> [0..5] ++ [32..36]  
[0,1,2,3,4,5,32,33,34,35,36]
```

Funções sobre listas: concatenador

```
Prelude> "haskell " ++ "eh " ++ "legal!"  
"haskell eh legal!"
```

Funções sobre listas: cons

```
Prelude> 0 : [1, 2, 3, 4, 5]  
[0,1,2,3,4,5]
```

Funções sobre listas: índice

```
Prelude> ["uva", "carro", "ok"] !! 2  
"ok"
```

Funções sobre listas: tamanho

```
Prelude> length ["uva", "carro", "ok"]  
3
```

Funções sobre listas: inversão

```
Prelude> reverse ['a' .. 'g']  
"gfedcba"
```

Funções sobre listas: head

```
Prelude> head [1, 2, 3, 4, 5]  
1
```

Funções sobre listas: tail

```
Prelude> tail [1, 2, 3, 4, 5]  
[2, 3, 4, 5]
```


Funções sobre listas: last

```
Prelude> last [1, 2, 3, 4, 5]  
5
```

Funções sobre listas: init

```
Prelude> init [1, 2, 3, 4, 5]  
[1,2,3,4]
```

Funções sobre listas: take

```
Prelude> take 3 [0..10]  
[0,1,2]
```

Funções sobre listas: drop

```
Prelude> drop 3 [0..10]  
[3,4,5,6,7,8,9,10]
```

Funções sobre listas: sum

```
Prelude> sum [0..10]  
55
```

Funções sobre listas: product

```
Prelude> product [1..10]  
3628800
```

Funções sobre listas: elem

```
Prelude> elem 55 [0..10]  
False
```

Compreensão de listas

```
Prelude> [x^2 | x <- [0..10]]  
[0,1,4,9,16,25,36,49,64,81,100]
```


Compreensão de listas

```
Prelude> [x^2 | x<- [0..10], mod x 2 == 0]  
[0,4,16,36,64,100]
```

A função FILTER

```
Prelude> filter ehPar [0..10]  
[0,2,4,6,8,10]
```

A função MAP

```
Prelude> map quad [0..10]  
[0,1,4,9,16,25,36,49,64,81,100]
```

A função FOLD

```
Prelude> let soma x y = x + y
```

```
Prelude> foldr soma 0 [0..10]
```

```
55
```

Recursão em listas

```
Prelude> map quad [0..10]  
[0,1,4,9,16,25,36,49,64,81,100]
```

Recursão em listas

```
vogais = "aeiou"  
removeV [] = []  
removeV (h:t) =  
    if elem h vogais then  
        removeV t  
    else  
        h : removeV t
```

Faça você mesmo!

Defina uma função recursiva que receba uma lista de números inteiros e retorne uma lista com todos estes elementos elevados ao cubo.

Faça você mesmo!

```
calcCubo [] = []
```

```
calcCubo (h:t) = h*h*h : calcCubo t
```


LAMBDA, CURRYING E COMPOSIÇÕES

Expressão Lambda

$\lambda n \rightarrow 2 * n$

Expressão Lambda

\n m -> 2 * n + m

Expressão Lambda

$\lambda n_1 \ n_2 \ \dots \ n_m \rightarrow \langle \text{corpo} \rangle$

Revendo a função filter

```
filter (\u -> u /= 'a') "americano"
```

Revendo a função map

```
map (\x -> x*x) [0..10]
```

Revendo a função foldr

```
foldr (\x acum -> x + acum) 0 [0..10]
```

Faça você mesmo!

Implemente as funções `filter'`, `map'` e `foldr'`, usando recursão em lista.

Faça você mesmo!

```
filter' p [] = []  
filter' p (h:t) =  
    if p h then  
        h : filter' p t  
    else  
        filter' p t
```

Faça você mesmo!

`map' f [] = []`

`map' f (h:t) = f h : map' f t`

Faça você mesmo!

$\text{reduce}' \ g \ n \ [] = n$

$\text{reduce}' \ g \ n \ (h:t) = \text{reduce}' \ g \ (g \ n \ h) \ t$

Currying

Sequenciamento de avaliação de funções que tomam mais de um argumento como parâmetro, afim de que recebam somente um argumento.

Currying

`soma n m = n + m`

`maisDois = soma 2`

Currying

```
pot x y = y ** x  
quad = pot 2
```

Composição de Funções

Uma forma de resolver um problema dividindo-o em problemas menores é através da composição de funções. Onde funções simples, quando compostas, resolvem um problema complexo.

Composição de Funções

```
remChar c [] = []  
remChar c (h:t) =  
  if c /= h then  
    h : remChar c t  
  else  
    remChar c t
```


Composição de Funções

```
> let remAE = remChar 'a' . remChar ' '
> remAE "haskell eh legal"
"hskelelehlegl"
```

Faça você mesmo!

Defina uma função chamada `filterMap` que filtra uma lista usando um predicado, após aplicado uma função sobre esta lista.

Faça você mesmo!

```
filterMap f p = filter p . map f
```

10

Escrever na saída padrão

```
lerPalavra = do
  putStrLn "Digite uma palavra:"
  palavra <- getLine
  putStrLn ("Voce digitou: " ++ palavra)
```

Ler da entrada padrão

```
lerPalavra = do
  putStrLn "Digite uma palavra:"
  palavra <- getLine
  putStrLn ("Voce digitou: " ++ palavra)
```

COMPILANDO PROGRAMAS

Main

```
main = do  
    putStrLn "Olá mundo"
```


Main

```
ghc --make <nome>.hs
```

```
lerNum = do
  linha <- getLine
  return (read linha :: Int)
```

```
main = do
  num <- lerNum
  if (mod num 2 == 0) then
    putStrLn "par!"
  else
    putStrLn "impar!"
```

QUERO APRENDER MAIS!

- Aprender Haskell será um grande bem para você! ([Link](#))
- Programação funcional com a Linguagem Haskell ([Zelda](#))
- Real World Haskell

THAT'S ALL, FOLKS

Gustavo Santos

`gfdsantos@inf.ufpel.edu.br`