

Autorreferência e recursividade

Marco A L Barbosa

malbarbo.pro.br

Departamento de Informática

Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-Compartilhual 4.0 Internacional.

<http://github.com/malbarbo/na-progfun>

Introdução

Projete uma função que some uma sequência de números.

Como representar e processar uma quantidade de dados arbitrária?

- Vamos criar tipos de dados com autorreferência, isto é, tipos de dados que são definidos em termos deles mesmos
- Vamos usar funções recursivas para processar dados com autorreferência

Listas

O tipo de dado com autorreferência mais comum nas linguagens funcionais é a lista.

Vamos tentar criar uma definição para lista de números.

A ideia é criar uma estrutura com dois campos. O primeiro campo representa o primeiro item na lista e o segundo campo representa o restante da lista (que é uma lista).

```
(struct lista (primeiro resto) #:transparent)  
;; Representa uma lista de números  
;; primeiro: Número - primeiro elemento da lista  
;; resto:     Lista - restante da lista
```

Utilizando esta definição, vamos tentar representar uma lista com os valores 4, 2 e 8.

```
(define lst (lista 4 (lista 2 (lista 8 ???))))
```

O problema com esta definição é que ela não tem fim. Uma lista é definida em termos de outra lista, que é definida em termos de outra lista, etc.

Precisamos de uma maneira de criar uma lista diretamente, que não seja em termos de outra lista. Que lista pode ser essa?

A lista vazia.

Uma `ListaDeNúmeros` é um dos valores:

- `(vazia)`; ou
- `(link Número ListaDeNúmeros)`, onde `link` é uma estrutura com dois campos: `primeiro` e `resto`

Definição no Racket

```
(struct vazia () #:transparent)
```

```
(struct link (primeiro resto) #:transparent)
```

```
;; Uma ListaDeNúmeros é um dos valores
```

```
;; - (vazia); ou
```

```
;; - (link Numero ListaDeNúmeros)
```

```
> (define lst1 (link 3 (vazia)))           ; Lista com o 3
> (define lst2 (link 8 (link 7 (vazia)))) ; Lista com o 8 e 7
> lst1
(link 3 (vazia))
> lst2
(link 8 (link 7 (vazia)))
> (link-primeiro lst2)
8
> (link-resto lst2)
(link 7 (vazia))
> (link-resto lst1)
(vazia)
> (link-primeiro (link-resto lst1))
. . link-primeiro: contract violation
  expected: link?
  given: (vazia)
```

```
;; Lista com os elementos 8 e 7
> (define lst2 (link 8 (link 7 (vazia))))
;; Define uma lista a partir de uma lista existente
> (define lst3 (link 4 lst2))
> lst3
(link 4 (link 8 (link 7 (vazia))))
> (link-primeiro lst3)
4
> (link-resto lst3)
(link 8 (link 7 (vazia)))
> (link-primeiro (link-resto lst3))
8
```

Nós vimos anteriormente que a forma do dado de entrada de uma função sugere uma forma para o corpo da função.

Qual é a forma para o corpo de uma função que o tipo da entrada `ListaDeNúmeros` sugere?

Uma condicional com dois casos:

- A lista é vazia
- A lista é um link

Em Racket

```
(define (fn-para-lst lst)
  (cond
    [(vazia? lst) ...]
    [else
     (... (link-primeiro lst)
          (link-resto lst))]))
```

Qual é o tipo do resultado de `(link-primeiro lst)`? Um número, que é um valor atômico.

Qual é o tipo do resultado de `(link-resto lst)`? Uma lista, que é uma união.

Um valor atômico pode ser processado diretamente, mas como processar uma lista? Fazendo análise dos casos...

Vamos fazer uma alteração no modelo `fn-para-lst` e adicionar uma chamada recursiva para processar `(link-resto lst)`. Essa alteração pode parecer meio “mágica” agora, mas ela vai ficar mais clara em breve.

Uma ListaDeNúmeros é um dos valores:

- `(vazia)`; ou
- `(link Número ListaDeNúmeros)`,
onde `link` é uma estrutura com dois
campos: `primeiro` e `resto`

Modelo para funções que a entrada é ListaDeNúmeros

```
(define (fn-para-lst lst)
  (cond
    [(vazia? lst) ...]
    [else
     (... (link-primeiro lst)
          (fn-para-lst (link-resto lst)))]))
```

Quais são as relações entre a definição de `ListaDeNúmeros` e o modelo?

- A definição tem dois casos, o modelo também
- A autorreferência na definição do dado sugere uma chamada recursiva no modelo

Defina uma função que some os valores de uma lista de números.

Especificação

```
;; ListaDeNúmeros -> Número
;; Soma os valores de lst.
(examples
 (check-equal? (soma (vazia)) 0)
 (check-equal? (soma (link 3 (vazia))) 3) ; (+ 3 0)
 (check-equal? (soma (link 2 (link 5 (vazia)))) 7) ; (+ 2 (+ 5 0))
 (check-equal? (soma (link 4 (link 1 (link 3 (vazia))))) 8) ; (+ 4 (+ 1 (+ 3 0)))

(define (soma lst) 0)
```

E agora, como escrevemos a implementação? Vamos partir do modelo que criamos!

Exemplo: soma

```
;; ListaDeNúmeros -> Número
;; Soma os valores de lst.
(examples
 (check-equal? (soma (vazia)) 0)
 (check-equal? (soma (link 3 (vazia))) 3) ; (+ 3 0)
 (check-equal? (soma (link 2 (link 5 (vazia)))) 7)) ; (+ 2 (+ 5 0))
 (check-equal? (soma (link 4 (link 1 (link 3 (vazia))))) 8)) ; (+ 4 (+ 1 (+ 3 0)))
```

```
(define (fn-para-lst lst)
  (cond
    [(vazia? lst) ...]
    [else
     (... (link-primeiro lst)
          (fn-para-lst (link-resto lst)))]))
```

O que fazemos agora?

Mudamos o nome da função tanto na definição quanto na chamada recursiva.

Exemplo: soma

```
;; ListaDeNúmeros -> Número
;; Soma os valores de lst.
(examples
 (check-equal? (soma (vazia)) 0)
 (check-equal? (soma (link 3 (vazia))) 3) ; (+ 3 0)
 (check-equal? (soma (link 2 (link 5 (vazia)))) 7) ; (+ 2 (+ 5 0))
 (check-equal? (soma (link 4 (link 1 (link 3 (vazia))))) 8) ; (+ 4 (+ 1 (+ 3 0)))
```

```
(define (soma lst)
  (cond
    [(vazia? lst) ...]
    [else
     (... (link-primeiro lst)
          (soma (link-resto lst)))]))
```

O que fazemos agora?

Vamos preencher as lagunas. Qual deve ser o resultado quando a lista é vazia? 0.

Exemplo: soma

```
;; ListaDeNúmeros -> Número
;; Soma os valores de lst.
(examples
 (check-equal? (soma (vazia)) 0)
 (check-equal? (soma (link 3 (vazia))) 3) ; (+ 3 0)
 (check-equal? (soma (link 2 (link 5 (vazia)))) 7) ; (+ 2 (+ 5 0))
 (check-equal? (soma (link 4 (link 1 (link 3 (vazia))))) 8) ; (+ 4 (+ 1 (+ 3 0)))
```

```
(define (soma lst)
  (cond
    [(vazia? lst) 0]
    [else
     (... (link-primeiro lst)
          (soma (link-resto lst)))]))
```

O que fazemos agora?

Analisamos o caso em que a lista não é vazia. O modelo está sugerindo fazer a chamada recursiva para o resto da lista.

Exemplo: soma

```
;; ListaDeNúmeros -> Número
;; Soma os valores de lst.
(examples
 (check-equal? (soma (vazia)) 0)
 (check-equal? (soma (link 3 (vazia))) 3) ; (+ 3 0)
 (check-equal? (soma (link 2 (link 5 (vazia)))) 7) ; (+ 2 (+ 5 0))
 (check-equal? (soma (link 4 (link 1 (link 3 (vazia))))) 8) ; (+ 4 (+ 1 (+ 3 0)))

(define (soma lst)
  (cond
   [(vazia? lst) 0]
   [else
    (... (link-primeiro lst)
         (soma (link-resto lst)))]))
```

Aqui vem o ponto crucial! Mesmo a função não estando completa, nós vamos **assumir** que ela produz a resposta correta para o resto da lista.

Exemplo: soma

```
;; ListaDeNúmeros -> Número
;; Soma os valores de lst.
(examples
 (check-equal? (soma (vazia)) 0)
 (check-equal? (soma (link 3 (vazia))) 3) ; (+ 3 0)
 (check-equal? (soma (link 2 (link 5 (vazia)))) 7) ; (+ 2 (+ 5 0))
 (check-equal? (soma (link 4 (link 1 (link 3 (vazia))))) 8) ; (+ 4 (+ 1 (+ 3 0)))
```

```
(define (soma lst)
  (cond
    [(vazia? lst) 0]
    [else
     (... (link-primeiro lst)
          (soma (link-resto lst)))]))
```

No exemplo 2 queremos obter a soma de `(link 3 (vazia))` que é 3. O que temos para compor o resultado?

- 3, que é `(link-primeiro lst)`
- 0, que é `(soma (link-resto lst))`

Como obtemos o resultado que queremos?

```
(+ 3 0)
```

Exemplo: soma

```
;; ListaDeNúmeros -> Número
;; Soma os valores de lst.
(examples
 (check-equal? (soma (vazia)) 0)
 (check-equal? (soma (link 3 (vazia))) 3) ; (+ 3 0)
 (check-equal? (soma (link 2 (link 5 (vazia)))) 7) ; (+ 2 (+ 5 0))
 (check-equal? (soma (link 4 (link 1 (link 3 (vazia))))) 8) ; (+ 4 (+ 1 (+ 3 0)))
```

```
(define (soma lst)
  (cond
    [(vazia? lst) 0]
    [else
     (... (link-primeiro lst)
          (soma (link-resto lst)))]))
```

No exemplo 3 queremos obter a soma de `(link 2 (link 5 (vazia)))` que é 7. O que temos para compor o resultado?

- 2, que é `(link-primeiro lst)`
- 5, que é `(soma (link-resto lst))`

Como obtemos o resultado que queremos?

```
(+ 2 5)
```

Exemplo: soma

```
;; ListaDeNúmeros -> Número
;; Soma os valores de lst.
(examples
 (check-equal? (soma (vazia)) 0)
 (check-equal? (soma (link 3 (vazia))) 3) ; (+ 3 0)
 (check-equal? (soma (link 2 (link 5 (vazia)))) 7) ; (+ 2 (+ 5 0))
 (check-equal? (soma (link 4 (link 1 (link 3 (vazia))))) 8) ; (+ 4 (+ 1 (+ 3 0)))
```

```
(define (soma lst)
  (cond
    [(vazia? lst) 0]
    [else
     (... (link-primeiro lst)
          (soma (link-resto lst)))]))
```

No exemplo 4 queremos obter a soma de
(link 4 (link 1 (link 3 (vazia)))) que é
8. O que temos para compor o resultado?

- 4, que é (link-primeiro lst)
- 4, que é (soma (link-resto lst))

Como obtemos o resultado que queremos?

```
(+ 4 4)
```

Exemplo: soma

Agora que compreendemos como o resultado é formado, podemos completar o corpo da função.

```
;; ListaDeNúmeros -> Número
;; Soma os valores de lst.
(examples
 (check-equal? (soma (vazia)) 0)
 (check-equal? (soma (link 3 (vazia))) 3) ; (+ 3 0)
 (check-equal? (soma (link 2 (link 5 (vazia)))) 7) ; (+ 2 (+ 5 0))
 (check-equal? (soma (link 4 (link 1 (link 3 (vazia))))) 8) ; (+ 4 (+ 1 (+ 3 0)))

(define (soma lst)
  (cond
   [(vazia? lst) 0]
   [else
    (+ (link-primeiro lst)
       (soma (link-resto lst)))]))
```


Verificação: Ok.

Revisão: Ok.

O Racket já vem com listas pré-definidas

- `empty` ao invés de `(vazia)`
- `cons` ao invés de `link`
- `first` ao invés de `link-primeiro`
- `rest` ao invés de `link-resto`

Outras funções pré-definidas (os propósitos são aproximados)

- `list?` verifica se um valor é uma lista
- `empty?` verifica se uma lista é vazia
- `cons?` verifica se uma lista não é vazia

Uma `ListaDeNúmeros` é um dos valores

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

```
;; Modelo de funções para ListaDeNúmeros
(define (fn-para-lst lst)
  (cond
    [(empty? lst) ...]
    [else
     ... (first lst)
     ... (fn-para-lst (rest lst)) ... ]))
```

```
> (define lst1 (cons 3 empty)) ; Lista com o elemento 3
> (define lst2 (cons 8 (cons 7 empty))) ; Lista com 8 e 7
> lst1
'(3)
> lst2
'(8 7)
> (first lst2)
8
> (rest lst2)
'(7)
> (rest (rest lst2))
'()
> (first (rest lst1))
. . first: contract violation
  expected: (and/c list? (not/c empty?))
  given: '()
```

```
;; Lista com os elementos 8 e 7
> (define lst2 (cons 8 (cons 7 empty)))
;; Defini uma lista a partir de uma lista existente
> (define lst3 (cons 4 lst2))
> lst3
'(4 8 7)
> (first lst3)
4
> (rest lst3)
'(8 7)
> (first (rest lst3))
8
```

O Racket oferece uma forma conveniente de criar listas

```
> (list 4 5 6 -2 20)
'(4 5 6 -2 20)
```

Em geral

```
(list <a1> <a2> ... <an>)
```

é equivalente a

```
(cons <a1>
      (cons <a2>
            (cons ...
                  (cons <an> empty) ...))))
```

Defina uma função que verifique se um dado valor está em uma lista de números.

Exemplo: contém

```
;; ListaDeNúmeros Número -> Booleano  
;; Produz #t se v está em lst; #f caso contrário.
```

(examples

```
(check-equal? (contem? empty 3) #f)  
(check-equal? (contem? (cons 3 empty) 3) #t)  
(check-equal? (contem? (cons 3 empty) 4) #f)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 4) #t)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 10) #t)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 8) #f))
```

```
(define (contem? lst v) #f)
```

Iniciamos com a especificação.

Exemplo: contém

```
;; ListaDeNúmeros Número -> Booleano  
;; Produz #t se v está em lst; #f caso contrário.
```

(examples

```
(check-equal? (contem? empty 3) #f)  
(check-equal? (contem? (cons 3 empty) 3) #t)  
(check-equal? (contem? (cons 3 empty) 4) #f)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 4) #t)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 10) #t)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 8) #f))
```

```
(define (fn-para-lst lst)
```

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

Para implementação partimos do modelo.

Exemplo: contém

```
;; ListaDeNúmeros Número -> Booleano  
;; Produz #t se v está em lst; #f caso contrário.
```

(examples

```
(check-equal? (contem? empty 3) #f)  
(check-equal? (contem? (cons 3 empty) 3) #t)  
(check-equal? (contem? (cons 3 empty) 4) #f)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 4) #t)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 10) #t)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 8) #f))
```

```
(define (contem? lst v)  
  (cond  
    [(empty? lst) ... v]  
    [else  
     ... (first lst)  
     ... v  
     ... (contem? (rest lst) v) ... ]))
```

Em seguida ajustamos o nome da função e adicionamos o parâmetro `v` na definição, na chamada recursiva e como valor disponível nos dois casos.

Exemplo: contém

```
;; ListaDeNúmeros Número -> Booleano
;; Produz #t se v está em lst; #f caso contrário
(examples
 (check-equal? (contem? empty 3) #f)
 (check-equal? (contem? (cons 3 empty) 3) #t)
 (check-equal? (contem? (cons 3 empty) 4) #f)
 (check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 4) #t)
 (check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 10) #t)
 (check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 8) #f))
```

```
(define (contem? lst v)
  (cond
    [(empty? lst) #f]
    [else
     ... (first lst)
     ... v
     ... (contem? (rest lst) v) ... ]))
```

Analisando os exemplos definimos o caso em que a lista é vazia.

Exemplo: contém

```
;; ListaDeNúmeros Número -> Booleano  
;; Produz #t se v está em lst; #f caso contrário
```

(examples

```
(check-equal? (contem? empty 3) #f)  
(check-equal? (contem? (cons 3 empty) 3) #t)  
(check-equal? (contem? (cons 3 empty) 4) #f)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 4) #t)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 10) #t)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 8) #f))
```

```
(define (contem? lst v)  
  (cond  
    [(empty? lst) #f]  
    [else  
     ... (first lst)  
     ... v  
     ... (contem? (rest lst) v) ... ]))
```

Agora analisamos o caso em que a `lst` não é vazia. Temos `(first lst)`, `v` e `(contem? (rest lst) v)` (se o resto de `lst` contém `v`). Como combinar esses elementos para determinar `(contem? lst v)` (se `lst` contém `v`)?

Exemplo: contém

```
;; ListaDeNúmeros Número -> Booleano  
;; Produz #t se v está em lst; #f caso contrário
```

(examples

```
(check-equal? (contem? empty 3) #f)  
(check-equal? (contem? (cons 3 empty) 3) #t)  
(check-equal? (contem? (cons 3 empty) 4) #f)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 4) #t)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 10) #t)  
(check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 8) #f))
```

```
(define (contem? lst v)  
  (cond  
    [(empty? lst) #f]  
    [else  
     (if (= v (first lst))  
         #t  
         (contem? (rest lst) v))]))
```

Quando `(first lst)` é igual a `v`, podemos gerar a resposta `#t` diretamente, independente da resposta da chamada recursiva. Caso contrário, a resposta se `lst` contém `v` e a mesma se `(rest lst)` contém `v`, ou seja, a resposta para `(contem? lst v)` é a equivalente a `(contem? (rest lst) v)`.

Exemplo: contém

```
;; ListaDeNúmeros Número -> Booleano
;; Produz #t se v está em lst; #f caso contrário
(examples
 (check-equal? (contem? empty 3) #f)
 (check-equal? (contem? (cons 3 empty) 3) #t)
 (check-equal? (contem? (cons 3 empty) 4) #f)
 (check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 4) #t)
 (check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 10) #t)
 (check-equal? (contem? (cons 4 (cons 10 (cons 3 empty)))) 8) #f))
```

Revisão.

```
(define (contem? lst v)
  (cond
    [(empty? lst) #f]
    [else
     (or (= v (first lst))
         (contem? (rest lst) v))]))
```

Defina uma função que remova todos os número negativos de uma lista de números.

Exemplo: remove negativos

```
;; ListaDeNúmeros -> ListaDeNúmeros
;; Produz uma nova lista removendo os valores negativos de lst.
(examples
 (check-equal? (remove-negativos empty) empty)
 (check-equal? (remove-negativos (cons -1 (cons 2 (cons -3 empty))))
               (cons 2 empty))
 (check-equal? (remove-negativos (cons 3 (cons 4 (cons -2 empty))))
               (cons 3 (cons 4 empty))))

(define (remove-negativos lst) empty)
```

Iniciamos com a especificação.

Exemplo: remove negativos

```
;; ListaDeNúmeros -> ListaDeNúmeros
;; Produz uma nova lista removendo os valores negativos de lst.
(examples
 (check-equal? (remove-negativos empty) empty)
 (check-equal? (remove-negativos (cons -1 (cons 2 (cons -3 empty))))
               (cons 2 empty))
 (check-equal? (remove-negativos (cons 3 (cons 4 (cons -2 empty))))
               (cons 3 (cons 4 empty))))

(define (remove-negativos lst)
  (cond
   [(empty? lst) ...]
   [else
    ... (first lst)
    ... (remove-negativos (rest lst)) ... ]))
```

Para implementação
partimos do modelo e
ajustamos o nome.

Exemplo: remove negativos

```
;; ListaDeNúmeros -> ListaDeNúmeros
;; Produz uma nova lista removendo os valores negativos de lst.
(examples
 (check-equal? (remove-negativos empty) empty)
 (check-equal? (remove-negativos (cons -1 (cons 2 (cons -3 empty))))
               (cons 2 empty))
 (check-equal? (remove-negativos (cons 3 (cons 4 (cons -2 empty))))
               (cons 3 (cons 4 empty))))

(define (remove-negativos lst)
  (cond
   [(empty? lst) empty]
   [else
    ... (first lst)
    ... (remove-negativos (rest lst)) ... ]))
```

Analisando os exemplos
definimos o caso em que a
lista é vazia.

Exemplo: remove negativos

```
;; ListaDeNúmeros -> ListaDeNúmeros
;; Produz uma nova lista removendo os valores negativos de lst.
(examples
 (check-equal? (remove-negativos empty) empty)
 (check-equal? (remove-negativos (cons -1 (cons 2 (cons -3 empty))))
               (cons 2 empty))
 (check-equal? (remove-negativos (cons 3 (cons 4 (cons -2 empty))))
               (cons 3 (cons 4 empty))))

(define (remove-negativos lst)
  (cond
    [(empty? lst) empty]
    [else
     (if (< (first lst) 0)
         (remove-negativos (rest lst))
         (cons (first lst)
               (remove-negativos (rest lst)))))]))
```

Analisando os exemplos
definimos o caso em que a
lista não é vazia.

Verificação: Ok.

Revisão: exercício.

Defina uma função que soma um valor x em cada elemento de uma lista de números.

Exemplo: soma x

```
;; ListaDeNúmeros Número -> ListaDeNúmeros
;; Produz uma nova lista somando x a cada elemento de lst.
(examples
 (check-equal? (soma-x empty 4)
               empty)
 (check-equal? (soma-x (cons 4 (cons 2 empty)) 5)
               (cons 9 (cons 7 empty)))
 (check-equal? (soma-x (cons 3 (cons -1 (cons 4 empty))) -2)
               (cons 1 (cons -3 (cons 2 empty)))))

(define (soma-x lst x) empty)
```

Iniciamos com a especificação.

Exemplo: soma x

```
;; ListaDeNúmeros Número -> ListaDeNúmeros
;; Produz uma nova lista somando x a cada elemento de lst.
(examples
 (check-equal? (soma-x empty 4)
               empty)
 (check-equal? (soma-x (cons 4 (cons 2 empty)) 5)
               (cons 9 (cons 7 empty)))
 (check-equal? (soma-x (cons 3 (cons -1 (cons 4 empty))) -2)
               (cons 1 (cons -3 (cons 2 empty)))))

(define (soma-x lst x)
  (cond
   [(empty? lst) ... x]
   [else
    ... (first lst)
    ... x
    ... (soma-x (rest lst) x) ... ]))
```

Para implementação partimos do modelo e ajustamos o nome e adicionamos o parâmetro x.

Exemplo: soma x

```
;; ListaDeNúmeros Número -> ListaDeNúmeros
;; Produz uma nova lista somando x a cada elemento de lst.
(examples
 (check-equal? (soma-x empty 4)
               empty)
 (check-equal? (soma-x (cons 4 (cons 2 empty)) 5)
               (cons 9 (cons 7 empty)))
 (check-equal? (soma-x (cons 3 (cons -1 (cons 4 empty))) -2)
               (cons 1 (cons -3 (cons 2 empty)))))

(define (soma-x lst x)
  (cond
   [(empty? lst) empty]
   [else
    ... (first lst)
    ... x
    ... (soma-x (rest lst) x) ... ]))
```

Analisando os exemplos definimos o caso em que a lista é vazia.

Exemplo: soma x

```
;; ListaDeNúmeros Número -> ListaDeNúmeros
;; Produz uma nova lista somando x a cada elemento de lst.
(examples
 (check-equal? (soma-x empty 4)
               empty)
 (check-equal? (soma-x (cons 4 (cons 2 empty)) 5)
               (cons 9 (cons 7 empty)))
 (check-equal? (soma-x (cons 3 (cons -1 (cons 4 empty))) -2)
               (cons 1 (cons -3 (cons 2 empty)))))

(define (soma-x lst x)
  (cond
   [(empty? lst) empty]
   [else
    (cons (+ x (first lst))
          (soma-x (rest lst) x))]))
```

Analisando os exemplos definimos o caso em que a lista não é vazia.

Verificação: Ok.

Revisão: Ok.

Projete uma função que junte todos os elementos de uma lista de strings (não vazias) separando-os com “, ” ou/e “ e ”, de acordo com a gramática do Português.

Exemplos: junta com “,” e “e”

```
;; ListaDeStrings -> String
;; Parece difícil escrever o propósito...
;; Vamos fazer os exemplos primeiro.
(define (junta-vmgula-e lst) "")
```

Exemplos

```
(junta-vmgula-e empty) → ""
```

```
(junta-vmgula-e (list "maça")) → "maça"
```

```
(junta-vmgula-e (list "banana" "maça")) → "banana e maça"
```

```
(junta-vmgula-e (list "mamão" "banana" "maça")) → "mamão, banana e maça"
```

```
(junta-vmgula-e (list "aveia" "mamão" "banana" "maça")) → "aveia, mamão, banana e maça"
```

Em todos os exemplos as respostas são calculadas da mesma forma? Não! Os três primeiros exemplos tem uma forma específica, que não é recursiva.

Então precisamos criar três casos base.

Exemplos: junta com “,” e “e”

```
;; ListaDeStrings -> String
;; Se a lista é vazia, devolve "".
;; Se a lista tem apenas um elemento, devolve
;; esse elemento.
;; Senão, junta as strings de lst, separando-as
;; com ", ", com exceção da última string, que
;; é separada com " e ".
```

(examples

```
(check-equal? (junta-vmrgula-e empty) "")
```

```
(check-equal?
 (junta-vmrgula-e (list "maça")) "maça")
```

```
(check-equal?
 (junta-vmrgula-e (list "mamão" "banana"
                       "maça"))
 "mamão, banana e maça")
```

```
(check-equal?
 (junta-vmrgula-e (list "aveia" "mamão"
                       "banana" "maça"))
 "aveia, mamão, banana e maça")
```

```
(define (junta-vmrgula-e lst) "")
```

Implementação

```
(define (junta-vmrgula-e lst)
```

```
  (cond
```

```
    [(empty? lst)
```

```
     ""]
```

```
    [(empty? (rest lst))
```

```
     (first lst)]
```

```
    [(empty? (rest (rest lst)))]
```

```
     (string-append (first lst)
```

```
                    " e "
```

```
                    (second lst)))]
```

```
    [else
```

```
     (string-append (first lst)
```

```
                    ", "
```

```
                    (junta-vmrgula-e (rest lst)))]))
```

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?

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)))]))
```



```
;; 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
```

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

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)))]))
```

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

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.

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)))]))
```

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 nós 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 diviroides 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?`.

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.

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 sequência 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.

```
;; 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 solução do problema pode ser expressa em termos da solução do problema estrutural. Para os demais problemas podemos tentar três abordagens:

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

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](#)