Próximo:Listar variáveis, Anterior:Elementos da lista, Acima:Listas [Conteúdo][Índice]

5.4 Construindo Células e Listas de Contras

Muitas funções criam listas, pois as listas residem no coração do Lisp. `consé` a função fundamental de construção de listas; no entanto, é interessante notar que `listé` usado mais vezes no código-fonte do Emacs do que `cons`.

Função: contras *object1 object2*

Esta função é a função mais básica para construir uma nova estrutura de lista. Ele cria uma nova célula cons, tornando *object1* o CAR e *object2* o CDR . Em seguida, ele retorna a nova célula contras. Os argumentos *object1* e *object2* podem ser quaisquer objetos Lisp, mas na maioria das vezes *object2* é uma lista.

```
(contras 1 '(2))
     ⇒ (1 2)
(contra 1 '())
     ⇒ (1)
(contras 1 2)
     ⇒ (1 . 2)
```

`consé` frequentemente usado para adicionar um único elemento à frente de uma lista. Isso é chamado *de consing do elemento na lista* . [4] Por exemplo:

```
(lista setq (lista de contras newelt))
```

Observe que não há conflito entre a variável nomeada `list` usada neste exemplo e a função nomeada `list`descrita abaixo; qualquer símbolo pode servir a ambos os propósitos.

Função: listar *&rest objetos*

Esta função cria uma lista com *objetos* como seus elementos. A lista resultante é sempre `nil`-terminada. Se nenhum *objeto* for fornecido, a lista vazia será retornada.

```
(lista 1 2 3 4 5)
     ⇒ (1 2 3 4 5)
(lista 1 2 '(3 4 5) 'foo)
     ⇒ (1 2 (3 4 5) foo)
(Lista)
     ⇒ nada
```

Função: *objeto de comprimento* da lista de criação

Esta função cria uma lista de elementos de *comprimento* , em que cada elemento é *objeto* . Compare `make-list`com `make-string`(veja Criando Strings).

```
(faça-lista 3 'porcos)
     ⇒ (porcos porcos porcos)
```

```
(fazer lista 0 'porcos)
     ⇒ nada
(setq l (make-list 3 '(ab)))
     ⇒ ((ab) (ab) (ab))
(eq (carro l) (cadr l))
     ⇒ t
```

Função: anexar *e descansar sequências*

Esta função retorna uma lista contendo todos os elementos das *sequências* . As *sequências* podem ser listas, vetores, vetores booleanos ou strings, mas a última geralmente deve ser uma lista. Todos os argumentos, exceto o último, são copiados, portanto, nenhum dos argumentos é alterado. (Veja `nconc`em Rearranjo , para uma maneira de juntar listas sem copiar.)

Mais geralmente, o argumento final para `append`pode ser qualquer objeto Lisp. O argumento final não é copiado ou convertido; torna-se o CDR da última célula contras na nova lista. Se o argumento final for uma lista, então seus elementos se tornarão elementos efetivos da lista de resultados. Se o elemento final não for uma lista, o resultado será uma lista pontilhada, pois seu CDR final não é `nil`o exigido em uma lista adequada (consulte Cons Cells).

Aqui está um exemplo de uso `append`:

```
(setq árvores '(pinheiro carvalho))
     ⇒ (carvalho de pinho)
(setq more-trees (anexar '(árvores de bétula de bordo)))
     ⇒ (maple vidoeiro pinheiro carvalho)


árvores
     ⇒ (carvalho de pinho)
mais-árvores
     ⇒ (maple vidoeiro pinheiro carvalho)
(eq árvores (cdr (cdr mais-árvores)))
     ⇒ t
```

Você pode ver como `append`funciona olhando para um diagrama de caixa. A variável `trees`é definida para a lista `(pine oak)`e, em seguida, a variável `more-trees`é definida para a lista `(maple birch pine oak)`. No entanto, a variável `trees`continua a se referir à lista original:

```
mais-árvores
| |
| --- --- --- --- -> --- --- --- ---
 --> | | |--> | | |--> | | |--> | | |--> nada
      --- --- --- --- --- --- --- ---
       | | | |
       | | | |
        --> bordo --> bétula --> pinho --> carvalho
```

Uma sequência vazia não contribui em nada para o valor retornado por `append`. Como consequência disso, um `nil`argumento final força uma cópia do argumento anterior:

```
árvores
     ⇒ (carvalho de pinho)
```

```
(setq madeira (anexar árvores nil))
     ⇒ (carvalho de pinho)
Madeira
     ⇒ (carvalho de pinho)
(eq árvores de madeira)
     ⇒ nada
```

Essa era a maneira usual de copiar uma lista, antes que a função `copy-sequence`fosse inventada. Veja Sequências Arrays Vetores .

Aqui mostramos o uso de vetores e strings como argumentos para `append`:

```
(anexar [ab] "cd" nil)
     ⇒ (ab 99 100)
```

Com a ajuda de `apply`(consulte Chamando Funções), podemos anexar todas as listas em uma lista de listas:

```
(aplica 'append' ((abc) nil (xyz) nil))
     ⇒ (abcxyz)
```

Se nenhuma *sequência* for fornecida, `nil`é retornado:

```
(acrescentar)
     ⇒ nada
```

Aqui estão alguns exemplos em que o argumento final não é uma lista:

```
(anexar '(xy) 'z)
     ⇒ (xy . z)
(anexar '(xy) [z])
     ⇒ (xy . [z])
```

O segundo exemplo mostra que quando o argumento final é uma sequência, mas não uma lista, os elementos da sequência não se tornam elementos da lista resultante. Em vez disso, a sequência se torna o CDR final , como qualquer outro argumento final não listado.

Função: árvore de cópia *e vecp opcional*

Esta função retorna uma cópia da árvore da *árvore* . Se a *árvore* for uma célula de contras, isso cria uma nova célula de contras com o mesmo CAR e CDR e copia recursivamente o CAR e o CDR da mesma maneira.

Normalmente, quando *tree* não é uma célula cons, `copy-tree`simplesmente retorna *tree* . No entanto, se *vecp* for não- `nil`, ele também copia vetores (e opera recursivamente em seus elementos).

Função: *árvore* achatada

Esta função retorna uma cópia “achatada” de *tree* , ou seja, uma lista contendo todos os `nil`nós não terminais, ou folhas, da árvore de cons células enraizadas em *tree* . As folhas na lista retornada

estão na mesma ordem que na *árvore* .

```
(aplanar-árvore '(1 (2. 3) nil (4 5 (6)) 7))
    ⇒(1 2 3 4 5 6 7)
```

Função: sequência numérica *de &opcional para separação*

Esta função retorna uma lista de números começando com *from* e incrementando por *separação* e terminando em ou imediatamente antes *de to* . *a separação* pode ser positiva ou negativa e o padrão é 1. Se *to* for `nil`ou numericamente igual a *from* , o valor será a lista de um elemento . Se *to* for menor que *from com uma separação* positiva ou maior que *from com uma separação* negativa , o valor será porque esses argumentos especificam uma sequência vazia. `(`*`from`*`)nil`

Se a *separação* for 0 e *to* não for `nil`nem numericamente igual a *from* , `number-sequence`sinaliza um erro, pois esses argumentos especificam uma sequência infinita.

Todos os argumentos são números. Os argumentos de ponto flutuante podem ser complicados, porque a aritmética de ponto flutuante é inexata. Por exemplo, dependendo da máquina, pode acontecer que `(number-sequence 0.4 0.6 0.2)`retorne a lista de um elemento `(0.4)`, enquanto `(number-sequence 0.4 0.8 0.2)`retorna uma lista com três elementos. O *enésimo* elemento da lista é calculado pela fórmula exata . Assim, se alguém quiser ter certeza de que *to* está incluído na lista, pode-se passar uma expressão desse tipo exato para *to* . Alternativamente, pode-se substituir *por* um valor um pouco maior (ou um valor um pouco mais negativo se a *separação* for negativa). `(+ `*`from`*` (* `*`n separation`*`))`

Alguns exemplos:

```
(sequência numérica 4 9)
     ⇒ (4 5 6 7 8 9)
(sequência numérica 9 4 -1)
     ⇒ (9 8 7 6 5 4)
(sequência numérica 9 4 -2)
     ⇒ (9 7 5)
(sequência numérica 8)
     ⇒ (8)
(sequência numérica 8 5)
     ⇒ nada
(sequência numérica 5 8 -1)
     ⇒ nada
(sequência numérica 1,5 6 2)
     ⇒ (1,5 3,5 5,5)
```

Notas de rodapé

(4)

Não existe uma maneira estritamente equivalente de adicionar um elemento ao final de uma lista. Você pode usar , que cria uma lista totalmente nova copiando *listname* e adicionando *newelt* ao final. Ou você pode usar , que modifica o *nome* da lista seguindo todos os CDRs e substituindo a terminação . Compare isso com a adição de um elemento ao início de uma lista com , que não copia nem modifica a lista. `(append `*`listname`*` (list `*`newelt`*`))(nconc `*`listname`*` (list `*`newelt`*`))nilcons`

Próximo:Listar variáveis, Anterior:Elementos da lista, Acima:Listas [Conteúdo][Índice]