Programação Funcional
Aula 1 — Apresentação

Pedro Vasconcelos
DCC/FCUP

2021

Conteúdo e objetivos

- Introdução à programação funcional usando a linguagem Haskell
- Objetivos de aprendizagem:
  - definir funções usando equações com padrões e guardas
  - definir novos tipos algébricos para estruturas de dados
  - definir algoritmos recursivos elementares
  - decompor problemas de programação elementares em funções
  - utlizar funções de ordem superior e *lazy evaluation*
  - escrever programas com I/O usando notação-*do*
  - provar propriedades de programas usando substituições algébricas e indução

Funcionamento

Aulas teóricas 2 × 1 h por semana (síncronas video-conferência)

Aulas práticas 2 h por semana (por video-conferência[1])

Docentes Pedro Vasconcelos (T, PL), Manuel Barbosa, Mário Florido, Sérgio Crisóstomo e Pedro Ângelo (PL)

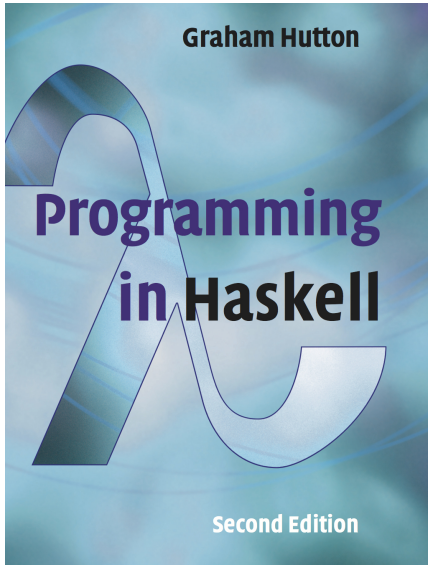
Página web `http://www.dcc.fc.up.pt/~pbv/aulas/funcional`

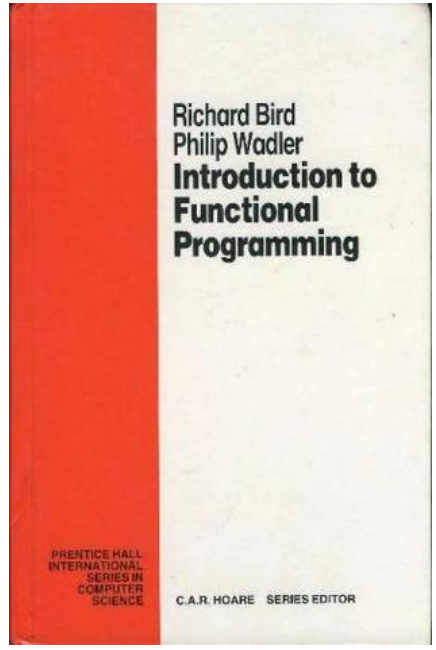
Avaliação

- Questionários no final de aulas teóricas (10%)
- Problemas de programação com correção automática (10%)
- Exame final (80%)

[1] Pelo menos até à Páscoa.

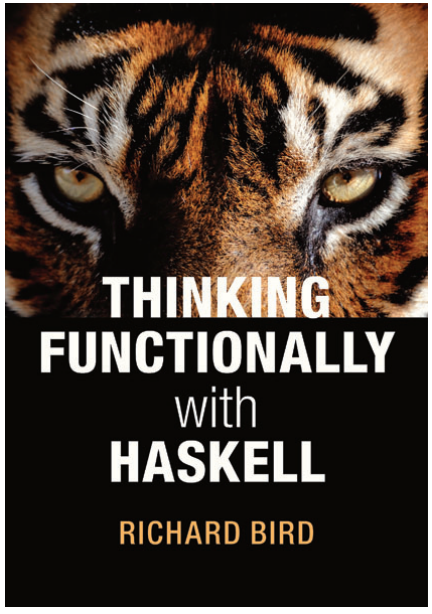
Bibliografia recomendada

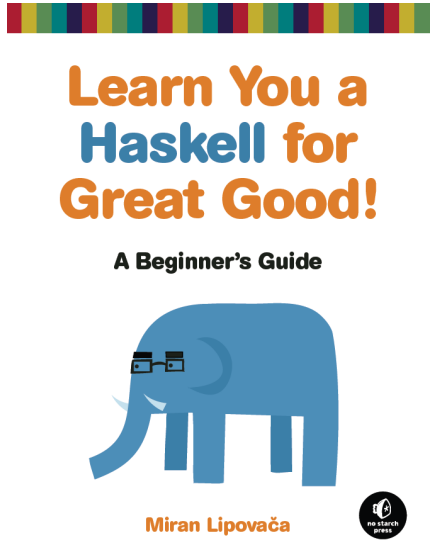




- *Programming in Haskell*, Graham Hutton, Cambridge University Press, 2007-2016.
- *Introduction to Functional Programming*, Richard Bird & Philip Wadler, Prentice-Hall International, 1988.

Outros livros





- *Thinking functionally with Haskell*, Richard Bird. Cambridge University Press, 2015.
- *Learn you a Haskell for great good!*, Miran Lipovača. `http://learnyouahaskell.com/`

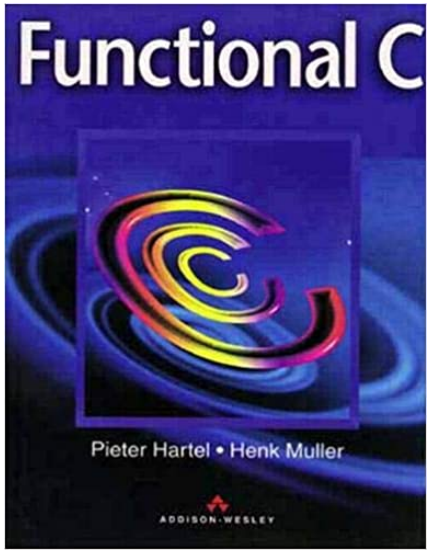
O que é a programação funcional?

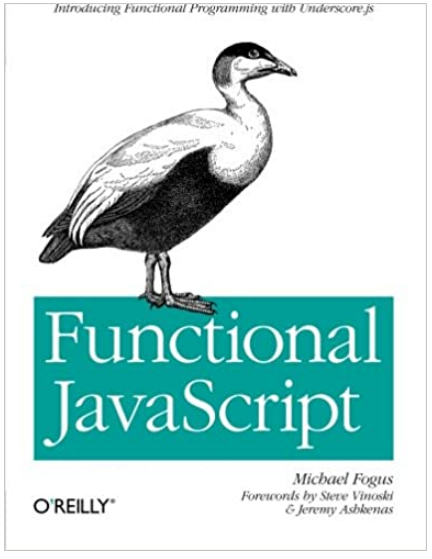
- Um paradigma: uma conceção sobre o que é um programa
- Programas em C ou Java são normalmente imperativos: sequências de comandos que modificam variáveis em memória
- No paradigma funcional, um programa é um conjunto de definições de funções que usamos para exprimir um algoritmo por composição
- Num programa puramente funcional nunca modificamos variáveis: só aplicamos funções!
- Um programa funcional é uma função

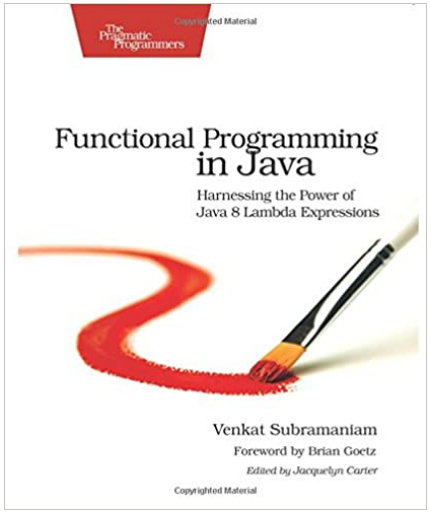
$$\textit{dados} \xrightarrow{\text{programa}} \textit{resultado}$$

expressa como composição de funções mais simples

Linguagens funcionais







- Podemos programar num estilo funcional em quase todas as linguagens. . .
- . . . mas as linguagens explicamente funcionais suportam melhor este paradigma
- Exemplos: Scheme, ML, OCaml, Haskell, F#, Scala

Exemplo

Para exemplificar a distinção de paradigmas vamos ver dois pequenos programas que calculam

$$1^2 + 2^2 + 3^2 + \cdots + 10^2$$

em C e em Haskell.

Somar os quadrados — versão imperativa

```
/* Programa imperativo em C */
int main() {
  int total = 0;
  for (int i=1; i<=10; ++i) {
     total = total + i*i;
  }
  printf("%d\n", total);
}
```

- O programa é uma sequência de instruções
- O resultado é calculado modificando as variáveis
- Compreender o programa significa compreender como os valores mudam ao longo do tempo

Execução passo-a-passo

Inspecionando valores de variáveis à entrada do ciclo:

passo	0	1	2	3	4	5	6	7	8	9	final
`i`	1	2	3	4	5	6	7	8	9	10	11
`total`	0	1	5	14	30	55	91	140	204	285	385

Somar quadrados — versão funcional

```
-- Programa funcional em Haskell
main = print (sum (map (^2) [1..10]))
```

- `[1..10]` é a sequência de inteiros de 1 a 10
- `map (^2)` calcula o quadrado de cada valor
- `sum` soma a sequência
- `print` imprime o resultado

Redução passo-a-passo

A execução de um programa funcional é redução da expressão até obter um resultado que não pode ser mais simplificado.

```
  sum (map (^2) [1..10])
=
  sum [1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2]
=
  1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2 + 9^2 +
  10^2
=
  1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100
=
  385
```

Redução passo-a-passo (cont.)

Ao contrário do programa imperativo: podemos efetuar reduções por outra ordem e obter o mesmo resultado.

```
  sum (map (^2) [1..10])
=
  sum [1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2, 10^2]
=
  sum [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
=
  1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100
=
  385
```

Porquê aprender programação funcional?

Pensar a um nível mais alto

- programas mais concisos
- próximos duma especificação matemática
- mais enfoque na análise do problema e menos em "debugging"
- ajuda a programar melhor em qualquer linguagem!

> *A language that doesn't affect the way you think about programming is not worth knowing.*

Alan Perlis (1922–1990), pioneiro norte-americano da ciência de computadores

Porquê aprender programação funcional? (cont.)

Mais modularidade

- decompor problemas em componentes pequenas e re-utilizáveis

Garantias de correção

- demonstrações de correção usando provas matemáticas
- maior facilidade em fazer testes automáticos

Concorrencia/paralelismo

- a ordem de execução não afecta os resultados

Desvantagens da programação funcional

Maior distância do *hardware*

- os compiladores e interpretadores são mais complexos
- difícil prever os custos de execução (tempo/espaço)
- alguns programas de baixo-nível necessitam de controlo preciso de tempo/espaço
- alguns algoritmos são mais eficientes quando implementados de forma imperativa

Um pouco de história

1930s Alonzo Church desenvolve o cálculo-λ, um formalismo matemático para exprimir computação usando funções

1950s Inspirado no cálculo-λ, John McCarthy desenvolve o LISP, uma das primeiras linguagens de programação

1970s–1980s Robin Milner desenvolve o Standard ML, a primeira linguagem funcional com *polimorfismo* e *inferência de tipos*

1970s–1980s David Turner desenvolve várias linguagens que empregam *lazy evaluation*, culminando na linguagem Miranda

1987 Um comité académico inicia o desenvolvimento do Haskell, uma linguagem funcional padronizada com *lazy evaluation*

Um pouco de história (cont.)

2003 Publicação do *Haskell 98*, uma definição padronizada da linguagem

2010 Publicação do padrão da linguagem *Haskell 2010*

Linguagem Haskell

`http://www.haskell.org`

- Uma linguagem funcional pura de uso genérico
- Nomeada em homenagem ao matemático americano Haskell Curry (1900–1982)
- Concebida para ensino e também para o desenvolvimento de aplicações reais
- Resultado de trinta anos de investigação por uma comunidade académica muito activa
- Utilização industrial crescente nos últimos 10–15 anos
- Implementação principal livre: *Glasgow Haskell Compiler* (GHC)

Haskell na indústria

Galois Investigação aplicada em segurança e sistemas críticos
`https://galois.com/`

Facebook Sistema de deteção de *Spam*
`https://engineering.fb.com/2015/06/26/security/fighting-spam-with-haskell/`

Github Projeto *Semantic* para análise de código-fonte de várias linguagens
`https://github.com/github/semantic`

Cardano Plataforma de cripto-moeda e *smart contracts*
`https://iohk.io/projects/cardano/`

Mais exemplos:
`http://www.haskell.org/haskellwiki/Haskell_in_industry`

Haskell em *open-source*

GHC o compilador de Haskell é escrito em Haskell
`https://www.haskell.org/ghc/`

Darcs um sistema distribuido para gestão de código-fonte
`http://darcs.net/`

Pandoc conversor entre formatos de "markup" de documentos
`https://pandoc.org/`

Codex um dos sistemas para exercícios de programação on-line usando no DCC ☺
`https://github.com/pbv/codex`

Glasgow Haskell Compiler (GHC)

- Compilador de Haskell código-máquina nativo
- Suporta Haskell 98, Haskell 2010 e bastantes extensões
- Otimização de código, interfaces a outras linguagens, *profilling*, grande conjunto de bibliotecas, etc.
- Inclui também um interpretador interativo `ghci` (útil para experimentação)
- Disponível em `http://www.haskell.org/ghc`

Primeiros passos

Linux/Mac OS: executar comando `ghci`

Windows: executar aplicação *WinGHCi* ou comando `ghci`

```
$ ghci
GHCi, version 8.6.5:
http://www.haskell.org/ghc/ :? for help
Prelude>
```

Uso do interpretador

O interpretador *lê uma expressão* do terminal, *calcula o seu valor* e *imprime-o*.[2]

```
Prelude> 2+3*5
17

Prelude> (2+3)*5
25

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

[2] Em inglês: *read-eval-print-loop* ou *REPL*.

Alguns operadores e funções aritméticas

`+`	adição
`-`	subtração
`*`	multiplicação
`/`	divisão fracionária
`^`	potência (expoente inteiro)
`div`	quociente de divisão inteira
`mod`	resto de divisão inteira
`sqrt`	raiz quadrada
`==`	comparação igualdade
`/=`	negação da igualdade (diferente)
`<` `>` `<=` `>=`	comparações de ordem

Algumas convenções sintáticas

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

Haskell	notação usual
`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 (cont.)

- Um operador pode ser usando como uma função escrevendo-o entre parêntesis
- Reciprocamente: uma função pode ser usada como operador escrevendo-a entre aspas esquerdas

```
(+) x y   ≡   x+y
(*) y 2   ≡   y*2

x`mod`2   ≡   mod x 2
f x `div` n   ≡   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
- . . . entre muitas outras

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

Algumas funções do prelúdio

```
> head [1,2,3,4]
1
> head "banana"
'b'
```

obter o 1º elemento

```
> tail [1,2,3,4]
[2,3,4]
> tail "banana"
"anana"
```

remover o 1º elemento

```
> length [1,2,3,4,5]
5
> length "banana"
6
```

comprimento

Algumas funções do prelúdio (cont.)

obter um prefixo

```
> take 3 [1,2,3,4,5]
[1,2,3]
> take 3 "banana"
"ban"
```

remover um prefixo

```
> drop 3 [1,2,3,4,5]
[4,5]
> drop 3 "banana"
"ana"
```

concatenar

```
> [1,2,3] ++ [4,5]
[1,2,3,4,5]
> "aba" ++ "cate"
"abacate"
```

Algumas funções do prelúdio (cont.)

```
> reverse [1,2,3,4,5]
[5,4,3,2,1]
> reverse "abacate"
"etacaba"
```

inverter a ordem

```
> [1,2,3,4,5] !! 3
4
> "abacate" !! 3
'c'
```

indexação a partir de 0

```
> sum [1,2,3,5]
11
```

soma dos valores

```
> product [1,2,3,5]
30
```

produto dos valores

Definir novas funções

- Vamos definir novas funções num ficheiro de texto
- Usamos um editor de texto externo (e.g. Emacs)
- O nome do ficheiro deve terminar em `.hs` (*Haskell script*)[3]

[3] Alternativa: `.lhs` (*literate Haskell script*)

Criar um ficheiro de definições

teste.hs

```
dobro x = 2*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..100]
[1,2,3,4,5,6,7,8]
```

Modificar o ficheiro

Acrescentamos novas definições e gravamos.

teste.hs

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

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

Usamos *:reload* no GHCi para carregar as modificações.

```
> :reload
> factorial 10
3628800
> media 2 3
2.5
```

Comandos úteis do interpretador

`:load` *ficheiro*	carregar um ficheiro
`:reload`	re-carregar modificações
`:edit`	editar o ficheiro actual
`:set editor` *prog*	definir o editor
`:type` *expr*	mostrar o tipo duma expressão
`:help`	obter ajuda
`:quit`	terminar a sessão

Podem ser abreviados:

- `:l` em vez de `:load`
- `:r` em vez de `:reload`
- `:t` em vez de `:type`
- `:q` em vez de `:quit`

Identificadores

Os nomes de funções e variáveis devem começar por letras mínusculas 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 âmbito das declarações; também podemos usar agrupamento explícito:

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

Indentação

Todas as definições num mesmo âmbito 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 entre as definições não é importante.

Comentários

Simples: começam por `--` até ao final de uma linha

Multi-linha: delimitados por `{-` e `-}`

```
-- calcular o factorial de um inteiro
factorial n = product [1..n]

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

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