

Programação Funcional

André Santos

(a partir de *slides* elaborados por **André Santos, Sérgio Soares , Fernando Castor e
Márcio Cornélio**)

O que é Programação Funcional?

- Baseia-se na idéia de calcular
- Paradigma de programação onde
 - Programas consistem em definições de **dados** e **funções**
 - **Execução de um programa = Avaliação de expressões**
 - Funções não têm **efeitos colaterais** e são valores de **primeira ordem**

Por que falar sobre programação funcional?

- Visão clara de conceitos fundamentais:
 - abstração e tipos abstratos de dados
 - recursão
 - genericidade, polimorfismo, sobrecarga
- Ajuda na programação em **outros paradigmas**
- Primeiro paradigma ensinado em várias universidades importantes
 - Stanford, Berkeley

Objetivos da programação funcional

- Programação com um alto nível de abstração, possibilitando:
 - alta produtividade
 - programas mais concisos
 - menos erros
 - provas de propriedades sobre programas

Usos práticos

- Programas com milhares de linhas de código: compiladores, provadores de teoremas, sistemas Web, serviços de chat de grande escala, etc.
 - Simulação para a estimativa de riscos em operações financeiras no ABN/AMRO (Haskell)
 - Chat do Facebook (Erlang)

Usos práticos (cont.)

- Erlang
 - Programação de switches de redes na Ericsson
 - Serviço de chat do Facebook
- Scala (linguagem híbrida, parte funcional)
 - Serviço de filas de mensagens no Twitter
- Scheme e LISP
 - Ensino de programação em várias universidades
- ML, F#
 - Verificação de HW e SW na Microsoft e na Intel

Neste curso...

- A ênfase é em conceitos
- Haskell é um ferramenta prática de apoio,
não o objetivo central

Ambiente utilizado no curso

- Linguagem de programação: Haskell
- Compilador GHC (**Glasgow Haskell Compiler**) está disponível gratuitamente para Windows e Unix
 - Inclui um interpretador: GHCi
 - <http://www.haskell.org/ghc/>

Bibliografia

- Livro adotado:
 - *Haskell – The Craft of Functional Programming*. Simon Thompson. Addison-Wesley, 1996.
- Referências auxiliares
 - *Real World Haskell*. Bryan O'Sullivan, Don Stewart, and John Goerzen. O'Reilly, 2008.
 - <http://book.realworldhaskell.org/read/>
 - *Learn You a Haskell for a Great Good – A Beginner's Guide*. Miran Liovaca. 2011
 - <http://learnyouahaskell.com/>
 - <http://www.lpac.ac.uk/SEL-HPC/Articles/FuncArchive.html>
 - <http://www.haskell.org>

Histórico

Máquina
de Turing



Linguagens de
programação
imperativa

Cálculo
Lambda



Linguagens de
programação
funcional

Histórico

- Anos 70
 - Linguagens funcionais:
 - forma de transpor a "**crise de software**"
 - “*Can Programming Be Liberated From the Von Neumann Style ? A Functional Style and Its Algebra of Programs*”, John Backus, 1978 CACM Turing Award Lecture
- Anos 80 e 90
 - Grande interesse acadêmico e **pouco da indústria**
- Anos 2000 => **Ressurgimento!**
 - Ou **surgimento**, pelo menos **na indústria**

Modelo Computacional

Mapeamento (função) de valores de entrada em valores de saída



Ausência de estado e comandos (atribuição + controle)

Relação clara e explícita entre entrada e saída

Programação Funcional

- Todos os subprogramas são vistos como funções
 - Eles recebem argumentos e retornam soluções simples.
 - A solução retornada depende apenas da entrada
 - O tempo em que uma função é chamada é irrelevante
 - Funções **sem efeitos colaterais**

Programação Funcional

Visão Crítica

- Vantagens
 - Manipulação mais simples de programas
 - Legibilidade
 - Modularidade
 - Corretude
 - Prova de propriedades
 - Concorrência explorada de forma natural
 - Sem estado compartilhado

Programação Funcional

Visão Crítica

- Problemas
 - “O mundo não é funcional!”
 - Esforço inicial não-desprezível
 - Implementações ineficientes
 - Relevância depende do domínio da aplicação
 - Mecanismos primitivos de E/S e formatação
 - Interface com o usuário

Notação: Programação baseada em definições

```
answer :: Int
```

```
answer = 42
```

```
greater :: Bool
```

```
greater = (answer > 71)
```

```
yes :: Bool
```

```
yes = True
```


Definição de Funções

```
square :: Int -> Int
```

```
square x = x * x
```

```
allEqual :: Int -> Int -> Int -> Bool
```

```
allEqual n m p = (n == m) && (m == p)
```

```
maxi :: Int -> Int -> Int
```

```
maxi n m | n >= m    = n  
          | otherwise = m
```

Aplicação de Funções

`square 5` `-- = 25`

`square (5)` `-- = 25`

`allEqual 1 2 3` `-- = False`

`allEqual (1,2,3)` `-- ERRO!!!`

`allEqual (1) (2) (3)` `-- = False`

`maxi 24 645` `-- = 645`

Prova de propriedades

- Exemplo:

$$\begin{aligned}\text{addD } a \ b &= 2 * (a+b) \\ &= 2 * (b+a) = \text{addD } b \ a\end{aligned}$$

- Válida para quaisquer argumentos a e b
- Não seria válida em linguagens imperativas, com variáveis globais ...

Em uma linguagem imperativa...

```
int b = 1;
```

```
...
```

```
int f (int x) {
```

```
    b = x;
```

```
    return (5)
```

```
}
```

```
addD (f 3) b == addD b (f 3) ?
```

Exemplo

Em um sistema de controle de vendas:

- suponha vendas semanais dadas pela função
`vendas :: Int -> Int`
- total de vendas da semana 0 à semana n?
 $\text{vendas } 0 + \text{vendas } 1 + \dots + \text{vendas } (n-1) + \text{vendas } n$

```
totalVendas :: Int -> Int
```

```
totalVendas n
```

```
  | n == 0      = vendas 0
```

```
  | otherwise = totalVendas (n-1) + vendas n
```

Recursão

- Definir **caso base**, i.e. valor para `fun 0`
- Definir o valor para `fun n`
usando o valor de `fun (n-1)`
Este é o **caso recursivo**.

```
maxVendas :: Int -> Int
maxVendas n
  | n == 0      = vendas 0
  | otherwise = maxi  (maxVendas (n-1))
                    (vendas n)
```

Casamento de Padrões

- Permite usar padrões no lugar de variáveis, na definição de funções:

```
maxVendas :: Int -> Int
maxVendas 0 = vendas 0
maxVendas n = maxi (maxVendas (n-1))
                  (vendas n)
```

```
totalVendas :: Int -> Int
totalVendas 0 = vendas 0
totalVendas n = totalVendas (n-1) +
                  vendas n
```

Casamento de Padrões

```
myNot :: Bool -> Bool
```

```
myNot True  = False
```

```
myNot False = True
```

```
myOr :: Bool -> Bool -> Bool
```

```
myOr True  x = True
```

```
myOr False x = x
```

```
myAnd :: Bool -> Bool -> Bool
```

```
myAnd False x = False
```

```
myAnd True  x = x
```


Regras para Padrões

- Todos os padrões (esquerda) devem ter tipos **compatíveis**
 - Não necessariamente **iguais**
- Os casos devem ser exaustivos
 - não é obrigatório → funções parciais

Não deve haver ambiguidade

- ordem dos padrões usada para resolver conflitos

Notação

- Maiúsculas:
 - Tipos e **Construtores** (para tipos algébricos)
- Minúsculas:
 - Funções, variáveis e parâmetros
- *Case sensitive*
- comentários:
 - isso é um comentario de uma linha
 - {- comentario de varias linhas... -}

Notação

- $f\ n + 1$ $-- = (f\ n) + 1$
- $f\ (n + 1)$
- $2 + 3$
- $(+) 2\ 3$
- $maxi\ 2\ 4$
- $2\ `maxi` 4$

Erros comuns

```
square x = x
```

```
*x
```

```
parse error on input `*'
```

```
answer = 42; newline = '\n' --OK
```

```
funny x = x +
```

```
1
```

```
parse error (possibly incorrect  
indentation)
```

```
Funny x = x+1
```

```
Not in scope: data constructor `Funny'
```

Exercícios

- Defina as seguintes funções:

- fatorial

`fat :: Int -> Int`

- compara se quatro números são iguais

`all4Equal :: Int -> Int -> Int -> Int -> Bool`

- `all4Equal` usando `allEqual`

- retorna quantos parâmetros são iguais

`equalCount :: Int -> Int -> Int -> Int`

Definições Locais

```
sumSquares :: Int -> Int -> Int
```

```
sumSquares x y = sqX + sqY  
  where sqX = x * x  
        sqY = y * y
```

```
sumSquares x y = sq x + sq y  
  where sq z = z * z
```

```
sumSquares x y = let sqX = x * x  
                  sqY = y * y  
                  in sqX + sqY
```

Definições Locais

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

```
maxThreeOccurs m n p = (mx, eqCount)
```

```
  where mx = maxiThree m n p
```

```
        eqCount = equalCount mx m n p
```

```
...
```

- **let** definições **in** expressão
- definições **where** definições