

# Paradigma Funcional: Introdução

Daniel Ventura  
INF/UFG

2024/02

## Conteúdo e objetivos

- Introdução à programação funcional usando Haskell
- Objetivos:
  1. definir funções usando *equações com padrões e guardas*;
  2. implementar *algoritmos recursivos elementares*;
  3. definir *tipos algébricos* para representar dados;
  4. decompor problemas usando *funções de ordem superior* e *lazy evaluation*;
  5. escrever programas interativos usando *notação-do*;
  6. provar propriedades de programas usando *teoria equacional* e *indução*.

## O que é programação funcional?

- É um **paradigma** de programação
- No paradigma imperativo, um programa é uma **sequência de instruções** que **mudam células na memória**
- No paradigma funcional, um programa é um conjunto de **definições de funções** que aplicamos a **valores**
- Podemos programar num estilo funcional em muitas linguagens: Java, Python, Javascript, Rust, Lua
  - Muitas linguagens de programação modernas suportam construtores funcionais (funções anônimas, listas em compreensão, etc...)
- Linguagens funcionais suportam melhor o paradigma funcional
- Exemplos: Scheme, ML, OCaml, Haskell, F#, Scala, Clojure, Erlang, Elixir

## Exemplo: somar os naturais de 1 a 10

Em linguagem C:

```
total = 0;
for (i=1; i<=10; ++i)
    total = total + i;
```

- O programa é uma *sequência de instruções*
- O resultado é obtido por *mutação* das variáveis `i` e `total`

## Execução passo-a-passo

passo	instrução	i	total
1	total=0	?	0
2	i=1	1	0
3	total=total+i	1	1
4	++i	2	1
5	total=total+i	2	3
6	++i	3	3
7	total=total+i	3	6
8	++i	4	6
9	total=total+i	4	10
⋮	⋮	⋮	⋮
21	total=total+i	10	55
22	++i	11	55

## Exemplo: somar os naturais de 1 a 10

Em Haskell:

```
sum [1..10]
```

O programa consiste na *aplicação* da função *sum* à *lista* dos inteiros entre 1 e 10.

## Redução passo-a-passo

Redução da expressão original até obter um resultado que não pode ser mais simplificado.

$$\begin{aligned} \text{sum } [1..10] &= \\ &= \text{sum } [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] = \\ &= 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = \\ &= 3 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = \\ &= 6 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = \\ &= 10 + 5 + 6 + 7 + 8 + 9 + 10 = \\ &\vdots \\ &= 55 \end{aligned}$$

## Outro exemplo: *Quicksort I*

Em C:

```
int partition (int arr[], int l, int h)
{
    int x = arr[h];
    int i = (l - 1);
    for (int j = l; j <= h- 1; j++) {
        if (arr[j] <= x) {
            i++;
            swap (&arr[i], &arr[j]);
        }
    }
    swap (&arr[i + 1], &arr[h]);
    return (i + 1);
}
```

## Outro exemplo: *Quicksort II*

```
void quickSort(int A[], int l, int h)
{
    if (l < h)
    {
        int p = partition(A, l, h);
        quickSort(A, l, p - 1);
        quickSort(A, p + 1, h);
    }
}
```

## Outro exemplo: *Quicksort III*

Em Haskell:

```
qsort [] = []
qsort (x:xs) = qsort xs1 ++ [x] ++ qsort xs2
    where xs1 = [x' | x' < -xs, x' <= x]
          xs2 = [x' | x' < -xs, x' > x]
```

# Vantagens da programação funcional I

## Alto Nível

- programas mais concisos
- próximos duma especificação matemática

## Modularidade

- polimorfismo, ordem superior, *lazy evaluation*
- permitem decompor problemas em componentes re-utilizáveis

# Vantagens da programação funcional II

## Garantias de correção

- demonstrações formais de correção (provas matemáticas)
- maior facilidade em efetuar testes

## Concorrência/paralelismo

- ordem de execução não afeta os resultados

# Desvantagens da programação funcional

## Maior distância do *hardware*

- compiladores/interpretadores mais complexos;
- difícil prever os custos de execução (tempo/espço);
- alguns algoritmos são mais eficientes quando implementados de forma imperativa.

## Programação funcional - evolução histórica

- 1930s: Alonzo Church desenvolve o lambda calculus, como uma teoria matemática baseada em funções.
- 1950s: John McCarthy desenvolve o Lisp (“LISt Processor”), a primeira linguagem de programação funcional, baseada no lambda calculus, mas mantendo atribuição de variáveis.
- 1960s: Peter Landin desenvolve o ISWIM (“If you See What I Mean”), a primeira linguagem de programação funcional pura, baseada no lambda calculus e sem atribuição.
- 1970s: John Backus desenvolve o FP (“Functional Programming”), uma linguagem funcional com ênfase em funções de ordem superior.
- 1970s: Robin Milner e outros desenvolvem o ML (“Meta Language”), a primeira linguagem funcional moderna, que introduz inferência de tipos e tipos polimórficos.

## Programação funcional - evolução histórica (cont)

- 1970s - 1980s: David Turner desenvolve um conjunto de linguagens funcionais lazy, que culminaram com o desenvolvimento da linguagem Miranda (produzida comercialmente).
- 1987: um comitê internacional de pesquisadores inicia o desenvolvimento do Haskell (cujo nome vem do lógico Haskell Curry), uma linguagem funcional lazy.
- 1990s: Philip Wadler et al. desenvolvem o conceito de classes de tipos (sobrecarga) e a utilização de monadas para o tratamento de efeitos computacionais.
- 2003: É publicado o relatório Haskell 98, definindo uma versão estável da linguagem, resultado de 15 anos de trabalho.
- 2010: Publicação do padrão da linguagem Haskell 2010.

### » Haskell

- Linguagem de programação puramente funcional lazy
- Nomeada em homenagem ao matemático americano Haskell B. Curry (1900–1982)
- Concebida para ensino e também para o desenvolvimento de aplicações reais
- Resultado de mais de vinte anos de investigação por uma comunidade de base acadêmica muito ativa
- Implementações abertas e livremente disponíveis

<http://www.haskell.org>



## Haskell no mundo real I

Alguns exemplos *open-source*:

- [GHC](#) o compilador de Haskell é escrito em Haskell
- [Xmonad](#) um gerenciador de janelas usando “tiling” automático
- [Darcs](#) um sistema distribuído para gestão de código-fonte
- [Pandoc](#) conversor entre formatos de “markup” de documentos

## Haskell no mundo real II

Utilizações em *backend* de aplicações *web*:

- [Bump](#) mover arquivos entre *smartphones*  
<http://devblog.bu.mp/haskell-at-bump>
- [Janrain](#) plataforma de *user management*  
<http://janrain.com/blog/functional-programming-social-web>
- [Chordify](#) extração de acordes musicais  
<http://chordify.net>

Mais exemplos:

[http://www.haskell.org/haskellwiki/Haskell\\_in\\_industry](http://www.haskell.org/haskellwiki/Haskell_in_industry)

## Hugs

- Um interpretador interativo de Haskell
- Suporta Haskell 98 e diversas extensões
- Para aprendizagem e desenvolvimento de pequenos programas
- Disponível em <http://www.haskell.org/hugs>

## *Glasgow Haskell Compiler (GHC)*

- Compilador que gera código de máquina (código nativo)
- Suporta Haskell 98, Haskell 2010 e diversas extensões
- Otimização de código, interfaces a outras linguagens, *profiling*, grande conjunto de bibliotecas, etc.
- Inclui também o interpretador `ghci` (alternativa ao Hugs)
- Disponível em <http://www.haskell.org/ghc>

## Primeiros passos

Linux/Mac OS: executar o hugs ou ghci

Windows: executar o *WinHugs* ou ghci

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/
Loading package base ... linking ... done.
Prelude>
```

## Uso do interpretador

1. o interpretador lê uma *expressão* do teclado;
2. calcula o seu *valor*;
3. por fim imprime-o.

```
> 2+3*5
17
```

```
> (2+3)*5
25
```

```
> sqrt (3^2 + 4^2)
5.0
```

## Alguns operadores e funções aritméticas

	+	adição
	-	subtração
	*	multiplicação
	/	divisão fracionária
	^	potência (expoente inteiro)
	div	quociente (divisão inteira)
	mod	resto (divisão inteira)
	sqrt	raiz quadrada
	==	igualdade
	/=	diferença
< >	<= >=	comparações
	&&	operadores lógicos

## Algumas convenções sintáticas I

- Os argumentos de funções são **separados por espaços**
- A aplicação tem **maior precedência** do que qualquer operador

Haskell	Matemática
$f\ x$	$f(x)$
$f\ (g\ x)$	$f(g(x))$
$f\ (g\ x)\ (h\ x)$	$f(g(x), h(x))$
$f\ x\ y\ +\ 1$	$f(x, y) + 1$
$f\ x\ (y+1)$	$f(x, y + 1)$
$sqrt\ x\ +\ 1$	$\sqrt{x} + 1$
$sqrt\ (x\ +\ 1)$	$\sqrt{x + 1}$

## Algumas convenções sintáticas II

- Um operador pode ser usado como uma função escrevendo-o entre parêntesis (infixa  $\rightarrow$  pré-fixa)
- Reciprocamente: uma função pode ser usada como operador escrevendo-a entre aspas esquerdas (pré-fixa  $\rightarrow$  infix)

$(+) \ x \ y \ = \ x+y$

$(*) \ y \ 2 \ = \ y*2$

$x \text{'mod'} 2 \ = \ \text{mod } x \ 2$

$f \ x \ \text{'div'} \ n \ = \ \text{div } (f \ x) \ n$

## O prelúdio-padrão (*standard Prelude*)

O módulo *Prelude* contém um grande conjunto de funções pré-definidas:

- os operadores e funções aritméticas;
- funções genéricas sobre *listas*

... e muitas outras.

O prelúdio-padrão é automaticamente carregado pelo interpretador/compilador e pode ser usado em qualquer programa Haskell.

## Algumas funções do prelúdio I

<pre>&gt; head [1,2,3,4] 1</pre>	obter o 1º elemento
<pre>&gt; tail [1,2,3,4] [2,3,4]</pre>	remover o 1º elemento
<pre>&gt; length [1,2,3,4,5] 5</pre>	comprimento
<pre>&gt; take 3 [1,2,3,4,5] [1,2,3]</pre>	obter um prefixo
<pre>&gt; drop 3 [1,2,3,4,5] [4,5]</pre>	remover um prefixo

## Algumas funções do prelúdio II

<pre>&gt; [1,2,3] ++ [4,5] [1,2,3,4,5]</pre>	concatenar
<pre>&gt; reverse [1,2,3,4,5] [5,4,3,2,1]</pre>	inverter
<pre>&gt; [1,2,3,4,5] !! 3 4</pre>	indexação
<pre>&gt; sum [1,2,3,4,5] 15</pre>	soma dos elementos
<pre>&gt; product [1,2,3,4,5] 120</pre>	produto dos elementos

## Definir novas funções

- Vamos definir novas funções num arquivo de texto
- Usamos um editor de texto externo (e.g. Emacs)
- O nome do arquivo deve terminar em `.hs` (*Haskell script*)<sup>1</sup>

---

<sup>1</sup>Alternativa: `.lhs` (*literate Haskell script*)

## Criar um arquivo de definições

Usando o editor, criamos um novo arquivo `test.hs`:

```
dobro x = x + x
quadruplo x = dobro (dobro x)
```

Usamos o comando `:load` para carregar estas definições no GHCi.

```
$ ghci
...
> :load teste.hs
[1 of 1] Compiling Main ( teste.hs, interpreted )
Ok, modules loaded: Main.
```

## Exemplos de uso

```
> dobro 2
```

```
4
```

```
> quadruplo 2
```

```
8
```

```
> take (quadruplo 2) [1..10]
```

```
[1,2,3,4,5,6,7,8]
```

## Modificar o arquivo

Acrescentamos duas novas definições ao arquivo:

```
fatorial n = product [1..n]
```

```
media x y = (x+y)/2
```

No interpretador usamos *:reload* para carregar as modificações.

```
> :reload
```

```
...
```

```
> fatorial 10
```

```
3628800
```

```
> media 2 3
```

```
2.5
```



## Comandos úteis do interpretador

<code>:load <i>arquiv</i></code>	carregar um arquivo
<code>:reload</code>	re-carregar modificações
<code>:edit</code>	editar o arquivo atual
<code>:set editor <i>prog</i></code>	definir o editor
<code>:type <i>expr</i></code>	mostrar o tipo duma expressão
<code>:help</code>	obter ajuda
<code>:quit</code>	terminar a sessão

Podem ser abreviados, e.g. `:l` em vez de `:load`.

## Sintaxe

- Nomes de funções e variáveis começam por letra minúscula, seguidos de letras, dígitos ou `_`
- Indentação: numa sequência de definições, cada definição deve começar na mesma coluna, ou separada por `;`
- Comentários...
- *Pattern-matching ...*
- *Definição com guards (guardas)...*
- *definições locais*

## Identificadores

Os nomes de funções e argumentos devem **começar por letras minúsculas** e podem incluir letras, dígitos, sublinhados e apóstrofes:

```
fun1      x_2      y'      fooBar
```

As seguintes **palavras reservadas** não podem ser usadas como identificadores:

```
case class data default deriving do else
if import in infix infixl infixr instance
let module newtype of then type where
```

## Definições locais

Podemos fazer definições locais usando `where`.

```
a = b+c
  where b = 1
        c = 2
d = a*2
```

A indentação indica o escopo das declarações; também podemos usar agrupamento explícito.

```
a = b+c
  where {b = 1;
        c = 2}
d = a*2
```

## Indentação

Todas as definições num mesmo escopo devem começar na mesma coluna.

```
a = 1
```

```
  b = 2
```

```
  c = 3
```

ERRADO

```
a = 1
```

```
  b  = 2
```

```
  c = 3
```

ERRADO

```
a = 1
```

```
b = 2
```

```
c = 3
```

OK

A ordem das definições **não é** relevante.

## Comentários

**linha única:** começam por `--` até ao final da linha

**linhas múltiplas:** delimitados por `{-` e `-}`

```
-- fatorial de um número inteiro
fatorial n = product [1..n]
```

```
-- média de dois valores
media x y = (x+y)/2
```

```
{- ** as definições seguintes estão comentadas **
dobro x = x+x
quadrado x = x*x
-}
```

## Pattern-matching, if-then-else, guardas

```
-- if-then-else
zero x = if x == 0 then True else False

-- Multiplas definicoes e wildcard
zero 0 = True
zero _ = False

-- expressoes case
zero x = case x of
           0 -> True
           otherwise -> False

-- guardas
zero x | x == 0 = True
       | otherwise = False
```

## Exercício: Escrever as funções seguintes

1. `min3:: Int -> Int -> Int -> Int`  
que calcula o mínimo de três inteiros
2. `tresiguais:: Int -> Int -> Int -> Bool`  
que determina se 3 inteiros são iguais
3. `media:: Float -> Float -> Float`  
que calcula a média de dois inteiros
4. `par:: Int -> Bool`  
que determina se um inteiro é par
5. `sinal:: Int -> Int`  
que determina o sinal de um determinado inteiro  
(ex: `sinal 1 = 1` e `sinal de -2 = -1`).