Próximo:Matrizes, Acima:Vetores de matrizes de sequências [Conteúdo][Índice]

6.1 Sequências

Esta seção descreve funções que aceitam qualquer tipo de sequência.

Função: *objeto* sequencep

Esta função retorna `t`se o *objeto* for uma lista, vetor, string, bool-vector ou char-table, `nil`caso contrário. Veja também `seqp`abaixo.

Função: *sequência* de comprimento

Esta função retorna o número de elementos em *sequência* . A função sinaliza o `wrong-type-argument`erro se o argumento não for uma sequência ou for uma lista pontilhada; sinaliza o `circular-list`erro se o argumento for uma lista circular. Para uma tabela de caracteres, o valor retornado é sempre um a mais do que o código de caractere máximo do Emacs.

Consulte Definição de safe-length , para a função relacionada `safe-length`.

```
(comprimento '(1 2 3))
    ⇒ 3
(comprimento ())
    ⇒ 0
(comprimento "foobar")
    ⇒ 6
(comprimento [1 2 3])
    ⇒ 3
(comprimento (fazer-bool-vetor 5 nil))
    ⇒ 5
```

Veja também `string-bytes`, em Representações de Texto .

Se você precisar calcular a largura de uma string em exibição, deve usar `string-width`(consulte Tamanho do texto exibido), não `length`, pois `length`conta apenas o número de caracteres, mas não considera a largura de exibição de cada caractere.

Função: *índice de sequência* elt

Esta função retorna o elemento da *sequência* indexado por *index* . Os valores legítimos de *índice* são inteiros que variam de 0 a um a menos que o comprimento da *sequência* . Se a *sequência* for uma lista, os valores fora do intervalo se comportarão como para `nth`. Consulte Definição de nth . Caso contrário, valores fora do intervalo acionarão um `args-out-of-range`erro.

```
(elt [1 2 3 4] 2)
     ⇒ 3
(elt '(1 2 3 4) 2)
     ⇒ 3
;; Usamos stringpara mostrar claramente qual caractere eltretorna.
(string (elt "1234" 2))
     ⇒ "3"
```

```
(elt [1 2 3 4] 4)
     erro→ Args fora do intervalo: [1 2 3 4], 4
(elt [1 2 3 4] -1)
     erro→ Args fora do intervalo: [1 2 3 4], -1
```

Essa função generaliza `aref`(consulte Funções de matriz) e `nth`(consulte Definição de nth).

Função: seqüência de cópia *seqr*

Esta função retorna uma cópia de *seqr* , que deve ser uma sequência ou um registro. A cópia é do mesmo tipo de objeto que o original e possui os mesmos elementos na mesma ordem. No entanto, se *seqr* estiver vazio, como uma string ou um vetor de comprimento zero, o valor retornado por esta função pode não ser uma cópia, mas um objeto vazio do mesmo tipo e idêntico a *seqr* .

Armazenar um novo elemento na cópia não afeta o *seqr* original e vice-versa. No entanto, os elementos da cópia não são cópias; eles são idênticos (`eq`) aos elementos do original. Portanto, as alterações feitas nesses elementos, conforme encontradas na cópia, também são visíveis no original.

Se o argumento for uma string com propriedades de texto, a lista de propriedades na cópia será uma cópia, não compartilhada com a lista de propriedades do original. No entanto, os valores reais das propriedades são compartilhados. Consulte Propriedades de texto .

Esta função não funciona para listas pontilhadas. Tentar copiar uma lista circular pode causar um loop infinito.

Veja também `appendem` Construindo Listas , `concatem` Criando Strings e `vconcatem` Funções Vetoriais , para outras formas de copiar sequências.

```
(setq bar (lista 1 2))
     ⇒ (1 2)
(setq x (vetor 'foo bar))
     ⇒ [foo (1 2)]
(setq y (sequência de cópia x))
     ⇒ [foo (1 2)]


(eq xy)
     ⇒ nada
(igual a xy)
     ⇒ t
(eq (elt x 1) (elt y 1))
     ⇒ t


;; Substituir um elemento de uma sequência.
(recurso x 0 'quux)
x ⇒ [quux (1 2)]
y ⇒ [foo (1 2)]


;; Modificando o interior de um elemento compartilhado.
(setcar (aref x 1) 69)
x ⇒ [quux (69 2)]
y ⇒ [foo (69 2)]
```

Função: *sequência* reversa

Esta função cria uma nova sequência cujos elementos são os elementos de *sequence* , mas na ordem inversa. *A sequência de* argumentos original *não* é alterada. Observe que as tabelas de caracteres não podem ser revertidas.

```
(setq x '(1 2 3 4))
     ⇒ (1 2 3 4)
(inverter x)
     ⇒ (4 3 2 1)
x
     ⇒ (1 2 3 4)
(conjunto x [1 2 3 4])
     ⇒ [1 2 3 4]
(inverter x)
     ⇒ [4 3 2 1]
x
     ⇒ [1 2 3 4]
(setq x "xyzzy")
     ⇒ "xyzzy"
(inverter x)
     ⇒ "yzzyx"
x
     ⇒ "xyzzy"
```

Função: *sequência* inversa

Esta função inverte a ordem dos elementos da *sequência* . Ao contrário da *sequência*`reverse` original pode ser modificada.

Por exemplo:

```
(setq x (lista 'a 'b 'c))
     ⇒ (abc)
x
     ⇒ (abc)
(nreverso x)
     ⇒ (cba)
;; A célula de contras que era a primeira agora é a última.
x
     ⇒ (a)
```

Para evitar confusão, geralmente armazenamos o resultado de `nreverse` back na mesma variável que continha a lista original:

```
(setq x (nreverse x))
```

Aqui está o `nreverse`nosso exemplo favorito, `(a b c)`, apresentado graficamente:

```
Cabeçalho da lista original:                        Lista invertida:
 ------------ ------------ ------------
| carro | cdr | | carro | cdr | | carro | cdr |
| um | nada |<-- | b | o |<-- | c | o |
| | | | | | | | | | | |
 ------------ | --------- | - | -------- | -
```

```
                    | | | |
                     ------------ ------------
```

Para o vetor, é ainda mais simples porque você não precisa de setq:

```
(setq x (sequência de cópia [1 2 3 4]))
     ⇒ [1 2 3 4]
(nreverso x)
     ⇒ [4 3 2 1]
x
     ⇒ [4 3 2 1]
```

Observe que, diferentemente `reverse`de , esta função não funciona com strings. Embora você possa alterar dados de string usando `aset`, é altamente recomendável tratar strings como imutáveis, mesmo quando são mutáveis. Veja Mutabilidade .

Função: *predicado de sequência de* classificação

Esta função classifica a *sequência de* forma estável. Observe que esta função não funciona para todas as sequências; ele pode ser usado apenas para listas e vetores. Se a *sequência* for uma lista, ela será modificada de forma destrutiva. Esta função retorna a *sequência* ordenada e compara os elementos usando o *predicado* . Uma classificação estável é aquela em que os elementos com chaves de classificação iguais mantêm sua ordem relativa antes e depois da classificação. A estabilidade é importante quando ordenações sucessivas são usadas para ordenar elementos de acordo com diferentes critérios.

O predicado do argumento deve ser uma função que aceita dois argumentos. É chamado com dois elementos de *sequência* . Para obter uma classificação de ordem crescente, o *predicado* deve retornar non- `nil`se o primeiro elemento for “menor” que o segundo, ou `nil`se não for.

O predicado da função de comparação deve fornecer resultados confiáveis para qualquer par de argumentos, pelo menos em uma única chamada para `sort`. Deve ser *antisimétrica* ; isto é, se *a* for menor que *b* , *b* não deve ser menor que *a* . Deve ser *transitivo* — isto é, se *a* é menor que *b* e *b* é menor que *c* , então *a* deve ser menor que *c* . Se você usar uma função de comparação que não atende a esses requisitos, o resultado `sort`é imprevisível.

O aspecto destrutivo das `sort`listas for é que ele reorganiza as células cons que formam a *sequência* alterando os CDRs . Uma função de classificação não destrutiva criaria novas células contra para armazenar os elementos em sua ordem de classificação. Se você deseja fazer uma cópia classificada sem destruir o original, copie-a primeiro com `copy-sequence`e depois classifique.

A ordenação não altera os CAR s das células cons em *seqüência* ; a célula cons que originalmente continha o elemento `a`em *sequência* ainda tem `a`em seu CAR após a ordenação, mas agora aparece em uma posição diferente na lista devido à mudança de CDR s. Por exemplo:

```
(setq nums (lista 1 3 2 6 5 4 0))
     ⇒ (1 3 2 6 5 4 0)
(ordenar números #'<)
     ⇒ (0 1 2 3 4 5 6)
números
     ⇒ (1 2 3 4 5 6)
```

Aviso : Observe que a lista `nums`não contém mais 0; esta é a mesma célula de contras que era antes, mas não é mais a primeira da lista. Não assuma que uma variável que anteriormente continha o argumento agora contém toda a lista ordenada! Em vez disso, salve o resultado `sort`e use isso. Na maioria das vezes, armazenamos o resultado de volta na variável que continha a lista original:

```
(setq nums (sort nums #'<))
```

Para uma melhor compreensão do que é classificação estável, considere o seguinte exemplo de vetor. Após a ordenação, todos os itens cujo `car`é 8 são agrupados no início de `vector`, mas sua ordem relativa é preservada. Todos os itens cujo `car`é 9 são agrupados no final de `vector`, mas sua ordem relativa também é preservada:

```
(conjunto
  vetor
  (vetor '(8 . "xxx") '(9 . "aaa") '(8 . "bbb") '(9 . "zzz")
         '(9 . "ppp") '(8 . "ttt") '(8 . "eee") '(9 . "fff")))
     ⇒ [(8 . "xxx") (9 . "aaa") (8 . "bbb") (9 . "zzz")
        (9 . "ppp") (8 . "ttt") (8 . "eee") (9 . "fff")]
(vetor de ordenação (lambda (xy) (< (carro x) (carro y))))
     ⇒ [(8 . "xxx") (8 . "bbb") (8 . "ttt") (8 . "eee")
        (9 . "aaa") (9 . "zzz") (9 . "ppp") (9 . "fff")]
```

Consulte Classificação , para obter mais funções que realizam classificação. Veja `documentation`em Acessando a Documentação , para um exemplo útil de `sort`.

O`sequência`library fornece as seguintes macros e funções de manipulação de sequência adicionais, prefixadas com `seq-`. Para usá-los, você deve primeiro carregar o`sequência`biblioteca.

Todas as funções definidas nesta biblioteca estão livres de efeitos colaterais; isto é, eles não modificam nenhuma sequência (lista, vetor ou string) que você passa como argumento. Salvo indicação em contrário, o resultado é uma sequência do mesmo tipo que a entrada. Para as funções que recebem um predicado, isso deve ser uma função de um argumento.

O`sequência`biblioteca pode ser estendida para trabalhar com tipos adicionais de estruturas de dados sequenciais. Para isso, todas as funções são definidas usando `cl-defgeneric`. Consulte Funções genéricas para obter mais detalhes sobre como usar `cl-defgeneric`para adicionar extensões.

Função: ***índice de sequência*** **seq-elt**

Essa função retorna o elemento da *sequência no índice* especificado , que é um inteiro cujo intervalo de valores válidos é de zero a um a menos que o comprimento da *sequência* . Para valores fora do intervalo em tipos de sequência internos, `seq-elt`se comporta como `elt`. Para obter detalhes, consulte Definição de elt .

```
(seq-elt [1 2 3 4] 2)
⇒ 3
```

`seq-elt`retorna lugares configuráveis usando `setf` (consulte Configurando Variáveis Generalizadas).

```
(configurar vec [1 2 3 4])
(setf (seq-elt vec 2) 5)
vec
⇒ [1 2 5 4]
```

Função: *sequência* de comprimento seq

Esta função retorna o número de elementos em *sequência* . Para tipos de sequência internos, `seq-length`se comporta como `length`. Consulte Definição de comprimento .

Função: *objeto* seqp

Esta função retorna um *objeto*`nil` não- se é uma sequência (uma lista ou array), ou qualquer tipo adicional de sequência definida via `sequência`funções genéricas. Esta é uma variante extensível do `sequencep`.

```
(seqp [1 2])
⇒ t
(seqp 2)
⇒ nada
```

Função: *sequência* seq-drop n

Esta função retorna todos, exceto os primeiros *n* (um inteiro) elementos de *sequence* . Se *n* for negativo ou zero, o resultado é *sequência* .

```
(seq-drop [1 2 3 4 5 6] 3)
⇒ [4 5 6]
(seq-drop "olá mundo" -4)
⇒ "olá mundo"
```

Função: *sequência* seq-take n

Esta função retorna os primeiros *n* (um inteiro) elementos de *sequence* . Se *n* for negativo ou zero, o resultado será `nil`.

```
(seq-take '(1 2 3 4) 3)
⇒ (1 2 3)
(seq-take [1 2 3 4] 0)
⇒ []
```

Função: *sequência de predicados* seq-take-while

Esta função retorna os membros da *sequência* em ordem, parando antes do primeiro para o qual o *predicado* retorna `nil`.

```
(seq-take-while (lambda (elt) (> elt 0)) '(1 2 3 -1 -2))
⇒ (1 2 3)
(seq-take-while (lambda (elt) (> elt 0)) [-1 4 6])
⇒ []
```

Função: *sequência de predicados* seq-drop-while

Esta função retorna os membros da *sequência* em ordem, começando pelo primeiro para o qual o *predicado* retorna `nil`.

```
(seq-drop-while (lambda (elt) (> elt 0)) '(1 2 3 -1 -2))
⇒ (-1 -2)
(seq-drop-while (lambda (elt) (< elt 0)) [1 4 6])
⇒ [1 4 6]
```

Função: ***sequência de funções*** **seq-do**

Essa função aplica a *função* a cada elemento da *sequência* por sua vez (presumivelmente para efeitos colaterais) e retorna a *sequência* .

Função: ***sequência da função*** **seq-map**

Esta função retorna o resultado da aplicação da *função* a cada elemento da *sequência* . O valor retornado é uma lista.

```
(seq-map #'1+'(2 4 6))
⇒ (3 5 7)
(seq-map #'symbol-name [foo bar])
⇒ ("foo" "bar")
```

Função: ***sequência*** **de função indexada pelo mapa seq**

Esta função retorna o resultado da aplicação da *função* a cada elemento da *sequência* e seu índice dentro de *seq* . O valor retornado é uma lista.

```
(seq-map-indexed (lambda (elt idx)
                   (lista idx elt))
                 '(abc))
⇒ ((0a) (1b) (2c))
```

Função: função seq-mapn ***e sequências de descanso***

Esta função retorna o resultado da aplicação da *função* a cada elemento das *sequências* . A aridade (veja subr-aridade) da *função* deve corresponder ao número de sequências. O mapeamento para no final da sequência mais curta e o valor retornado é uma lista.

```
(seq-mapn #'+ '(2 4 6) '(20 40 60))
⇒ (22 44 66)
(seq-mapn #'concat '("moskito" "mordida") ["abelha" "picada"])
⇒ ("moskitobee" "mordendo")
```

Função: ***sequência de predicado*** **seq-filter**

Essa função retorna uma lista de todos os elementos em *sequência* para os quais o *predicado* retorna não- `nil`.

```
(seq-filtro (lambda (elt) (> elt 0)) [1 -1 3 -3 5])
⇒ (1 3 5)
```

```
(seq-filtro (lambda (elt) (> elt 0)) '(-1 -3 -5))
⇒ nada
```

Função: *sequência de predicados* seq-remove

Esta função retorna uma lista de todos os elementos em *sequência* para os quais o *predicado* retorna `nil`.

```
(seq-remove (lambda (elt) (> elt 0)) [1 -1 3 -3 5])
⇒ (-1 -3)
(seq-remove (lambda (elt) (< elt 0)) '(-1 -3 -5))
⇒ nada
```

Função: *valor inicial da sequência da função* seq-reduce

Esta função retorna o resultado de chamar a *função* com *valor inicial* e o primeiro elemento da *sequência* , depois chamar a *função* com esse resultado e o segundo elemento da *sequência* , depois com esse resultado e o terceiro elemento da *sequência* , etc. A *função* deve ser uma função de dois argumentos.

função é chamada com dois argumentos. *valor inicial* (e, em seguida, o valor acumulado) é usado como o primeiro argumento e os elementos em *sequência* são usados para o segundo argumento.

Se a *sequência* estiver vazia, isso retornará *o valor inicial* sem chamar a *função* .

```
(seq-reduce #'+ [1 2 3 4] 0)
⇒ 10
(seq-reduce #'+'(1 2 3 4) 5)
⇒ 15
(seq-reduce #'+'() 3)
⇒ 3
```

Função: seq-some *sequência de predicados*

Essa função retorna o primeiro não `nil`valor retornado aplicando *predicado* a cada elemento da *sequência* por vez.

```
(seq-some #'numberp ["abc" 1 nil])
⇒ t
(seq-some #'numberp ["abc" "def"])
⇒ nada
(seq-some #'null ["abc" 1 nil])
⇒ t
(seq-some #'1+ [2 4 6])
⇒ 3
```

Função: seq-find *sequência de predicados e padrão opcional*

Essa função retorna o primeiro elemento em *sequência* para o qual o *predicado* retorna não- `nil`. Se nenhum elemento corresponder ao *predicado* , a função retornará *o padrão* .

Observe que esta função tem uma ambiguidade se o elemento encontrado for idêntico ao *default* , pois nesse caso não se pode saber se um elemento foi encontrado ou não.

```
(seq-find #'numberp ["abc" 1 nil])
⇒ 1

(seq-find #'numberp ["abc" "def"])
⇒ nada
```

Função: ***sequência de predicados*** **seq-every-p**

Esta função retorna non- `nil`se aplicar *predicado* a cada elemento da *sequência* retorna non- `nil`.

```
(seq-every-p #'numberp [2 4 6])
⇒ t

(seq-every-p #'numberp [2 4 "6"])
⇒ nada
```

Função: ***sequência*** **seq-empty-p**

Esta função retorna non - `nil`se a *sequência* estiver vazia.

```
(seq-empty-p "não vazio")
⇒ nada

(seq-vazio-p "")
⇒ t
```

Função: ***sequência de predicados*** **seq-count**

Esta função retorna o número de elementos em *sequência* para os quais o *predicado* retorna não- `nil`.

```
(seq-count (lambda (elt) (> elt 0)) [-1 2 0 3 -2])
⇒ 2
```

Função: ***sequência de funções*** **seq-sort**

Esta função retorna uma cópia de *sequence* que é classificada de acordo com *function* , uma função de dois argumentos que retorna non- `nil`se o primeiro argumento deve ser classificado antes do segundo.

Função: ***sequência de predicados da função*** **seq-sort-by**

Esta função é semelhante a `seq-sort`, mas os elementos da *sequência* são transformados aplicando a *função* neles antes de serem ordenados. *função* é uma função de um argumento.

```
(seq-sort-by #'seq-length #'> ["a" "ab" "abc"])
⇒ ["abc" "ab" "a"]
```

Função: sequência seq-contém-p ***elt e função opcional***

Esta função retorna non- `nil`se pelo menos um elemento na *sequência* for igual a *elt* . *Se a função* de argumento opcional for non- `nil`, é uma função de dois argumentos a ser usada em vez do padrão `equal`.

```
(seq-contém-p '(símbolo1 símbolo2) 'símbolo1)
⇒ t
(seq-contém-p '(símbolo1 símbolo2) 'símbolo3)
⇒ nada
```

Função: seq-set-equal-p *sequence1 sequence2 & opcional testfn*

Esta função verifica se *sequence1* e *sequence2* contêm os mesmos elementos, independentemente da ordem. Se o argumento opcional *testfn* for não- `nil`, é uma função de dois argumentos a ser usada em vez do padrão `equal`.

```
(seq-set-igual-p '(abc) '(cba))
⇒ t
(seq-set-igual-p '(abc) '(cb))
⇒ nada
(seq-set-equal-p '("a" "b" "c") '("c" "b" "a"))
⇒ t
(seq-set-equal-p '("a" "b" "c") '("c" "b" "a") #'eq)
⇒ nada
```

Função: sequência de posição seq *elt e função opcional*

Esta função retorna o índice do primeiro elemento na *sequência* que é igual a *elt* . *Se a função* de argumento opcional for non- `nil`, é uma função de dois argumentos a ser usada em vez do padrão `equal`.

```
(posição seq '(abc) 'b)
⇒ 1
(posição seq '(abc) 'd)
⇒ nada
```

Função: sequência seq-uniq *e função opcional*

Esta função retorna uma lista dos elementos da *sequência* com duplicatas removidas. *Se a função* de argumento opcional for non- `nil`, é uma função de dois argumentos a ser usada em vez do padrão `equal`.

```
(seq-uniq '(1 2 2 1 3))
⇒ (1 2 3)
(seq-uniq '(1 2 2.0 1.0) #'=)
⇒ (1 2)
```

Função: início da sequência seq-subseq *e final opcional*

Esta função retorna um subconjunto de *sequência* do *início* ao *fim* , ambos inteiros (*end* padroniza para o último elemento). Se *start* ou *end* for negativo, conta a partir do final da *sequência* .

```
(seq-subseq '(1 2 3 4 5) 1)
⇒ (2 3 4 5)
(seq-subseq '[1 2 3 4 5] 1 3)
⇒ [2 3]
```

```
(seq-subseq '[1 2 3 4 5] -3 -1)
⇒ [3 4]
```

Função: tipo seq-concatenate *e sequências de descanso*

Esta função retorna uma sequência do tipo *type* feita da concatenação de *sequências* . *tipo* pode ser: `vector`, `listou` `string`.

```
(seq-concatenar 'lista' (1 2) '(3 4) [5 6])
⇒ (1 2 3 4 5 6)

(seq-concatenate 'string "Olá " "mundo")
⇒ "Olá mundo"
```

Função: sequência de função seq-mapcat *e tipo opcional*

Esta função retorna o resultado da aplicação `seq-concatenate` ao resultado da aplicação da *função* a cada elemento da *sequência* . O resultado é uma sequência de tipo *type* , ou uma lista se *type* for `nil`.

```
(seq-mapcat #'seq-reverse '((3 2 1) (6 5 4)))
⇒ (1 2 3 4 5 6)
```

Função: *sequência* de partição seq n

Esta função retorna uma lista dos elementos da *sequência* agrupados em subsequências de comprimento *n* . A última sequência pode conter menos elementos que *n* . *n* deve ser um número inteiro. Se *n* for um número inteiro negativo ou 0, o valor de retorno será `nil`.

```
(seq-partição '(0 1 2 3 4 5 6 7) 3)
⇒ ((0 1 2) (3 4 5) (6 7))
```

Função: sequência de interseção *seq1 sequência2 e função opcional*

Essa função retorna uma lista dos elementos que aparecem em *sequence1* e *sequence2* . *Se a função* de argumento opcional for non- `nil`, é uma função de dois argumentos a ser usada para comparar elementos em vez do padrão `equal`.

```
(seq-interseção [2 3 4 5] [1 3 5 6 7])
⇒ (3 5)
```

Função: seq-difference *sequence1 sequence2 e função opcional*

Esta função retorna uma lista dos elementos que aparecem em *sequence1* mas não em *sequence2* . *Se a função* de argumento opcional for non- `nil`, é uma função de dois argumentos a ser usada para comparar elementos em vez do padrão `equal`.

```
(seq-diferença '(2 3 4 5) [1 3 5 6 7])
⇒ (2 4)
```

Função: *sequência de função* seq-group-by

Esta função separa os elementos da *sequência* em uma lista cujas chaves são o resultado da aplicação da *função* a cada elemento da *sequência* . As chaves são comparadas usando `equal`.

```
(seq-group-by #'inteiro '(1 2.1 3 2 3.2))
⇒ ((t 1 3 2) (nil 2,1 3,2))
(seq-group-by #'car '((a 1) (b 2) (a 3) (c 4)))
⇒ ((b (b 2)) (a (a 1) (a 3)) (c (c 4)))
```

Função: *tipo de sequência* seq-into

Esta função converte a sequência de *sequência* em uma sequência do tipo *type* . *type* pode ser um dos seguintes símbolos: `vector`, `string`ou `list`.

```
(seq-into [1 2 3] 'lista)
⇒ (1 2 3)
(seq-in nil 'vetor)
⇒ []
(seq-in "olá" 'vetor)
⇒ [104 101 108 108 111]
```

Função: *sequência* seq-min

Esta função retorna o menor elemento da *sequência* . Os elementos de *sequência* devem ser números ou marcadores (ver Marcadores).

```
(seq-min [3 1 2])
⇒ 1
(seq-min "Olá")
⇒ 72
```

Função: *sequência* seq-max

Esta função retorna o maior elemento da *sequência* . Os elementos da *sequência* devem ser números ou marcadores.

```
(seq-max [1 3 2])
⇒ 3
(seq-max "Olá")
⇒ 111
```

Macro: seq-doseq *(sequência var) corpo…*

Esta macro é como `dolist`(veja dolist), exceto que a *sequência* pode ser uma lista, vetor ou string. Isso é útil principalmente para efeitos colaterais.

Macro: seq-let *var-sequence val-sequence corpo…*

Esta macro liga as variáveis definidas em *var-sequence* aos valores que são os elementos correspondentes de *val-sequence* . Isso é conhecido como *vinculação de desestruturação* . Os elementos de *var-sequence* podem incluir sequências, permitindo a desestruturação aninhada.

A *sequência var-sequence* também pode incluir o `&rest` marcador seguido por um nome de variável a ser vinculado ao restante de *val-sequence* .

```
(seq-let [primeiro segundo] [1 2 3 4]
  (listar primeiro segundo))
⇒ (1 2)

(seq-let (_ a _ b) '(1 2 3 4)
  (lista ab))
⇒ (2 4)

(seq-let [a [b [c]]] [1 [2 [3]]]
  (lista ab))
⇒ (1 2 3)

(seq-let [ab & resto outros] [1 2 3 4]
  outras)
⇒ [3 4]
```

Os `pcase`padrões fornecem um recurso alternativo para desestruturar a ligação, consulte Desestruturando com padrões pcase .

Função: *sequência* seq-random-elt

Esta função retorna um elemento de *sequência* escolhido aleatoriamente.

```
(seq-random-elt [1 2 3 4])
⇒ 3
(seq-random-elt [1 2 3 4])
⇒ 2
(seq-random-elt [1 2 3 4])
⇒ 4
(seq-random-elt [1 2 3 4])
⇒ 2
(seq-random-elt [1 2 3 4])
⇒ 1
```

Se a *sequência* estiver vazia, esta função sinaliza um erro.

Próximo:Matrizes, Acima:Vetores de matrizes de sequências [Conteúdo][Índice]