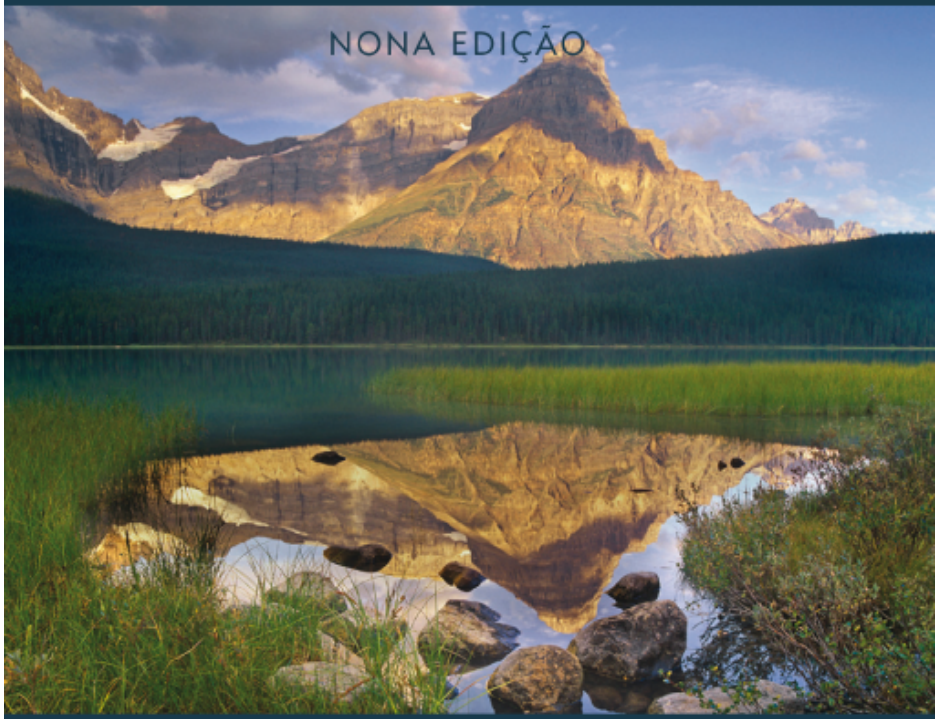
CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

NONA EDIÇÃO

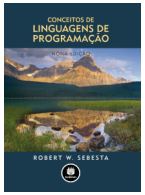


ROBERT W. SEBESTA

bookman

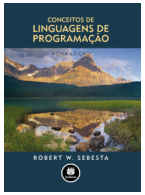
Capítulo 15

Linguagens de Programação Funcional



Introdução

- O projeto das linguagens imperativas é baseado diretamente na arquitetura de computadores von Neumann
  - Eficiência é a principal preocupação, em vez da adequação da linguagem para desenvolvimento de software
- O projeto das linguagens funcionais é baseado em *funções matemáticas*
  - Uma sólida base teórica, também próxima do usuário, mas relativamente despreocupada com a arquitetura das máquinas em que os programas serão executados
- Resultam em programas mais legíveis, mais confiáveis e mais propensos a serem corretos.
- Nem expressões nem funções tem efeitos colaterais

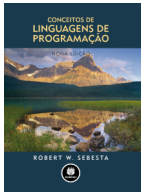


Funções matemáticas

- Uma função matemática é um mapeamento de membros de um conjunto, chamado de conjunto domínio, para outro conjunto, chamado de conjunto imagem
- A ordem de avaliação de suas expressões é controlada por recursão e expressões condicionais, e não por sequência e repetição interativa.
- Funções matemáticas *definem* valores, enquanto funções de linguagens de programação *produzem* valores. (sempre definem o mesmo valor e não mudam de status)
- Uma *expressão lambda* especifica o parâmetro e o mapeamento de uma função com a seguinte forma

  `λ(x) x * x * x`

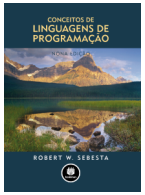
  para a função `cube (x) = x * x * x`



Funções Simples

- Geralmente escritas como um nome de função, seguido de uma lista de parâmetros entre parênteses, seguidos pela expressão de mapeamento:
- `cubo(x) ≡ x * x * x`

- ≡ (é definido como)



Expressões lambda

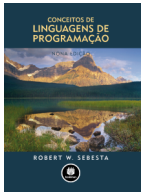
- Expressões lambda descrevem funções sem nome
- Uma *expressão lambda* especifica os parâmetros e o mapeamento de uma função com a seguinte forma

  `λ(x) x * x * x`

- Antes da avaliação um parâmetro representa qualquer membro do conjunto domínio, mas durante a avaliação ele é vinculadod a um membro em particular.
- São aplicadas aos parâmetros colocando-se os parâmetros depois da expressão

  por exemplo, `(λ(x) x * x * x)(2)`

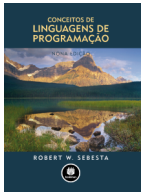
  que resulta no valor `8`
- Expressões lambda, como outras definições de função, podem ter mais de um parâmetro



Formas funcionais

- Uma função de ordem superior, ou *forma funcional*, é uma função que recebe funções como parâmetros ou uma que leva a uma função como resultado, ou ambos

- Composição de funções e Aplicar-para-todos são dois exemplos de formas funcionais.



Composição de funções

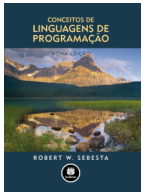
- É uma forma funcional que tem dois parâmetros funcionais ou leva a uma função cujo valor é o primeiro parâmetro de função real aplicado ao resultado do segundo

  Forma: `h ≡ f ° g`

  que significa `h (x) ≡ f ( g ( x))`

  Para `f (x) ≡ x + 2` e `g (x) ≡ 3 * x`,

  `h ≡ f ° g` resulta `(3 * x)+ 2`



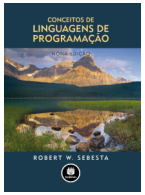
Aplicar-para-todos (apply-to-all)

- É uma forma funcional que recebe uma única função como um parâmetro e produz uma lista de valores obtidos pela aplicação da função de cada elemento de uma lista de parâmetros

  Forma: α

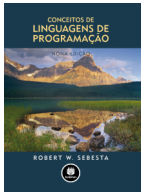
  Para `h (x) ≡ x * x`

  α`( h, (2, 3, 4))` resulta `(4, 9, 16)`



Fundamentos das linguagens de programação funcional

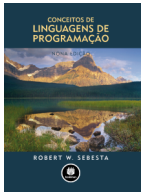
- O objetivo do projeto de uma linguagem de programação funcional é mimetizar funções matemáticas ao máximo possível
- A abordagem para a solução de problemas é fundamentalmente diferente de abordagens usadas com linguagens imperativas
  - Em uma linguagem imperativa, as operações são realizadas e os resultados são armazenados em variáveis para uso posterior
  - Gestão das variáveis é uma preocupação constante e uma fonte de complexidade para a programação imperativa
- Em uma linguagem de programação funcional, variáveis não são necessárias, como é o caso da matemática



Fundamentos das linguagens de programação funcional (continuação)

- *Transparência referencial* – Em uma linguagem de programação funcional, a avaliação de uma função sempre produz o mesmo resultado, dado os mesmos parâmetros

- *Recursão em cauda* – Se uma chamada recursiva em uma função é a última expressão na função. Escrever funções recursivas que podem ser convertidas automaticamente para iteração

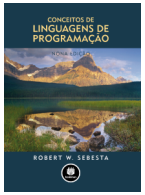


Haskell

- Similar a ML (usa uma sintaxe similar, tem escopo estático, é fortemente tipada e usa o mesmo método de inferência)
- É *puramente* funcional

```
fatorial 0 = 1
fatorial n = n * fact (n - 1)

fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```



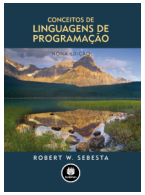
Definições de funções com diferentes variações de parâmetros

- ```
  fact n
    |  n == 0 = 1
    |  n > 0 = n * fact(n - 1)
  ```

```
sub n
  | n < 10 = 0
  | n > 100      = 2
  | otherwise    = 1

square x = x * x
```

  - Funciona com qualquer tipo numérico de `x`
```

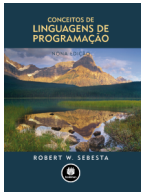


Listas

- Notação de lista: coloca os elementos entre colchetes
  por exemplo,

```
directions = ["north", "south", "east", "west"]
```

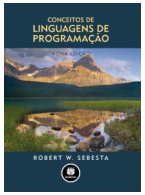
- Comprimento: `#`
  por exemplo, `#directions` é `4`
- Séries aritméticas com o operador ..
  por exemplo, `[2, 4..10]` is `[2, 4, 6, 8, 10]`
- Concatenação é com `++`
  por exemplo, `[1, 3] ++ [5, 7]` resulta `[1, 3, 5, 7]`
- `1:[3, 5, 7]` resulta `[1, 3, 5, 7]`



Função fatorial

```
product [] = 1
product (a:x) = a * product x

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



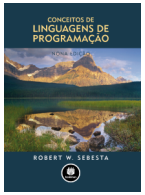
Compreensão de lista

- Lista de quadrados dos números de 1 a 20:

```
[n * n | n ← [1..20]]
```

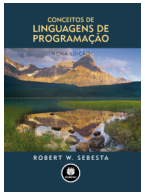
- Retorna uma lista de todos os fatores:

```
factors n = [i | i ← [1..n `div` 2],
              n `mod` i == 0]
```



Quicksort (ordenação rápida)

```
sort [] = []
sort (a:x) =
     sort [b | b ← x; b <= a] ++
     [a] ++
     sort [b | b ← x; b > a]
```



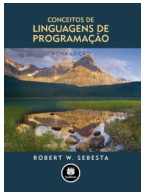
Avaliação preguiçosa (atrasada, adiada ou postergada)

- Uma linguagem é *estrita* se ela requer que todos os parâmetros reais sejam completamente avaliados
- Uma linguagem é *não estrita* se ela não tem o requisito de ser estrita
- Linguagens não estritas são mais eficientes e permitem alguns recursos interessantes – *listas infinitas*
- Avaliação preguiçosa – Computa apenas aqueles valores que são necessários
- Números positivos

```
positives = [0..]
```

- Determinando se `16` é um número quadrado

```
member [] b = False
member(a:x) b=(a == b)||member x b
squares = [n * n | n ← [0..]]
member squares 16
```



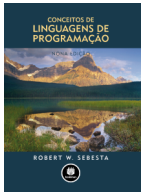
Função member

- A função member poderia ser escrito assim:

```
member [] b = False
member(a:x) b=(a == b)||member x b
```

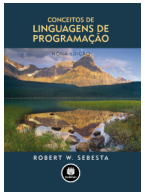
- No entanto, a definição de member funcionaria corretamente com squares apenas se o número dado fosse um quadrado perfeito; se não, continuaria gerando quadrados para sempre. A versão a seguir sempre funcionará:

```
member2 (m:x) n
     | m < n = member2 x n
     | m == n = True
     | otherwise = False
```



Aplicações das linguagens funcionais

- APL é usada para programas “descartáveis”
- LISP é usada para inteligência artificial
  - Representação do conhecimento
  - Aprendizado de máquina
  - Processamento de linguagem natural
  - Modelo de fala e visão
- Scheme é usada para ensinar programação introdutória em algumas universidades



Comparando linguagens funcionais e imperativas

- Linguagens imperativas:
  - Execução eficiente
  - Semântica complexa
  - Sintaxe complexa
  - Concorrência é projetada pelo programador
- Linguagens funcionais:
  - Semântica simples
  - Sintaxe simples
  - Execução ineficiente
  - Programas podem ter concorrência automaticamente