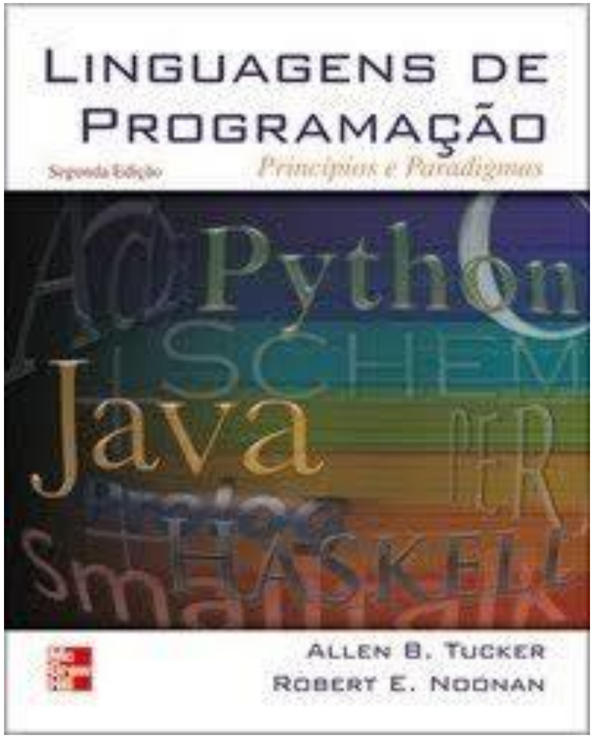
Paradigma Funcional

Fabio Lubacheski
fabio.lubacheski@mackenzie.br

Leituras recomendadas



TUCKER, A. B.; NOONAN, R. E. **Linguagens de programação: Princípios e Paradigmas**. Capítulo 14 sessões 14.1 e 14.3

Paradigma funcional

- O **paradigma funcional** modela um problema computacional de maneira diferente em relação aos paradigmas **imperativos** e **orientada a objetos**.

- O **paradigma funcional** enfatiza a **aplicação de funções**, em contraste ao **paradigma imperativo**, que enfatiza mudanças no **estado do programa** (variáveis) e **comandos** (Máquina de Turing).

- Já o **paradigma orientada a objetos** é baseada nas formas e comportamento dos objetos e suas interações (mensagens)

Paradigma funcional

- No **paradigma funcional "puro"** não há uma **noção de estado** e, portanto, não há necessidade de uma **instrução de atribuição**, e consequentemente **não temos variável**, as iterações (repetições) são representadas por **funções recursivas**.

- Em **termos práticos** muitas **linguagens funcionais s**uportam as noções de variável e atribuição.

- As expressões (funções) são utilizadas para **cálculo de valores com dados imutáveis**.

Paradigma funcional

- Em linguagens de programação imperativa, uma variável como x representa uma localização na memória.

- A instrução abaixo significa "*atualizar o* ***estado do programa*** *somando 1 ao valor armazenado na célula de memória denominada* x *e depois armazenar aquela soma novamente naquela célula de memória*".

$$x = x + 1$$

Paradigma funcional – funções matemáticas

- Um problema no paradigma funcional é modelado como uma coleção de **funções matemáticas**, mapeando **entrada** e **saída**.

Em uma linguagem funcional é "*pura"* a soma de um valor é representada por uma função matemática definida por:

$$f(x) = x + 1$$

Cálculo de lambda

- A base da programação funcional é o *cálculo lambda*, desenvolvido por Alonzo Church (1941).

  https://galois.com/team/alonzo-church/

  λ

- Uma **expressão *lambda*** específica os parâmetros e a definição de uma função (corpo da função), mas não seu nome. Para a função que representa a soma de um valor

  $$\boldsymbol{f(x) = x + 1}$$

  a sua representação usando cálculo de lambda seria:

  $$\boldsymbol{(\lambda\, x\, .\, x + 1)}$$

Cálculo de lambda

x calcula x somado mais 1

$$(\lambda \quad x \quad . \quad x + 1)$$

Uma função de que

- O identificador x é um parâmetro usado no corpo da função, a aplicação de uma expressão lambda a um valor é representado por:

$$((\lambda\, x\, .\, x + 1)\ 10)$$

Que dá o resultado **11**.

Aplicação do paradigma funcional

- O **paradigma funcional** surgiu motivado pela necessidade dos pesquisadores no desenvolvimento de **inteligência artificial**, **prova de teoremas**, **computação simbólica** e **processamento de linguagem natural**.
- Essas necessidades não eram particularmente bem resolvidas pelas linguagens imperativas da época.
  - **Lisp**: 1ª linguagem funcional desenvolvida por John McCarthy em 1960,
  - **Scheme**: um variação do Lisp criado por Steele e Sussman em 1975.
  - **Haskell**: Introduz conceitos recentes da programação funcional, a linguagem Haskell foi desenvolvida Peyton-Jones e outros em 1990.

Linguagem Haskell

- A linguagem de programação Haskell é fundamentada no **cálculo de lambda**, e o nome da linguagem é uma homenagem ao matemático **Haskell Curry**, um dos pioneiros no desenvolvimento do cálculo lambda.

  https://en.wikipedia.org/wiki/Haskell_Curry

- O compilador mais popular de Haskell é o GHC (*Glasgow Haskell Compiler*).

  http://www.haskell.org/ghc/download.html

  O GHC compreende um compilador de linha de comando (**ghc**) e também um ambiente interativo (**GHCi**), que permite a avaliação de expressões de forma interativa.

Alguns links para aprender Haskell

- Aprender Haskell será um grande bem para você!

  http://haskell.tailorfontela.com.br/

- Wiki para Haskell

  https://wiki.haskell.org/Haskell_em_10_minutos

- Wiki Books

  https://pt.wikibooks.org/wiki/Haskell/Primeiro_passo

- Tutorial spoint

  https://www.tutorialspoint.com/haskell/index.htm

Operadores Matemáticos em Haskell

Sejam `x` e `y` valores numéricos

adição:	`x + y`
subtração:	`x - y`
multiplicação:	`x * y`
divisão:	`x / y`
divisão inteira:	`div x y`
resto da divisão:	`mod x y`
potencia expoente inteiro:	`x ^ y`
potencia expoente real:	`x ** y`

Operadores Relacionais em Haskell

Sejam `x` e `y` valores numéricos

igualdade: `x == y`

diferente: `x /= y`

maior: `x > y`

menor: `x < y`

maior ou igual: `x >= y`

menor ou igual: `x <= y`

Operadores Lógicos em Haskell

E lógico: `True && True  == True`

E lógico: `True && False == False`

OU lógico: `True || False == True`

OU lógico: `False || False == False`

Não lógico: `not True == False`

Linguagem Haskell – Cálculo de lambda

- O cálculo de lambda pode ser representado por uma expressão (função anônima) em Haskell, formada por uma sequência de padrões representando os argumentos da função, e um corpo que especifica como o resultado pode ser calculado usando os argumentos:

$$(\backslash padrão_1 \; ... \; padrão_n \; -> \; expressão)$$

- Assim a expressão lambda

$$\mathbf{(\lambda\, \mathit{x} \,.\, \mathit{x} + 1)}$$

- Em Haskell ficaria assim:

```
(\x -> x + 1)
```

Linguagem Haskell – Cálculo de lambda

- No Haskell usa-se o símbolo **** em vez da letra grega λ, e podemos usar uma expressão lambda para calcular um valor.

```
(\x -> x + 1)10
=> 11
```

- Escreva em Haskell as expressões lambda abaixo, e depois teste com algum valor para verificar o resultado do cálculo:

$(\lambda\, x\ \ y.\ x + y)$

$(\lambda\, x\ .\ x^{2})$

$(\lambda\, x\ .\ (\lambda\, y\ .\ x*y)\ )$

$(\lambda\, x\ .\ (\lambda\, y\ .\ x^{y})\ )$

Linguagem Haskell – Funções

- Podemos nomear uma expressões lambda e reutiliza-la em outras funções.

```
(\x y -> x + y)
```

- A sintaxe para definir uma função seria:

$$f \; arg_1 \; ... \; arg_{\mathrm{n}} \;\; = exp$$

- Onde f é o nome da função e arg_1 ... arg_{n} são os argumentos da função (sem parênteses ou vírgula), a resposta da função é o valor de exp. A expressão lambda convertida em função ficaria:

```
-- calcula soma entre dois numeros
soma x y = x+y
```

Linguagem Haskell – Funções

- Para usar a função basta salva-la em um programa com a extensão `.hs` e carregá-la no interpretador `GHCi`:

```
:load testaSoma.hs
[1 of 1] Compiling Main ( testaSoma.hs, interpreted )
Ok, one module loaded.
soma 10 30
=> 40
```

- Na chamada da função também **não usamos parênteses nem vírgulas**.

Fluxo de condicional if-then-else

- Para calcular o resposta de uma função podemos usar a uma estrutura condicional **`if`**, a ***condição*** é uma **expressão booleana** (chamada **predicado**) e exp_1 (chamada ***consequência***) e exp_2 (chamada ***alternativa***) são expressões de um **mesmo tipo**.
- A resposta da função é o va*lor* de exp_1 se a ***condição*** é verdadeira, ou o valor de exp_2 se a ***condição*** é falsa.

$$f\ arg_1 \ldots arg_{\mathrm{n}} = \texttt{if}\ condição\ \texttt{then}\ exp_1\ \texttt{else}\ exp_2$$

- Uma função para encontrar o menor entre dois números seria:

```
menor x y = if x <= y then x else y
```

Fluxo de condicional com guarda (switch..case)

- Uma função pode ser definidas através de **equações com guardas**. Uma **guarda** é formada por uma **sequência de cláusulas** escritas logo após a lista de argumentos.
- Cada **cláusula** tem barra vertical (|=pipe) e uma ***condição*** e uma expressão (*expr*), separados por (=). **`otherwise`** é uma *condição* que captura todas as outras situações que ainda não foram consideradas. As **guardas** devem ficar todas com o mesmo **alinhamento à direita**.

$$f\ arg_1 \ldots arg_n$$

$$|\quad condição_1 = exp_1$$

$$\vdots$$

$$|\quad condição_n = exp_n$$

$$|\quad \texttt{otherwise} = exp_{\text{caso contrário}}$$

Funções com guarda

- Com exemplo veja a função para calcular o menor entre 3 números.

```
menor_tres x y z
      | x <= y && x <= z = x
      | y <= z = y
      | otherwise = z
```

Funções com where

- Em Haskell é possível definir **valores** e **funções auxiliares** em uma definição principal de uma função.

- Isto pode ser feito escrevendo-se uma cláusula **`where`** ao final de uma cláusula com guarda ou para acrescentar parâmetros a uma função.

- A cláusula **`where`** faz definições que são locais à definição principal de uma função, ou seja, o escopo dos nomes definidos em uma cláusula **`where`** restringe-se a somente a função principal.

Funções com where

- Um exemplo mostra uma função para análise do índice de massa corporal.

```
calc_imc peso altura
       | imc <= 18.5 = "Abaixo do peso!"
       | imc <= 25.0 = "Peso ideal!"
       | imc <= 30 = "Acima do peso!"
       | otherwise = "Muito acima do peso!"
       where imc = peso / altura ^ 2
```

Exercícios funções

1) Escreva um função que recebe um número e verifica se ele é par. A função retorna `True` caso o número seja par ou `False` caso contrário.

2) Usando a função par escreva a função impar que verifica se um número é impar utilizando a função par para calcular a resposta, o resultado da função par é negado com o operado `not` do Haskell,

3) Dado um n, escreva uma função que devolva a diferença absoluta entre n e 21, exceto devolva o dobro da diferença absoluta se n for maior que 21.

Exercícios funções

4) Escreva uma função que receba três valores para os lados de um triângulo. Na função verifique se os lados fornecidos realmente formam um triângulo, se formarem devolva tipo de triângulo temos: isósceles, escaleno ou equilátero.

5) Escreva uma função que receba 3 valores quaisquer e verifique se os valores podem ser considerados uma tripla de Pitágoras, ou seja, a soma dos quadrados de dois números é igual ao quadrado terceiro. Caso tenhamos uma tripla de Pitágoras a função devolve “eh uma tripla de Pitagoras” e caso não seja o a função devolve “nao eh tripla de Pitagoras”. Exemplos:

3 5 4 é uma tripla de Pitágoras

5 3 4 é uma tripla de Pitágoras

2 4 3 Não é tripla de Pitágoras

Recursividade

- **Recursividade** é uma ideia inteligente que desempenha um papel central na **programação funcional**, é o mecanismo básico **para repetições** nas linguagens funcionais.

- Um problema é dito **recursivo** se ele for definido em **termos de si próprio**, ou seja, recursivamente. Por exemplo veja a função de **recorrência** do cálculo do fatorial.

$$n! = \begin{cases} 1 & se\ n = 0 \\ n.(n-1)! & se\ n > 0 \end{cases}$$

Recursividade

- Dado que o problema é recursivo podemos escrever uma função recursiva com a seguinte estratégia:

se a entrada do problema é pequena e conhecida então

resolva-a diretamente e **pare a recursão**;

senão,

reduza-a uma entrada menor do mesmo problema,

aplique este método à entrada menor

e volte à entrada original.

Recursividade

- A estrutura recursiva do cálculo do fatorial fica evidente. Assim, teremos a seguinte implementação recursiva na linguagem C.

```
int fatorial( int n ){
    #base da recursão (condição de parada)
    if( n == 0 )
        return 1;
    else
        return n*fatorial(n-1);
}
```

Uma implementação recursiva se caracteriza por conter no corpo da função uma **chamada para si mesma**.

A chamada de si mesma é dita **chamada recursiva**.

Recursividade em Haskell

- Em Haskell a função fatorial, poderia ser escrita das seguintes maneiras.

```
-- definições equivalentes do fatorial
fatorial1 0 = 1
fatorial1 n = n * fatorial1(n-1)

fatorial2 n = if n == 0 then 1 else n * fatorial2(n-1)

fatorial3 n
   | n == 0 = 1
   | otherwise = n * fatorial3 (n - 1)
```

Recursividade em Haskell

- A primeira versão, `fatorial1`, é escrita em um estilo recursivo, de modo que os casos especiais são definidos primeiro e seguidos pelo caso geral.

- A segunda versão, `fatorial2`, usa o estilo de definição mais tradicional `if-then-else`.

- A terceira versão, `fatorial3`, usa guarda em cada direção; esse estilo é útil quando há mais de duas alternativas.

Recursividade em Haskell

- Como em Haskell não são necessários parênteses na definição e na chamada da função, mas na chamada

  `fatorial1(n-1)` os parênteses se fazem necessário pois senão seria feita a chamada primeiro depois a subtração.
- Haskell é sensível a maiúsculas/minúsculas, e **obrigatoriamente** as funções e parâmetros devem começar com uma letra minúscula, além disso, uma função não pode redefinir uma função padrão Haskell.
- As funções em Haskell são fortemente tipadas e polimórficas e por padrão, o Haskell usa inteiros de precisão infinita,

  ```
  fatorial1 30
  265252859812191058636308480000000
  ```

Recursividade em Haskell

- Escreva uma função que receba como argumento um número natural maior que 1, ou seja, somente um argumento na entrada da função. A função verifica se o número informado é primo ou não. Caso o número seja primo é devolvido `True` e caso contrário `False`.

- Como podemos fazer as divisões recursivas do número informado ?

Exercícios

1) A sequência [0, 1, 1, 2, 3, 5, 8, 13, 21, ...] é conhecida como sequência ou série de Fibonacci, e tem aplicações teóricas e práticas, na medida em que alguns padrões na natureza parecem segui-la. Pode ser obtida através da recorrência abaixo:

$$\mathrm{Fib}(n) = \begin{cases} 0 & se\ n = 0 \\ 1 & se\ n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & se\ n > 1 \end{cases}$$

Escreva uma função em Haskell que calcula a sequência de Fibonacci.

Exercícios

2) Imagine que a linguagem Haskell não possui mais o operador de multiplicação (*), por sorte os matemáticos definiram que a multiplicação pode ser definida através de somas sucessivas:

$a*b = a + a ... + a$ ou seja a somado b vezes.

Para a =4 e b =5 teremos 4+4+4+4+4, ou seja, 4 somado 5 vezes.

Escreva a **função recursiva** em Haskell que tem como entrada a e b, a função calcula e devolve a multiplicação de $a*b$ utilizando a regra acima.

Exercícios

3) Imagine agora que a linguagem Haskell não possui o operador de potência (^ ou **), sabemos que a potência pode ser definida através de multiplicações, assim:

x^$y = x * x ...*x$ ou seja x multiplicado y vezes.

Para x=2 e y=5 teremos 2*2*2*2*2 que é igual a 32.

Escreva a função recursiva em Haskell que tem como entrada x e y, a função calcula e devolve a potência de x^y utilizando a regra acima,

Exercícios

4) Imagine agora que a linguagem Haskell não possui o operador de potência (^ ou **), nem o operador de multiplicação ($*$), sabemos que a potência pode ser definida através de multiplicações e a multiplicação pode ser definida por somas sucessivas, assim:

x^$y = x * x$ ...*x ou seja x multiplicado y vezes.

Para x=2 e y=5 teremos 2*2*2*2*2 que é igual a 32.

Escreva a função recursiva em Haskell que tem como entrada x e y, a função calcula e devolve a potência de x^y utilizando a regra acima, para calcular a multiplicação utilize a função implementada no exercício 2.

Exercícios

5) A multiplicação Russa consiste em:

a. Escrever os números A e B, que se deseja multiplicar na parte superior das colunas.

b. Dividir A por 2, sucessivamente, ignorando o resto até chegar à unidade, escrever os resultados da coluna A.

c. Multiplicar B por 2 tantas vezes quantas se haja dividido A por 2, escrever os resultados sucessivos na coluna B.

d. Somar todos os números da coluna B que estejam ao lado de um número ímpar da coluna A.

Exemplo: 27 × 82 = **2214**

```
A    B     Parcelas
27   82    82
13   164   164
6    328   -
3    656   656
1    1312  1312
```

Soma: 2214

Escreva uma função em Haskell que calcula a multiplicação russa de 2 entradas;

Exercícios

6) A função abaixo calcula o máximo divisor comum dos inteiros positivos m e n. Escreva uma função em Haskell que faz o mesmo cálculo.

```
int mdc( int m, int n ){

    while( n != 0){
        r = m % n;
        m = n;
        n = r
    }
    return m;
}
```

Exercícios

7) A expressão abaixo converge para a raiz quadrada de A, sendo A>0. Calcule um valor aproximado da raiz quadrada de um número dado A

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{A}{x_{n-1}} \right), \quad x_0 = 1, n \in N$$

Escreva uma função em Haskell que calcule a raiz quadrada de um número usando o algoritmo através de **n** iterações.

Exercícios

8) O fatorial duplo de um número natural n é o produto de todos os números de 1 (ou 2) até n, contados de 2 em 2.

Por exemplo, o fatorial duplo de 8 é 8 6 4 2 = 384, e o fatorial duplo de 7 é 7 5 3 1 = 105.

Defina uma função em Haskell para calcular o fatorial duplo usando recursividade.

9) A raiz quadrada inteira de um número inteiro positivo n é o maior número inteiro cujo quadrado é menor ou igual a n. Por exemplo, a raiz quadrada inteira de 15 é 3, e a raiz quadrada inteira de 16 é 4.

Defina uma função recursiva em Haskell para calcular a raiz quadrada inteira.

Exercicios

10) Escreva uma função equivalente em Haskell para o código abaixo:

```
int função( int n)
   int x = 1;
   for (int i = 1; i <= 10; i++) {
     x = x*2;
   }
   return x;
}
```

Fim