# Autorreferência e recursividade

## Parte II

Programação Funcional
Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

# Números Naturais

Um número natural é atômico ou composto?

- Atômico quando usado em operações aritméticas, comparações, etc
- Composto quando uma iteração precisa ser feita baseado no valor do número

Se um número natural pode ser visto como dado composto

- Quais são as partes que compõe o número?
- Como (de)compor um número?

# Definição

Um número **Natural** é

- 0; ou
- `(add1 n)` onde *n* é um número **Natural**

Baseado nesta definição, criamos um modelo para funções com números naturais.

```
(define (fn-para-natural n)
  (cond
    [(zero? n) ...]
    [else
      (... n
           (fn-para-natural (sub1 n)))]))
```

# Definição

```
;; as funções add1, sub1 e zero? são pré-definidas

;; compõe um novo natural a partir de um existente
;; semelhante ao cons
> (add1 8)
9
;; decompõe um natural
;; semelhante a rest
> (sub1 8)
7
;; verifica se um natural é 0
;; semelhante ao empty?
> (zero? 8)
#f
> (zero? 0)
#t
```

# Exemplo: soma naturais

Dado um número natural $n$, defina uma função que some os números naturais menores ou iguais a $n$.

# Exemplo: soma naturais

Especificação

```
;; Natural -> Natural
;; Soma todos os números naturais de 0 até n
(examples
 (check-equal? (soma-nat 0) 0)
 (check-equal? (soma-nat 1) 1)  ; (+ 1 0)
 (check-equal? (soma-nat 2) 3)  ; (+ 2 1 0)
 (check-equal? (soma-nat 3) 6)) ; (+ 3 2 1 0)
(define (soma-nat n) 0)
```

# Exemplo: soma naturais

Implementação: a partir do modelo.

```
;; Natural -> Natural
;; Soma todos os números naturais de 0 até n
(examples
 (check-equal? (soma-nat 0) 0)
 (check-equal? (soma-nat 1) 1)  ; (+ 1 0)
 (check-equal? (soma-nat 2) 3)  ; (+ 2 1 0)
 (check-equal? (soma-nat 3) 6)) ; (+ 3 2 1 0)
(define (soma-nat n)
  (cond
    [(zero? n) ...]
    [else
      (... n
           (soma-nat (sub1 n)))]))
```

# Exemplo: soma naturais

Implementação: o caso base.

```
;; Natural -> Natural
;; Soma todos os números naturais de 0 até n
(examples
 (check-equal? (soma-nat 0) 0)
 (check-equal? (soma-nat 1) 1)  ; (+ 1 0)
 (check-equal? (soma-nat 2) 3)  ; (+ 2 1 0)
 (check-equal? (soma-nat 3) 6)) ; (+ 3 2 1 0)
(define (soma-nat n)
  (cond
    [(zero? n) 0]
    [else
      (... n
           (soma-nat (sub1 n)))]))
```

# Exemplo: soma naturais

Implementação: o outro caso.

```
;; Natural -> Natural
;; Soma todos os números naturais de 0 até n
(examples
 (check-equal? (soma-nat 0) 0)
 (check-equal? (soma-nat 1) 1)  ; (+ 1 0)
 (check-equal? (soma-nat 2) 3)  ; (+ 2 1 0)
 (check-equal? (soma-nat 3) 6)) ; (+ 3 2 1 0)
(define (soma-nat n)
  (cond
    [(zero? n) 0]
    [else
      (+ n
         (soma-nat (sub1 n)))]))
```

## Exemplo: lista de números

Dado um número natural *n*, defina uma função que devolva `(list 1 2 ... n-1 n)`.

# Exemplo: lista de números

Especificação

```
;; Natural -> ListaDeNúmeros
;; Cria uma lista com os valores 1 2 ... n-1 n
(examples
  (check-equal? (lista-num 0) empty)
  (check-equal? (lista-num 1) (cons 1 empty))
  (check-equal? (lista-num 2) (cons 1 (cons 2 empty)))
  (check-equal? (lista-num 3) (cons 1 (cons 2 (cons 3 empty)))))
(define (lista-num n) empty)
```

# Exemplo: lista de números

Implementação: modelo.

```
;; Natural -> ListaDeNúmeros
;; Cria uma lista com os valores 1 2 ... n-1 n
(examples
  (check-equal? (lista-num 0) empty)
  (check-equal? (lista-num 1) (cons 1 empty))
  (check-equal? (lista-num 2) (cons 1 (cons 2 empty)))
  (check-equal? (lista-num 3) (cons 1 (cons 2 (cons 3 empty)))))
(define (lista-num n)
  (cond
    [(zero? n) ...]
    [else
      (... n
           (lista-num (sub1 n)))]))
```

# Exemplo: lista de números

Implementação: caso base.

```
;; Natural -> ListaDeNúmeros
;; Cria uma lista com os valores 1 2 ... n-1 n
(examples
  (check-equal? (lista-num 0) empty)
  (check-equal? (lista-num 1) (cons 1 empty))
  (check-equal? (lista-num 2) (cons 1 (cons 2 empty)))
  (check-equal? (lista-num 3) (cons 1 (cons 2 (cons 3 empty)))))
(define (lista-num n)
  (cond
    [(zero? n) empty]
    [else
      (... n
           (lista-num (sub1 n)))]))
```

# Exemplo: lista de números

Implementação: o outro caso.

```
;; Natural -> ListaDeNúmeros
;; Cria uma lista com os valores 1 2 ... n-1 n
(examples
  (check-equal? (lista-num 0) empty)
  (check-equal? (lista-num 1) (cons 1 empty))
  (check-equal? (lista-num 2) (cons 1 (cons 2 empty)))
  (check-equal? (lista-num 3) (cons 1 (cons 2 (cons 3 empty)))))
(define (lista-num n)
  (cond
    [(zero? n) empty]
    [else
      (cons-fim n
                (lista-num (sub1 n)))]))
```

# Exemplo: adiciona no final da lista

Especificação para `cons-fim`.

```
;; Número ListaDeNúmeros -> ListaDeNúmeros
;; Adiciona n ao final de lst.
(define (cons-fim n lst) lst)
```

# Exemplo: adiciona no final da lista

Especificação para `cons-fim`.

```
;; Número ListaDeNúmeros -> ListaDeNúmeros
;; Adiciona n ao final de lst.
(examples
  (check-equal? (cons-fim 3 empty)
                (cons 3 empty))
  (check-equal? (cons-fim 4 (cons 3 empty))
                (cons 3 (cons 4 empty)))
  (check-equal? (cons-fim 1 (cons 3 (cons 4 empty)))
                (cons 3 (cons 4 (cons 1 empty)))))
(define (cons-fim n lst) lst)
```

# Exemplo: adiciona no final da lista

Implementação: modelo.

```
;; Número ListaDeNúmeros -> ListaDeNúmeros
;; Adiciona n ao final de lst.
(examples
  (check-equal? (cons-fim 3 empty)
                (cons 3 empty))
  (check-equal? (cons-fim 4 (cons 3 empty))
                (cons 3 (cons 4 empty)))
  (check-equal? (cons-fim 1 (cons 3 (cons 4 empty)))
                (cons 3 (cons 4 (cons 1 empty)))))
(define (cons-fim n lst)
  (cond
    [(empty? lst) ... n]
    [else
     (... (first lst)
          (cons-fim n (rest lst)))]))
```

# Exemplo: adiciona no final da lista

Implementação: caso base.

```
;; Número ListaDeNúmeros -> ListaDeNúmeros
;; Adiciona n ao final de lst.
(examples
  (check-equal? (cons-fim 3 empty)
                (cons 3 empty))
  (check-equal? (cons-fim 4 (cons 3 empty))
                (cons 3 (cons 4 empty)))
  (check-equal? (cons-fim 1 (cons 3 (cons 4 empty)))
                (cons 3 (cons 4 (cons 1 empty)))))
(define (cons-fim n lst)
  (cond
    [(empty? lst) (cons n empty)]
    [else
     (... (first lst)
          (cons-fim n (rest lst)))]))
```

# Exemplo: adiciona no final da lista

Implementação: outro caso.

```
;; Número ListaDeNúmeros -> ListaDeNúmeros
;; Adiciona n ao final de lst.
(examples
  (check-equal? (cons-fim 3 empty)
                (cons 3 empty))
  (check-equal? (cons-fim 4 (cons 3 empty))
                (cons 3 (cons 4 empty)))
  (check-equal? (cons-fim 1 (cons 3 (cons 4 empty)))
                (cons 3 (cons 4 (cons 1 empty)))))
(define (cons-fim n lst)
  (cond
    [(empty? lst) (cons n empty)]
    [else
     (cons (first lst)
           (cons-fim n (rest lst)))]))
```

# Inteiros

Às vezes queremos utilizar um caso base diferente de 0.

Podemos generalizar a definição de número natural para incluir um limite inferior diferente de 0.

# Definição Inteiro

Um número **Inteiro>=a** é

- *a*; ou
- `(add1 n)` onde *n* é um número **Inteiro>=a**

```
(define (fn-para-inteiro>=a n)
  (cond
    [(<= n a) ...]
    [else
      (... n
           (fn-para-inteiro>=a (sub1 n)))]))
```

# Árvores binárias

Como podemos definir uma árvore binária?

Uma **ÁrvoreBinária** é

- `empty`; ou
- `(no Número ÁrvoreBinária ÁrvoreBinária)`, onde `no` é uma estrutura com os campos `valor`, `esq` e `dir`

```
(struct no (valor esq dir) #:transparent)
```

Modelo

```
(define (fn-para-ab t)
  (cond
    [(empty? t) ...]
    [else
      (... (no-valor t)
           (fn-para-ab (no-esq t))
           (fn-para-ab (no-dir t)))]))
```

# Exemplo: altura árvore

Defina uma função que calcule a altura de uma árvore binária. A altura de uma árvore binária é a distância entre a raiz e o seu descendente mais afastado. Uma árvore com um único nó tem altura 0.

# Exemplo: altura árvore

```
;;     t4  3
;;       /   \
;;  t3  4     7  t2
;;     /     / \
;;    3     8   9  t1
;;              /
;;         t0  10
```

```
(define t0 (no 10 empty empty))
(define t1 (no 9 t0 empty))
(define t2 (no 7 (no 8 empty empty) t1))
(define t3 (no 4 (no 3 empty empty) empty))
(define t4 (no 3 t3 t2))
```

```
;; ÁrvoreBinária -> Natural
;; Devolve a altura da árvore binária. A altura de
;; uma árvore binária é a distância da raiz a seu
;; descendente mais afastado. Uma árvore com um
;; único nó tem altura 0.
(examples
  (check-equal? (altura empty) ?)
  (check-equal? (altura t0) 0)
  (check-equal? (altura t1) 1)
  (check-equal? (altura t2) 2)
  (check-equal? (altura t3) 1)
  (check-equal? (altura t4) 3))
(define (altura t)
  (cond
    [(empty? t) ...]
    [else (... (no-valor t)
               (altura (no-esq t))
               (altura (no-dir t)))]))
```

# Exemplo: altura árvore

```
;;     t4  3
;;       /   \
;;  t3  4     7  t2
;;     /     / \
;;    3     8   9  t1
;;              /
;;         t0  10
```

```
(define t0 (no 10 empty empty))
(define t1 (no 9 t0 empty))
(define t2 (no 7 (no 8 empty empty) t1))
(define t3 (no 4 (no 3 empty empty) empty))
(define t4 (no 3 t2 t3))
```

```
;; ÁrvoreBinária -> Natural
;; Devolve a altura da árvore binária. A altura de
;; uma árvore binária é a distância da raiz a seu
;; descendente mais afastado. Uma árvore com um
;; único nó tem altura 0. Uma árvore vazia tem
;; altura -1.
(examples
  (check-equal? (altura empty) -1)
  (check-equal? (altura t0) 0)
  (check-equal? (altura t1) 1)
  (check-equal? (altura t2) 2)
  (check-equal? (altura t3) 1)
  (check-equal? (altura t4) 3))
(define (altura t)
  (cond
    [(empty? t) -1]
    [else (add1 (max (altura (no-esq t))
                     (altura (no-dir t))))]))
```

# Listas aninhadas

# Listas aninhadas

Às vezes é necessário criar uma lista, que contenha outras listas, e estas listas contenham outras listas, etc.

```
> (list 1 4 (list 5 empty (list 2) 9) 10)
'(1 4 (5 () (2) 9) 10)
```

Chamamos este tipo de lista de lista aninhada. Como podemos definir uma lista aninhada?

# Listas aninhadas

Uma **ListaAninhada** é

- `empty`; ou
- `(cons ListaAninhada ListaAninhada)`
- `(cons Número ListaAninhada)`

Modelo

```
(define (fn-para-ladn lst)
  (cond
    [(empty? lst) ...]
    [(list? (first lst))
     (... (fn-para-ladn (first lst))
          (fn-para-ladn (rest lst)))]
    [else
     (... (first lst)
          (fn-para-ladn (rest lst)))]))
```

# Exemplo: soma*

Defina uma função que some todos os números de uma lista aninhada de números.

# Exemplo: soma*

```
;; ListaAninhada -> Número
;; Devolve a soma de todos os elementos de lst.
(examples
 (check-equal? (soma* empty)
               0)
 (check-equal? (soma* (list (list 1 (list empty 3)) (list 4 5) 4 6 7))
               30))
(define (soma* lst)
  (cond
    [(empty? lst) ...]
    [(list? (first lst))
     (... (soma* (first lst))
          (soma* (rest lst)))]
    [else
     (... (first lst)
          (soma* (rest lst)))]))
```

# Exemplo: soma*

```
;; ListaAninhada -> Número
;; Devolve a soma de todos os elementos de lst.
(examples
 (check-equal? (soma* empty)
               0)
 (check-equal? (soma* (list (list 1 (list empty 3)) (list 4 5) 4 6 7))
               30))
(define (soma* lst)
  (cond
    [(empty? lst) 0]
    [(list? (first lst))
     (+ (soma* (first lst))
        (soma* (rest lst)))]
    [else
     (+ (first lst)
        (soma* (rest lst)))]))
```

# Exemplo: aplaina

Defina uma função que aplaine uma lista aninhada, isto é, transforme uma lista aninhada em uma lista sem listas aninhadas com os mesmos elementos e na mesma ordem da lista aninhada.

# Exemplo: aplaina

```
;; ListaAninhada -> ListaDeNúmeros
;; Devolve uma versão não aninhada de lst, isto é, uma lista com os mesmos
;; elementos de lst, mas sem aninhamento.
(examples
 (check-equal? (aplaina empty) empty)
 (check-equal? (aplaina (list (list 1 (list empty 3)) (list 4 5) 4 6 7))
               (list 1 3 4 5 4 6 7)))
(define (aplaina lst)
  (cond
    [(empty? lst) ...]
    [(list? (first lst))
     (... (aplaina (first lst))
          (aplaina (rest lst)))]
    [else
     (... (first lst)
          (aplaina (rest lst)))]))
```

# Exemplo: aplaina

```
;; ListaAninhada -> ListaDeNúmeros
;; Devolve uma versão não aninhada de lst, isto é, uma lista com os mesmos
;; elementos de lst, mas sem aninhamento.
(examples
 (check-equal? (aplaina empty) empty)
 (check-equal? (aplaina (list (list 1 (list empty 3)) (list 4 5) 4 6 7))
               (list 1 3 4 5 4 6 7)))
(define (aplaina lst)
  (cond
    [(empty? lst) empty]
    [(list? (first lst))
     (append (aplaina (first lst))
             (aplaina (rest lst)))]
    [else
     (cons (first lst)
           (aplaina (rest lst)))]))
```

# Limitações

# Limitações

Cada tipo com autorreferência tem um modelo de função que podemos usar como ponto de partida para implementar funções que processam o tipo de dado.

Embora o modelo seja um ponto de partida, em algumas situações ele pode não ser útil.

Considere o problema de verificar se uma lista de números é palíndromo (a lista tem os mesmos elementos quando lida da direita para a esquerda e da esquerda para direita).

Para verificar se `(list 5 4 1 4)` é palíndromo, o modelo sugere verificar se `(list 4 1 4)` é palíndromo.

Como a verificação se `(list 4 1 4)` é palíndromo pode nos ajudar a determinar se `(list 5 4 1 4)` é palíndromo? Ou seja, a solução para o resto pode nos ajustar a compor o resultado para o todo? Não pode...

Considere o problema de verificar se um número natural $n$ é primo (tem exatamente dois divisores distintos, 1 e $n$).

Para verificar se $n = 13$ é primo, o modelo sugere verificar se 12 é primo.

Como a verificação se 12 é primo pode nos ajudar a determinar se 13 é primo? Não pode...

O problema nos dois casos é o mesmo: a solução do problema original não pode ser obtida a partir da solução do subproblema gerado pela decomposição estrutural do dado.

Como fazemos nesse caso? Temos algumas opções:

- Redefinimos o problema de forma que a solução para o subproblema estrutural possa ser usado na construção da solução do problema original;
- Fazemos uma decomposição em subproblema(s) de maneira não estrutural e utilizamos a solução desse(s) subproblema(s) para construir a solução do problema original;
- Criamos uma plano (sequência de etapas) para construir a solução sem necessariamente pensar na decomposição da entrada em subproblemas do mesmo tipo.

Para o problema do número primo, podemos reescrever o problema da seguinte forma: Dado dois números naturais $n$ e $a \leq n$, projete uma função que determine a quantidade de divisores de $n$ que são $\leq a$.

Se temos a quantidade de divisores de $n$ que são $\leq a - 1$, como obtemos a quantidade de divisores de $n$ que são $\leq a$? Somando 1 se $a$ é divisor de $n$.

Como podemos utilizar essa função para determinar se um número $n$ é primo? Com a expressão `(= (num-divisores n n) 2)`

# Número primo

```
;; Natural -> Boolean
;; Produz #t se n é um número primo, isto é, tem exatamente dois divisores distintos (1 e n).
;; Produz #f caso contrário.
(examples
 (check-equal? (primo? 1) #f)
 (check-equal? (primo? 2) #t)
 (check-equal? (primo? 6) #f)
 (check-equal? (primo? 7) #t)
 (check-equal? (primo? 10) #f))
(define (primo? n)
  ;; Natural Natural -> Natural
  ;; Calcula o número de divirores de n que são <= a.
  (define (num-divisores n a)
    (cond
      [(zero? a) 0]
      [(= (remainder n a) 0) (add1 (num-divisores n (sub1 a)))]
      [else (num-divisores n (sub1 a))]))
  (= (num-divisores n n) 2))
```

Para o problema da lista palíndromo, vamos considerar a entrada `(list 4 1 5 1 4)`.

Como podemos obter um subproblema da entrada de maneira que a resposta para o subproblema possa nos ajudar a compor a resposta para o problema original? Removendo o primeiro e último elemento da lista.

Se sabemos que uma lista `lst` sem o primeiro e o último elemento é palíndromo, como determinar se `lst` é palíndromo? Verificando se o primeiro e o último elemento de `lst` são iguais.

# Palíndromo 1

```
;; Lista -> Booleano
;; Produz #t se lst é palindromo, isto é, tem os mesmos elementos
;; quando lida da direita para esquerda e da esquerda para direita.
;; Produz #f caso contrário.
(examples
 (check-equal? (palindromo? empty) #t)
 (check-equal? (palindromo? (list 2)) #t)
 (check-equal? (palindromo? (list 1 2)) #f)
 (check-equal? (palindromo? (list 3 3)) #t)
 (check-equal? (palindromo? (list 3 7 3)) #t)
 (check-equal? (palindromo? (list 3 7 3 3)) #f))
(define (palindromo? lst)
  (cond
    [(empty? lst) #t]
    [(empty? (rest lst)) #t]
    [else (and (equal? (first lst) (last lst))
               (palindromo? (sem-extremos lst)))]))
```

Exercício: Implemente a função `sem-extremos` e revise a função `palindromo?`.

# Decomposição não estrutural

Funções recursivas que operam em subproblemas obtidos pela decomposição estrutural dos dados são chamadas de **funções recursivas estruturais**.

Funções recursivas que operam em subproblemas arbitrários (não estruturais) são chamadas de **funções recursivas generativas**.

O projeto de função recursivas generativas pode requerer um "*insight*" e por isso tentamos primeiramente resolver os problemas com recursão estrutural.

# Plano

Ainda para o problema da lista palíndromo, ao invés de pensarmos em decompor o problema em um subproblema da mesma natureza, podemos pensar em um plano, uma sequencia de etapas que resolva problemas intermediários mas que gerem o resultado que estamos esperando no final.

Por exemplo, podemos primeiramente inverter a lista e depois verificar se a lista de entrada e a lista invertida são iguais.

Note que para este caso precisaríamos projetar duas novas funções. Estas funções poderiam ser implementadas usando recursão estrutural.

# Palíndromo 2

```
;; Lista -> Booleano
;; Produz #t se lst é palindromo, isto é, tem os mesmos elementos
;; quando lida da direita pra esquerda e da esquerda para direita.
;; Produz #f caso contrário.
(examples
 (check-equal? (palindromo? empty) #t)
 (check-equal? (palindromo? (list 2)) #t)
 (check-equal? (palindromo? (list 1 2)) #f)
 (check-equal? (palindromo? (list 3 3)) #t)
 (check-equal? (palindromo? (list 3 7 3)) #t)
 (check-equal? (palindromo? (list 3 7 3 3)) #f))
(define (palindromo? lst)
  (equal? lst (reverse lst)))
```

Note que usamos as funções pré-definidas `equal?` e `reverse`. Como exercício, implemente essas funções.

Revisão

Usamos tipos com autorreferências quando queremos representar dados de tamanhos arbitrários.

- Usamos funções recursivas para processar dados de tipos com autorreferências.

Para ser bem formada, uma definição com autorreferência deve ter:

- Pelo menos um caso base (sem autorreferência): são utilizados para criar os valores iniciais
- Pelo menos um caso com autorreferência: são utilizados para criar novos valores a partir de valores existentes

As vezes é interessante pensar em números inteiros e naturais como sendo compostos e definidos com autorreferência.

Existem dois tipos de recursão: estrutural e generativa.

- A recursão estrutural é aquela feita na decomposição natural do dado (para as partes que são autorreferências na definição do dado).
- A recursão generativa é aquela que não é estrutural.

A recursão estrutural só pode ser utilizada quando a solução do problema pode ser expressa em termos da solução do subproblema estrutural. Para os demais problemas podemos tentar três abordagem:

- Alterar o problema e utilizar recursão estrutural;
- Usar recursão generativa;
- Usar um plano (sequência de etapas).

# Referências

# Referências

Básicas

- Vídeos Self-Reference
- Vídeos Naturals
- Capítulos 8 a 12 do livro HTDP
- Seções 2.3, 2.4 e 3.8 do Guia Racket

Complementares

- Seções 2.1 (2.1.1 - 2.1.3) e 2.2 (2.2.1) do livro SICP
- Seções 3.9 da Referência Racket
- Seção 6.3 do livro TSPL4