Autorreferência

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

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 um valor 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 - é o primeiro elemento
;;                     da lista
;;  resto:    Lista  - é o restante da lista
```

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

```
(define ldn (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)
```

Listas

```
> (define ldn1 (link 3 (vazia)))          ; Lista com o 3
> (define ldn2 (link 8 (link 7 (vazia)))) ; Lista com o 8 e 7
> ldn1
(link 3 (vazia))
> ldn2
(link 8 (link 7 (vazia)))
> (link-primeiro ldn2)
8
> (link-resto ldn2)
(link 7 (vazia))
> (link-resto ldn1)
(vazia)
> (link-primeiro (link-resto ldn1))
. . link-primeiro: contract violation
  expected: link?
  given: (vazia)
```

```
;; Lista com os elementos 8 e 7
> (define ldn2 (link 8 (link 7 (vazia))))
;; Define uma lista a partir de uma lista existente
> (define ldn3 (link 4 ldn2))
> ldn3
(link 4 (link 8 (link 7 (vazia))))
> (link-primeiro ldn3)
4
> (link-resto ldn3)
(link 8 (link 7 (vazia)))
> (link-primeiro (link-resto ldn3))
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-ldn ldn)
  (cond
    [(vazia? ldn) ...]
    [else
     (... (link-primeiro ldn)
          (link-resto ldn))]))
```

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

Qual o resultado de `(link-rest ldn)`? 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-ldn` e adicionar uma chamada recursiva para processar `(link-resto ldn)`. Essa alteração pode parecer meio "mágica" no início, mas ela vai ficar mais clara no futuro.

Modelo para funções que a entrada é ListaDeNúmeros

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

Observe a relação com a definição de `ListaDeNúmeros`

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

Exemplo: soma

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

Exemplo: soma

Especificação

```
;; ListaDeNúmeros -> Número
;; Soma os valores de ldn.
(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 ldn) 0)
```

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

Exemplo: soma

```
;; ListaDeNúmeros -> Número
;; Soma os valores de ldn.
(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-ldn ldn)
  (cond
    [(vazia? ldn) ...]
    [else
     (... (link-primeiro ldn)
          (fn-para-ldn (link-resto ldn)))]))
```

O que fazemos agora?

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

Exemplo: soma

```
;; ListaDeNúmeros -> Número
;; Soma os valores de ldn.
(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 ldn)
  (cond
    [(vazia? ldn) ...]
    [else
     (... (link-primeiro ldn)
          (soma (link-resto ldn)))]))
```

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

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.

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

Agora temos o primeiro elemento e a soma do resto da lista, como podemos combinar esses dois valores para obter a soma dos elementos da lista? Somando.

Exemplo: soma

```
;; ListaDeNúmeros -> Número
;; Soma os valores de ldn.
(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 ldn)
  (cond
    [(vazia? ldn) 0]
    [else
     (+ (link-primeiro ldn)
        (soma (link-resto ldn)))]))
```

Verificação: Ok.

Revisão: exercício.

Listas

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 (os propósitos são aproximados)

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

Uma **ListaDeNúmeros** é um dos valores

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

```
;; Modelo para funções com entrada ListaDeNúmeros
#;
(define (fn-para-ldn ldn)
  (cond
    [(empty? ldn) ...]
    [else ... (first ldn)
          ... (fn-para-ldn (rest ldn)) ... ]))
```

Listas

```
> (define ldn1 (cons 3 empty)) ; Lista com o elemento 3
> (define ldn2 (cons 8 (cons 7 empty))) ; Lista com 8 e 7
> ldn1
'(3)
> ldn2
'(8 7)
> (first ldn2)
8
> (rest ldn2)
'(7)
> (rest (rest ldn2))
'()
> (first (rest ldn1))
. . first: contract violation
  expected: (and/c list? (not/c empty?))
    given: '()
```

```
;; Lista com os elementos 8 e 7
> (define ldn2 (cons 8 (cons 7 empty)))
;; Defini uma lista a partir de uma lista existente
> (define ldn3 (cons 4 ldn2))
> ldn3
'(4 8 7)
> (first ldn3)
4
> (rest ldn3)
'(8 7)
> (first (rest ldn3))
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) ...)))
```

Exemplo: contém

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 ldn; #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? ldn v) #f)
```

Iniciamos com a especificação.

Exemplo: contém

Para implementação partimos do modelo.

```
;; ListaDeNúmeros Número -> Booleano
;; Produz #t se v está em ldn; #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-ldn ldn)
  (cond
    [(empty? ldn) ...]
    [else
      ... (first ldn)
      ... (fn-para-ldn (rest ldn)) ... ]))
```

Exemplo: contém

```
;; ListaDeNúmeros Número -> Booleano
;; Produz #t se v está em ldn; #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? ldn v)
  (cond
    [(empty? ldn) ... v]
    [else
      ... (first ldn)
      ... (contem? (rest ldn) v) ... ]))
```

Em seguida ajeitamos o nome da função e adicionamos o parâmetro v.

Exemplo: contém

```
;; ListaDeNúmeros Número -> Booleano
;; Produz #t se v está em ldn; #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? ldn v)
  (cond
    [(empty? ldn) #f]
    [else
      ... (first ldn)
      ... (contem? (rest ldn) v) ... ]))
```

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

Exemplo: contém

```
;; ListaDeNúmeros Número -> Booleano
;; Produz #t se v está em ldn; #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? ldn v)
  (cond
    [(empty? ldn) #f]
    [else
     (if (= v (first ldn))
         #t
         (contem? (rest ldn) v))]))
```

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

Exemplo: contém

Revisão.

```
;; ListaDeNúmeros Número -> Booleano
;; Produz #t se v está em ldn; #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? ldn v)
  (cond
    [(empty? ldn) #f]
    [else
     (or (= v (first ldn))
         (contem? (rest ldn) v))]))
```

Exemplo: remove negativos

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

Exemplo: remove negativos (especificação)

Iniciamos com a especificação.

```
;; ListaDeNúmeros -> ListaDeNúmeros
;; Produz uma nova lista removendo os valores negativos de ldn.
(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 ldn) empty)
```

Exemplo: remove negativos

```
;; ListaDeNúmeros -> ListaDeNúmeros
;; Produz uma nova lista removendo os valores negativos de ldn.
(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 ldn)
  (cond
    [(empty? ldn) ...]
    [else
      ... (first ldn)
      ... (remove-negativos (rest ldn)) ... ]))
```

Para implementação partimos do modelo e ajustamos o nome.

Exemplo: remove negativos

```
;; ListaDeNúmeros -> ListaDeNúmeros
;; Produz uma nova lista removendo os valores negativos de ldn.
(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 ldn)
  (cond
    [(empty? ldn) empty]
    [else
      ... (first ldn)
      ... (remove-negativos (rest ldn)) ... ]))
```

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

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

Verificação: Ok.

Revisão: exercício.

Exemplo: soma x

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

Exemplo: soma x (especificação)

```
;; ListaDeNúmeros Número -> ListaDeNúmeros
;; Produz uma nova lista somando x a cada elemento de ldn.
(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 ldn x) empty)
```

Iniciamos com a especificação.

Exemplo: soma x

```
;; ListaDeNúmeros Número -> ListaDeNúmeros
;; Produz uma nova lista somando x a cada elemento de ldn.
(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 ldn x)
  (cond
    [(empty? ldn) ... x]
    [else
      ... (first ldn)
      ... (soma-x (rest ldn) 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 ldn.
(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 ldn x)
  (cond
    [(empty? ldn) empty]
    [else
      ... (first ldn)
      ... (soma-x (rest ldn) 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 ldn.
(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 ldn x)
  (cond
    [(empty? ldn) empty]
    [else
     (cons (+ x (first ldn))
           (soma-x (rest ldn) x))]))
```

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

Verificação: Ok.

Revisão: Ok.

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 ao 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 3) 6)) ; (+ 3 (+ 2 (+ 1 0)))
(define (soma-nat n) 0)
```

Implementação: 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 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 3) 6)) ; (+ 3 (+ 2 (+ 1 0)))
(define (soma-nat n)
  (cond
    [(zero? n) 0]
    [else
      (... n
           (soma-nat (sub1 n)))]))
```

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 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 a lista `(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 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 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 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 (veja o vídeo da aula para entender como chegamos nesse código).

```
;; 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 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 ldn.
(define (cons-fim n ldn) ldn)
```

Exemplo: adiciona no final da lista

Especificação para `cons-fim`.

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

Exemplo: adiciona no final da lista

Implementação: modelo.

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

Exemplo: adiciona no final da lista

Implementação: caso base.

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

Exemplo: adiciona no final da lista

Implementação: outro caso.

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

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**

Modelo

```
(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 **ÁrvoreBináriaDeNúmeros** é

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

Modelo

```
(define (fn-para-abdn t)
  (cond
    [(empty? t) ...]
    [else
      (... (no-valor t)
           (fn-para-abdn (no-esq t))
           (fn-para-abdn (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
;; ÁrvoreBináriaDeNúmeros -> Natural
(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
;; ÁrvoreBináriaDeNúmeros -> Natural
(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

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?

Uma **ListaAninhadaDeNúmeros** é

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

```
;; Modelo
#;
(define (fn-para-ladn ldn)
  (cond
    [(empty? ldn) ...]
    [(list? (first ldn))
     (... (fn-para-ladn (first ldn))
          (fn-para-ladn (rest ldn)))]
    [else
     (... (first ldn)
          (fn-para-ladn (rest ldn)))]))
```

Exemplo: soma*

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

Exemplo: soma*

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

Exemplo: soma*

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

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

```
;; ListaAninhadaDeNúmeros -> ListaDeNúmeros
;; Devolve uma versão não aninhada de ldn, isto é, uma lista com os mesmos
;; elementos de ldn, 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 ldn)
  (cond
    [(empty? ldn) ...]
    [(list? (first ldn))
     (... (aplaina (first ldn))
          (aplaina (rest ldn)))]
    [else
     (... (first ldn)
          (aplaina (rest ldn)))]))
```

Exemplo: aplaina

```
;; ListaAninhadaDeNúmeros -> ListaDeNúmeros
;; Devolve uma versão não aninhada de ldn, isto é, uma lista com os mesmos
;; elementos de ldn, 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 ldn)
  (cond
    [(empty? ldn) empty]
    [(list? (first ldn))
     (append (aplaina (first ldn))
             (aplaina (rest ldn)))]
    [else
     (cons (first ldn)
           (aplaina (rest ldn)))]))
```

Observações finais

Observações finais

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

- Usamos funções recursivas para processar dados 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.

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
- Capítulo 9.3 do livro HTDP