

ERRATA

ALGORITMOS FUNCIONAIS

Autor: José Augusto Navarro Garcia Manzano

1º Edição

REF: 978-85-508-1447-6

**PÁGINA 89 (ÚLTIMO PARÁGRAFO)
ONDE SE LÊ:**

$$\begin{aligned}x_0 &= 1 \\x_1 &= x \\x_{(n+1)} &= x \cdot (x_n)\end{aligned}$$

LEIA-SE:

$$\begin{aligned}x^0 &= 1 \\x^1 &= x \\x^{(n+1)} &= x \cdot (x^n)\end{aligned}$$

**PÁGINA 94 (ÚLTIMO PARÁGRAFO)
ONDE SE LÊ:**

$$\begin{aligned}F_0 &= 0 \\F_1 &= 1 \\F_2 &= 1 \\F_n &= F_{n-1} + F_{n-2}\end{aligned}$$

LEIA-SE:

$$\begin{aligned}F_0 &= 0 \\F_1 &= 1 \\F_2 &= 1 \\F_n &= F_{n-1} + F_{n-2}\end{aligned}$$

PÁGINA 95 (PRIMEIRO PARÁGRAFO)
ONDE SE LÊ:

$$\begin{aligned} \text{fib}(n) &= 0, & \text{se } n &= 0 \\ &= 1, & \text{se } n &= 1 \\ &= 1, & \text{se } n &= 2 \\ &= \text{fib}(n-1) + \text{fib}(n-2), & \text{se } n &> 2 \end{aligned}$$

LEIA-SE:

$$\begin{aligned} \text{fib}(n) &= 0, & \text{se } n &= 0 \\ &= 1, & \text{se } n &= 1 \\ &= 1, & \text{se } n &= 2 \\ &= \text{fib}(n-1) + \text{fib}(n-2), & \text{se } n &> 2 \end{aligned}$$

PÁGINA 111 (PRIMEIRO PARÁGRAFO)
ONDE SE LÊ:

Em um contexto maior de aplicação, respeitando o uso de variáveis com definição imutável de valor, expressões lambda nas linguagens de programação funcionais são processadas com o uso de avaliação preguiçosa. Assim sendo, considere a função $\lambda x.(\lambda y.x \cdot x + y \cdot y)$ que tem por finalidade apresentar o resultado do quadrado da soma de dois valores definidos como argumentos da função. Com base **nos valores 2 e 4** observe como o processamento da operação é executado, adaptador de Moura (2019, p. 34).

LEIA-SE:

Em um contexto maior de aplicação, respeitando o uso de variáveis com definição imutável de valor, expressões lambda nas linguagens de programação funcionais são processadas com o uso de avaliação preguiçosa. Assim sendo, considere a função $\lambda x.(\lambda y.x \cdot x + y \cdot y)$ que tem por finalidade apresentar o resultado do quadrado da soma de dois valores definidos como argumentos da função. Com base **nos valores 2 e 5** observe como o processamento da operação é executado, adaptador de Moura (2019, p. 34).

PÁGINA 111 (SEGUNDO PARÁGRAFO)

ONDE SE LÊ:

Na linha 1 é estabelecida a função anônima e a definição dos argumentos X com o valor 2 e Y com o valor 4. Na sequência, na linha 2 usando avaliação preguiçosa pega primeiro o argumento indicado mais à esquerda e aplica sua operação registrando o resultado da multiplicação de 2 por 2 como 4 na linha 3. Após o encerramento da ação entre as linhas 2 e 3 entra a operação da linha 4 que de posse do valor 5 efetua sua multiplicação por ele mesmo e aponta a ação da linha 5 que gera na linha 6 o resultado da operação como 29.

LEIA-SE:

Na linha 1 é estabelecida a função anônima e a definição dos argumentos X com o valor 2 e Y com o valor 5. Na sequência, na linha 2 usando avaliação preguiçosa pega primeiro o argumento indicado mais à esquerda e aplica sua operação registrando o resultado da multiplicação de 2 por 2 como 4 na linha 3. Após o encerramento da ação entre as linhas 2 e 3 entra a operação da linha 4 que de posse do valor 5 efetua sua multiplicação por ele mesmo e aponta a ação da linha 5 que gera na linha 6 o resultado da operação como 29.

PÁGINA 139 (QUINTO PARÁGRAFO)

DESCONSIDERAR TRECHO:

Com base na definição da função `complista`(conjunto, qualificador) considere para seu uso a lista [1, 2, 3] a seguir:

PÁGINA 149 (PRIMEIRO PARÁGRAFO)

ONDE SE LÊ:

Veja em seguida a definição da função `checa_primo([x])` junto a linguagem Hope.

LEIA-SE:

Veja em seguida as definições das funções `tamanho([x])` e `checa_primo([x])` junto a linguagem Hope.

(TAMBÉM NO SEGUNDO PARÁGRAFO)

ONDE SE LÊ:

Veja em seguida a definição da função `checa_primo([x])` junto a linguagem Haskell.

LEIA-SE:

Veja em seguida as definições das funções `tamanho([x])` e `checa_primo([x])` junto a linguagem Haskell.

PÁGINA 185 (ÚLTIMO PARÁGRAFO)

ONDE SE LÊ:

O processamento de sequências sobre as relações funcionais entre conjuntos é uma ação de tratamento de dados aplicada recursivamente a partir do uso de funções de ordem superior ou de alta ordem sobre os elementos individuais de uma lista a partir das ações operacionais: dobragem, mapeamento, filtragem, **redução, compactação e descompactação**. Este conjunto de funções são ferramentas de excelente performance para o trabalho de extração de dados, principalmente em operações de mineração de dados.

LEIA-SE:

O processamento de sequências sobre as relações funcionais entre conjuntos é uma ação de tratamento de dados aplicada recursivamente a partir do uso de funções de ordem superior ou de alta ordem sobre os elementos individuais de uma lista a partir das ações operacionais: dobragem, mapeamento, filtragem, **redução e compactação**. Este conjunto de funções são ferramentas de excelente performance para o trabalho de extração de dados, principalmente em operações de mineração de dados.

PÁGINA 187 (DOIS ÚLTIMOS PARÁGRAFOS)

ONDE SE LÊ:

Veja em seguida a definição da função **fatiar(i, f, [x])** junto a linguagem Hope que deverá ser usada a partir da sintaxe "mapa([1, 2, 3, 4, 5], \ x => x * 3);".

LEIA-SE:

Veja em seguida a definição da função **mapa([x], \função)** junto a linguagem Hope que deverá ser usada a partir da sintaxe "mapa([1, 2, 3, 4, 5], \ x => x * 3);".

ONDE SE LÊ:

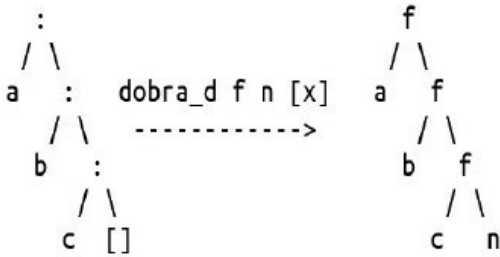
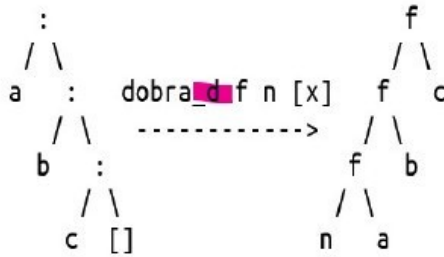
Note em seguida a definição da função **fatiar i f [x]** junto a linguagem Haskell que deverá ser usada a partir da sintaxe "mapa [1, 2, 3, 4, 5] (* 3)".

LEIA-SE:

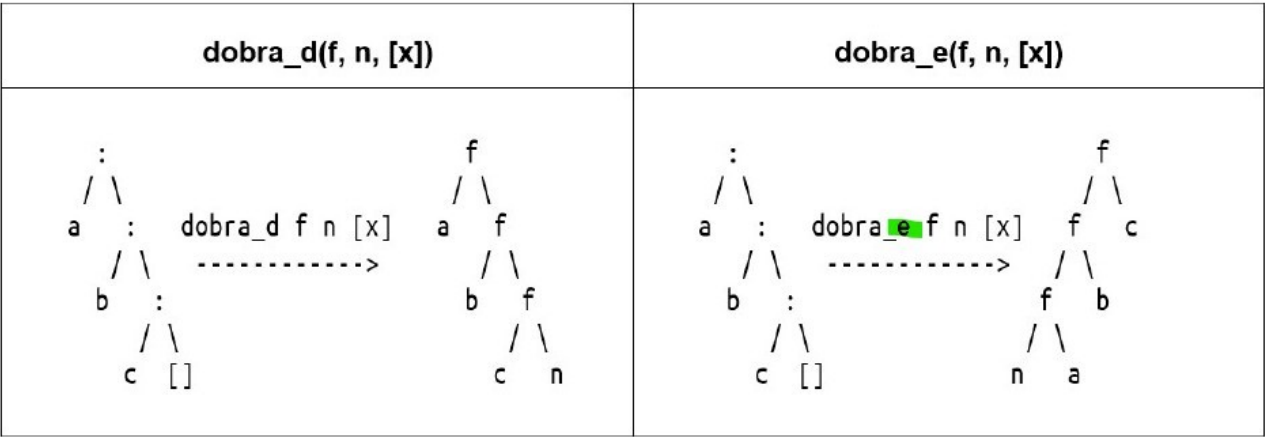
Note em seguida a definição da função **mapa [x] (função)** junto a linguagem Haskell que deverá ser usada a partir da sintaxe "mapa [1, 2, 3, 4, 5] (* 3)".

PÁGINA 193 (FIGURA 4.5)

ONDE SE LÊ:

dobra_d(f, n, [x])	dobra_e(f, n, [x])
	

LEIA-SE:



**PÁGINA 195 (SEGUNDO PARÁGRAFO)
ONDE SE LÊ:**

Veja, em seguida, as definições das funções `dobra_d (f) n [x]` e `dobra_e (f) n [x]` com a linguagem Haskell:

```
:{
dobra_d :: (Num a) => (a -> a -> a) -> a -> [a] -> a
dobra_d f n [] = n
dobra_d f n (x : xs) = f x (dobra_d f n xs)
dobra_e :: (Num a) => (a -> a -> a) -> a -> [a] -> a
dobra_e f n [] = n
dobra_e f n (x : xs) = dobra_e f (f n x) xs
:}
```

LEIA-SE:

Veja, em seguida, as definições das funções `dobra_d (f) n [x]` e `dobra_e (f) n [x]` com a linguagem Haskell:

```
:{
dobra_d :: (Num a) => (a -> a -> a) -> a -> [a] -> a
dobra_d f n [] = n
dobra_d f n (x : xs) = f x (dobra_d f n xs)

dobra_e :: (Num a) => (a -> a -> a) -> a -> [a] -> a
dobra_e f n [] = n
dobra_e f n (x : xs) = dobra_e f (f n x) xs
:}
```

TÓPICO 4.4 (Gerenciamento de listas)

ONDE SE LÊ:

Para se obter a posição cardinal em que certo valor existente dentro de uma lista se encontra é apresentada a função chamada `busca(n, [x])` que mostra uma de duas respostas possíveis: **mostra o valor da posição cardinal onde o valor se encontra na lista ou informa mensagem orientando que o valor na lista é inexistente.** Observe o código definido em *português funcional*:

LEIA-SE:

(...) **mostra o valor da posição cardinal considerando a lista da esquerda para a direita onde o elemento pesquisado se encontra ou informa mensagem orientando que o valor na lista é inexistente.**

Para fazer uso da função `busca(n, [x])` há diversas estratégias. A seguir apresenta-se uma maneira muito simples a partir de outras funções já definidas. Neste caso, será criada uma função auxiliar denominada `pega_pos(n, [x])` que será executada a partir da função `busca(n, [x])`.

A função `pega_pos(n, [x])` localiza a posição cardinal de um valor de uma lista ao contrário do que é desejado, ou seja, da direita para a esquerda. Desta forma, o valor de posição do elemento no extremo direito da lista é o valor `0` e o último valor no extremo esquerdo da lista é o de posição `n - 1`. Caso o valor pesquisado não exista a função retorna uma mensagem de advertência. Assim sendo, observe o código em *português funcional*.

```
1: pega_pos (número, lista número) >> número  
2: pega_pos (_, []) << escreva "elemento nao existe na lista"  
3: pega_pos (n, x :: xs) << se (n = x) então tamanho (xs) senão pega_pos (n, xs)
```

Com base na definição da função `pega_pos(n, [x])` considere para seu uso as operações a seguir:

```
?| pega_pos(4, [1,2,3,4,5])  
  
>| 1  
?| _  
  
?| pega_pos(negativo(2), [1,2,3,4,5])  
>| elemento nao existe na lista  
?/ _
```

A linha `1` especifica o protótipo da função para a recepção de um valor referente a um elemento e a lista a ser pesquisada, retornando como resposta o valor de sua posição se o elemento obviamente existir na lista.

Na linha `2` detecta pela correspondência de padrões o uso de qualquer valor que obtenha ao final uma lista vazia e quando isso ocorrer a mensagem de advertência é apresentada.

Na linha `3` a correspondência de padrão indica a recepção do elemento `"n"` sobre a lista `"x ::xs"`. A cada iteração realizada recursivamente é verificado se o valor de `"n"` é igual ao valor de `"x"` e sendo a busca é encerrada e o valor da posição do elemento é retornado pela função `tamanho (xs)`, caso contrário a iteração recursiva é processada.

Veja em seguida a definição da função `pega_pos(n, [x])` junto a linguagem Hope.

```
dec pega_pos : num # list num -> num;

--- pega_pos (_, []) <= error "elemento nao existe na lista";
--- pega_pos (n, x :: xs) <= if (n = x) then tamanho (xs) else pega_pos (n, xs);
```

Note em seguida a definição da função `pega_pos n [x]` junto a linguagem Haskell.

```
:{

pega_pos :: (Eq a, Num a) => a -> [a] -> Int
pega_pos _ [] = error "elemento nao existe na lista"
pega_pos n (x : xs) = if n == x then tamanho xs else pega_pos n xs
:}
```

A partir da definição da função `pega_pos(n, [x])` será preparada a função `busca(n, [x])` que apresentará a posição ordinal de certo valor dentro de uma lista no sentido da esquerda para a direita. Para esta função funcionar como esperado é necessário fazer a chamada da função `pega_pos(n, [x])` passando o valor a ser pesquisado e realizando a execução oposta da lista. Assim sendo, observe o código da função `busca(n, [x])` em *português funcional*.

```
1: busca (número, lista número) >> número

2: busca (_, []) << escreva "lista invalida"
3: busca (n, x :: xs) << pega_pos (n, oposto(x :: xs))
```

A linha 1 especifica o protótipo da função para a recepção de um valor referente a um elemento e a lista a ser pesquisada, retornando como resposta o valor de sua posição se o elemento obviamente existir na lista.

A linha 2 faz a verificação da correspondência do padrão para validar a operação da função. Se uma lista vazia for fornecida a mensagem informando que a lista é invalida é apresentada.

Na linha 3 a correspondência de padrão indica a recepção da posição "n" sobre lista `x :: xs` e executa a chamada da função `pega_pos()` passando o valor do elemento "n" e a inversão da lista `x :: xs` com o uso da função `oposto()`.

```
?| busca(4, [1,2,3,4,5])

>| 3
?| _

?| busca(6, [1,2,3,4,5])
>| elemento nao existe na lista
?| _
```

Veja em seguida a definição da função `busca(n, [x])` junto a linguagem Hope.

```
dec busca : num # list num -> num;

--- busca (_, []) <= error "lista invalida";
--- busca (n, x :: xs) <= pega_pos (n, oposto (x :: xs));
```

Note em seguida a definição da função `busca n [x]` junto a linguagem Haskell.

```
:{
```

```
busca :: (Eq a, Num a) => a -> [a] -> Int
busca _ [] = error "lista invalida"
busca n (x : xs) = pega_pos n (oposto (x: xs))
:}
```