

Programação Funcional

1ª Aula — Apresentação

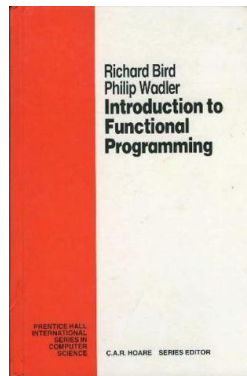
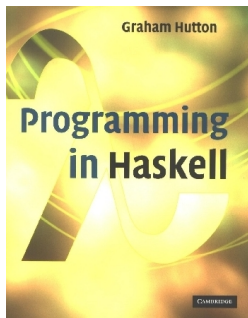
PROF. BONFIM AMARO JUNIOR

bonfimamaro@ufc.br

Conteúdo e objetivos

- Introdução à programação funcional usando Haskell
- Objetivos de aprendizagem:
 - definir funções usando *equações com padrões e guardas*
 - implementar *algoritmos recursivos elementares*;
 - definir *tipos algébricos* para representar dados;
 - decompor problemas usando *funções de ordem superior* e *lazy evaluation*;
 - escrever programas interativos usando *notação-do*;

Bibliografia recomendada



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

- *Learn you a Haskell for great good!*
<http://learnyouahaskell.com/>
- *Real World Haskell*
<http://book.realworldhaskell.org/>

O que é a programação funcional?

- 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
- Linguagens funcionais suportam melhor o paradigma funcional
- Exemplos: Scheme, ML, O'Caml, Haskell, F#, Scala

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 |

Somar os naturais de 1 a 10

Em Haskell:

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

O programa é 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.

$$\begin{aligned} & \text{sum } [1..10] = \\ &= \text{sum } [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] = \\ &= 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = \\ &= 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = \\ &= 7 + 5 + 6 + 7 + 8 + 9 + 10 = \\ &= 12 + 6 + 7 + 8 + 9 + 10 = \\ &\quad \vdots \\ &= 55 \end{aligned}$$

Vantagens da programação funcional

Nível mais alto

- programas mais concisos
- próximos de uma especificação matemática

Mais modularidade

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

Vantagens da programação funcional (cont.)

Garantias de correção

- demonstrações de correção usando provas matemáticas
- maior facilidade em efetuar testes

Concorrencia/paralelismo

- a 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.

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 Robin Milner desenvolve o **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 comercial *Miranda*TM

Um pouco de história (cont.)

- 1987 Um comitê acadêmico inicia o desenvolvimento do **Haskell**, uma linguagem funcional *lazy* padronizada e aberta
- 2003 Publicação do **Haskell 98**, uma definição padronizada da linguagem
- 2010 Publicação do padrão da linguagem **Haskell 2010**

- Uma linguagem funcional pura de uso genérico
- 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 académica muito activa
- Implementações abertas e livremente disponíveis

`http://www.haskell.org`

Haskell no mundo real

Algumas exemplos *open-source*:

GHC o compilador de Haskell é escrito em Haskell (!)

Xmonad um gestor de janelas usando “tiling” automático

Darcs um sistema distribuido para gestão de código-fonte

Pandoc conversor entre formatos de “markup” de documentos

Haskell no mundo real (cont.)

Utilizações em *backend* de aplicações *web*:

Chordify extração de acordes musicais

<http://chordify.net>

Mais exemplos:

http://www.haskell.org/haskellwiki/Haskell_in_industry

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

Glasgow Haskell Compiler (GHC)

- Compilador que gera código-máquina nativo
- Suporta Haskell 98, Haskell 2010 e bastantes 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 6.8.3: 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 |

Algumas convenções sintáticas

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

| Haskell | Matemática |
|----------------------------|-----------------|
| <code>f x</code> | $f(x)$ |
| <code>f (g x)</code> | $f(g(x))$ |
| <code>f (g x) (h x)</code> | $f(g(x), h(x))$ |
| <code>f x y + 1</code> | $f(x, y) + 1$ |
| <code>f x (y+1)</code> | $f(x, y + 1)$ |
| <code>sqrt x + 1</code> | $\sqrt{x} + 1$ |
| <code>sqrt (x + 1)</code> | $\sqrt{x + 1}$ |

Algumas convenções sintáticas (cont.)

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

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

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

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

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

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

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

obter o 1º elemento

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

remover o 1º elemento

```
> length [1,2,3,4,5]  
5
```

comprimento

```
> take 3 [1,2,3,4,5]  
[1,2,3]
```

obter um prefixo

```
> drop 3 [1,2,3,4,5]  
[4,5]
```

remover um prefixo

Algumas funções do prelúdio (cont.)

```
> [1,2,3] ++ [4,5]  
[1,2,3,4,5]
```

concatenar

```
> reverse [1,2,3,4,5]  
[5,4,3,2,1]
```

inverter

```
> [1,2,3,4,5] !! 3  
4
```

indexação

```
> sum [1,2,3,4,5]  
15
```

soma dos elementos

```
> product [1,2,3,4,5]  
120
```

produto dos elementos

Definir novas funções

- Vamos definir novas funções num ficheiro de texto
- Usamos um editor de texto externo (e.g. Bloco de Notas)
- O nome do ficheiro deve terminar em `.hs` (*Haskell script*)²

²Alternativa: `.lhs`
(*iterate Haskell script*)

Criar um ficheiro de definições

Usando o editor, criamos um novo ficheiro `teste.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 ficheiro

Acrescentamos duas novas definições ao ficheiro:

```
factorial n = product [1..n]  
media x y = (x+y)/2
```

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

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

Comandos úteis do interpretador

| | |
|--------------------------------------|---------------------------------|
| <code>:load <i>fich</i></code> | carregar um ficheiro |
| <code>:reload</code> | re-carregar modificações |
| <code>:edit</code> | editar o ficheiro actual |
| <code>:set editor <i>prog</i></code> | definir o editor |
| <code>:type <i>expr</i></code> | mostrar o tipo de uma expressão |
| <code>:help</code> | obter ajuda |
| <code>:quit</code> | terminar a sessão |

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

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 âmbito das declarações; também podemos usar agrupamento explícito.

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

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 das definições **não é** relevante.

Comentários

Simples: começam por -- até ao final da linha

Múltiplos: delimitados por {- e -}

```
-- factorial de um número inteiro  
factorial 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  
-}
```

Exercícios

1) Escreva uma função que calcula a área de um círculo. Sua função deve receber apenas o raio como parâmetro de entrada.

$$\text{area} = \pi * r^2$$

a) Usando o GHCi aplique os parâmetros para um círculo de $r = 12$

2) Escreva uma função que retorna se um número passado por parâmetro é par.

a) Usando o GHCi teste sua função

3) Escreva uma função que calcula o somatório dos números do intervalo $[x, y]$ ambos parâmetros. (Recursiva)**

Ex. Para $x = 1$ e $y = 10$ o resultado será 55

4) Defina uma função **recursiva** chamada potencia tal que o resultado seja x elevado a y

Exercícios

5) Implemente uma função 'suc x' que recebe x e retorna x+1. $\text{suc } x = x + 1$

Sem usar o operador (+), apresente uma funcao soma x y que produza a mesma saída

OBS: Considere apenas números inteiros não negativos
