

# Acumuladores

---

Marco A L Barbosa  
malbarbo.pro.br

Departamento de Informática  
Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.  
<http://github.com/malbarbo/na-progfun>

# Introdução

## Falta de contexto na recursão

- Não nos preocupamos com o contexto do uso quando criamos funções recursivas
  - Não importa se é a primeira vez que a função está sendo chamada ou se é a 100ª
- Este princípio de independência do contexto facilita a escrita de funções recursivas, mas pode gerar problemas em algumas situações
- Vamos ver um exemplo

## Exemplo 7.1

Dado uma lista de distâncias relativas entre pontos (começando da origem) em uma linha, defina uma função que calcule a distância absoluta a partir da origem

## Exemplo 7.1

```
;; Lista(Número) -> Lista(Número)  
;; Converte uma lista de distâncias relativas para uma lista  
;; de distâncias absolutas. O primeiro item da lista  
;; representa a distância da origem.  
(define relativa->absoluta-tests  
  (test-suite  
    "relativa->absoluta tests"  
    (check-equal? (relativa->absoluta empty) empty)  
    (check-equal? (relativa->absoluta (list 50 40 70 30 30))  
                  (list 50 90 160 190 220))))
```

## Exemplo 7.1

```
(define (relativa->absoluta lst)
  (define (somador n) ( $\lambda$  (x) (+ x n)))
  (cond
    [(empty? lst) empty]
    [else
     (cons (first lst)
           (map (somador (first lst))
                (relativa->absoluta (rest lst))))]))
```

## Exemplo 7.1

- Esta função realiza muito trabalho para resolver o problema (tempo de execução de  $\Theta(n^2)$ )
- Se tivéssemos que resolver este problema manualmente, utilizaríamos outro método, o de somar a distância absoluta de um ponto com a distância relativa do próximo
- Vamos tentar definir uma função mais parecida com este método manual

## Exemplo 7.1

- Como é uma função que processa uma lista, começamos com o modelo

```
(define (rel->abs lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (rel->abs (rest lst)))]))
```

## Exemplo 7.1

- Como seria a avaliação de `(rel->abs (list 3 2 7))`?

```
(rel->abs (list 3 2 7))  
(cons ... 3 ...  
  (converte (list 2 7)))  
(cons ... 3 ...  
  (cons ... 2 ...  
    (converte (list 7))))
```

- O primeiro item da lista deve ser 3, e é fácil calcular este item. Mas o segundo item deve ser `(+ 3 2)` e a segunda chamada de `rel->abs` não tem como “saber” qual era o valor do primeiro item da lista. Este “conhecimento” foi perdido.

## Exemplo 7.1

- Vamos acrescentar um parâmetro `acc-dist` que representa a distância acumulada, ou seja, a distância absoluta até o ponto anterior
- Conforme os números da lista são processados, eles são somados a `acc-dist`
- O valor inicial de `acc-dist` precisa ser `0`, então definimos `rel->abs` como uma função local e fazemos a chamada inicial com `acc-dist` apropriado

## Exemplo 7.1

```
(define (relativa->absoluta lst0)
  (define (rel->abs lst acc-dist)
    (cond
      [(empty? lst) empty]
      [else
       (cons (+ (first lst) acc-dist)
              (rel->abs (rest lst)
                        (+ (first lst) acc-dist)))]))
  (rel->abs lst0 0))
```

## Falta de contexto na recursão

- No exemplo anterior vimos que a falta de contexto tornou uma função mais complicada do que necessária (e também mais lenta)
- Veremos a seguir um exemplo em que a falta de contexto faz uma função usar mais memória do que é necessário

# **Processos iterativos e recursivos**

## Processos iterativos e recursivos

- Considere as seguintes implementações para a função que soma dois números naturais utilizando a função `add1` e `zero?`

```
(define (soma a b)
  (if (zero? b)
      a
      (add1 (soma a (sub1 b)))))
```

```
(define (soma-alt a b)
  (if (zero? b)
      a
      (soma-alt (add1 a) (sub1 b))))
```

- Qual é o processo gerado quando cada função é avaliada com os parâmetros `4` e `3`?

# Processos iterativos e recursivos

```
(define (soma a b)
  (if (zero? b)
      a
      (add1
       (soma a (sub1 b))))))
```

```
(soma 4 3)
(add1 (soma 4 2))
(add1 (add1 (soma 4 1)))
(add1 (add1 (add1 (soma 4 0))))
(add1 (add1 (add1 4)))
(add1 (add1 5))
(add1 6)
7
```

Este é um **processo recursivo**

Ele caracterizado por uma sequência de operações adiadas e tem um padrão de “cresce e diminui”

# Processos iterativos e recursivos

```
(define (soma-alt a b)
  (if (zero? b)
      a
      (soma-alt (add1 a) (sub1 b))))
```

(soma-alt 4 3)  
(soma-alt 5 2)  
(soma-alt 6 1)  
(soma-alt 7 0)  
7

Este é um **processo iterativo**

O “espaço” necessário para fazer a substituição não depende do tamanho da entrada

- Na avaliação da expressão (soma-ait 4 3) no exemplo anterior, o valor de a foi usado como um acumulador, armazenando a soma parcial
- O uso de acumulador neste problema reduziu o uso de memória

## **Recursão em cauda**

- Uma **chamada em cauda** é a chamada de uma função que acontece como última operação dentro de uma função
- Uma **função recursiva em cauda** é aquela em que todas as chamadas recursivas são em cauda

- A forma de criar processos iterativos em linguagens funcionais é utilizando recursão em cauda
- Os compiladores/interpretadores de linguagens funcionais otimizam as recursões em cauda de maneira que não é necessário manter a pilha da chamada recursiva, o que torna a recursão tão eficiente quanto um laço em uma linguagem imperativa. Esta técnica é chamada de **eliminação da chamada em cauda**

## **Projetando funções com acumuladores**

## Projetando funções com acumuladores

- Usar acumuladores é algo que fazemos **depois** que definimos a função e não antes
- Os princípios para projetar funções com acumuladores são
  - Identificar que a função se beneficia ou precisa de um acumulador
    - Torna a função mais simples
    - Diminui o tempo de execução
    - Diminui o consumo de memória
  - Entender o que o acumulador significa

- Vamos reescrever diversas funções utilizando acumuladores

## Exemplo 7.2

```
;; Lista -> Natural  
;; Conta a quantidade de elementos de uma lista.  
(define tamanho-tests  
  (test-suite  
    "tamanho tests"  
    (check-equal? (tamanho empty) 0)  
    (check-equal? (tamanho (list 4)) 1)  
    (check-equal? (tamanho (list 4 7)) 2)  
    (check-equal? (tamanho (list 4 8 -4)) 3)))  
  
(define (tamanho lst)  
  (cond  
    [(empty? lst) 0]  
    [else (add1 (tamanho (rest lst)))]))
```

## Exemplo 7.2

- Existe algum benefício em utilizar acumulador?
  - Como o tamanho da resposta não depende do tamanho da entrada, esta função está usando mais memória do que é necessário, portanto ela pode beneficiar-se do uso de acumuladores
- Qual o significado do acumulador?
  - O conhecimento que se perde na chamada recursiva é a quantidade de elementos já “vistos”
  - Portanto, vamos criar um acumulador que representa esta quantidade

## Exemplo 7.2

```
(define (tamanho lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst) (add1 acc))]))
  (iter lst0 0))
```

## Exemplo 7.3

```
;; Lista(Número) -> Número
;; Soma os números de uma lista.
(define soma-tests
  (test-suite
   "soma tests"
   (check-equal? (soma empty) 0)
   (check-equal? (soma (list 3)) 3)
   (check-equal? (soma (list 3 5)) 8)
   (check-equal? (soma (list 3 5 -2)) 6)))

(define (soma lst)
  (cond
   [(empty? lst) 0]
   [else (+ (first lst)
            (soma (rest lst)))]))
```

## Exemplo 7.3

- Existe algum benefício em utilizar acumulador?
  - Como o tamanho da resposta não depende do tamanho da entrada, esta função está usando mais memória do que é necessário, portanto ela pode beneficiar-se do uso de acumuladores
- Qual o significado do acumulador?
  - O conhecimento que se perde na chamada recursiva é a soma dos elementos já “vistos”
  - Portanto, vamos criar um acumulador que representa este valor

## Exemplo 7.3

```
(define (soma lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst) (+ (first lst) acc))]))
  (iter lst0 0))
```

## Exemplo 7.4

```
;; Lista -> Lista
;; Inverte a ordem dos elementos de lst.
(define invertre-tests
  (test-suite
    "invertre tests"
    (check-equal? (invertre empty) empty)
    (check-equal? (invertre (list 2)) (list 2))
    (check-equal? (invertre (list 2 8 9)) (list 9 8 2))))

(define (invertre lst)
  (cond
    [(empty? lst) empty]
    [else (append (invertre (rest lst))
                  (list (first lst)))]))
```

## Exemplo 7.4

- Existe algum benefício em utilizar acumulador?
  - Neste caso a função é mais complicada do que o necessário. Isto porque o resultado da chamada recursiva é processada por outra função recursiva (append). Além disso, o tempo de execução desta função é  $\Theta(n^2)$  (o que intuitivamente é muito para inverter uma lista)
- Qual o significado do acumulador?
  - O conhecimento que se perde na chamada recursiva são os elementos que já foram “vistos”
  - Vamos criar um acumulador que representa os elementos já vistos (uma lista)

## Exemplo 7.4

```
(define (inverta lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst)
                  (cons (first lst) acc))]))
  (iter lst0 empty))
```

**Função foldl**

- Vamos observar as semelhanças das funções tamanho, soma e `inverte`

## Função foldl

```
(define (tamanho lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst) (add1 acc))]))
  (iter lst0 0))
```

## Função foldl

```
(define (soma lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst) (+ (first lst) acc))]))
  (iter lst0 0))
```

## Função foldl

```
(define (inverta lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst) (cons (first lst) acc))]))
  (iter lst0 empty))
```

- Vamos criar uma função chamada `reduz-acc` (pré-definida em Racket com o nome `foldl`) que abstrai este comportamento

## Função foldl

```
;; (X Y -> Y) Y Lista(X) -> Y
;; (reduz-acc f base (list x1 x2 ... xn) devolve
;; (f xn ... (f x2 (f x1 base))))
;; Veja a função pré-definida foldl.
(define (reduz-acc f base lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst)
                  (f (first lst) acc))]))
  (iter lst0 base))
```

## Função foldl

- Redefinimos as funções em termos de `reduz-acc`

```
(define (tamanho lst)
  (define (soma1-no-segundo a b)
    (add1 b))
  (reduz-acc soma1-no-segundo 0 lst))
```

```
(define (soma lst)
  (reduz-acc + 0 lst))
```

```
(define (inverte lst)
  (reduz-acc cons empty lst))
```

`foldr vs foldl`

- `foldr` e `foldl` produzem o mesmo resultado se a função `f` for associativa
- Quando possível, utilize a função `foldl`, pois ela pode utilizar menos memória
- Não tenha receio de utilizar a função `foldr`, muitas funções ficam mais complicadas, ou não podem ser escritas em termos de `foldl`, como por exemplo, `map` e `filter`

## Referências

- Seções 30 e 31 do livro HTDP
- Seção 1.2 do livro SICP