Próximo:Listas de Associação, Anterior:Modificando listas, Acima:Listas [Conteúdo][Índice]

5.7 Usando Listas como Conjuntos

Uma lista pode representar um conjunto matemático não ordenado - simplesmente considere um valor como um elemento de um conjunto se ele aparecer na lista e ignore a ordem da lista. Para formar a união de dois conjuntos, use `append`(desde que não se importe de ter elementos duplicados). Você pode remover `equal`duplicatas usando `delete-dups`. Outras funções úteis para conjuntos incluem `memq`e `delq`, e suas `equal`versões, `member`e `delete`.

Nota do Common Lisp: Common Lisp possui funções `union`(que evitam elementos duplicados) e `intersection`para operações de conjunto. No Emacs Lisp, as variantes dessas facilidades são fornecidas pelo `cl-lib`biblioteca. Consulte Listas como Conjuntos em Extensões Common Lisp .

Função: lista de *objetos* memq

Esta função testa para ver se o *objeto* é um membro de *list* . Se for, `memq`retorna uma lista começando com a primeira ocorrência de *object* . Caso contrário, ele retorna `nil`. A carta '`q`' in `memq`diz que ele usa `eq`para comparar o *objeto* com os elementos da lista. Por exemplo:

```
(memq 'b' (abcba))
     ⇒ (bcba)
(memq '(2) '((1) (2))); Os dois (2)s não precisam ser eq.
     ⇒ Não especificado; pode ser nilou ((2)).
```

Função: lista de *objetos* delq

Essa função remove destrutivamente todos os elementos `eq`para *object* de *list* e retorna a lista resultante. A carta '`q`' in `delq`diz que ele usa `eq`para comparar o *objeto* com os elementos da lista, como `memq`e `remq`.

Normalmente, quando você invoca `delq`, você deve usar o valor de retorno atribuindo-o à variável que continha a lista original. A razão para isso é explicada abaixo.

A `delq`função exclui elementos da frente da lista simplesmente avançando na lista e retornando uma sublista que começa após esses elementos. Por exemplo:

```
(delq 'a '(abc)) ≡ (cdr '(abc))
```

When an element to be deleted appears in the middle of the list, removing it involves changing the CDRs (see Setcdr).

```
(setq sample-list (list 'a 'b 'c '(4)))
     ⇒ (a b c (4))
(delq 'a sample-list)
     ⇒ (b c (4))
sample-list
     ⇒ (a b c (4))
(delq 'c sample-list)
     ⇒ (a b (4))
```

```
sample-list
     ⇒ (a b (4))
```

Note that `(delq 'c sample-list)` modifies `sample-list` to splice out the third element, but `(delq 'a sample-list)` does not splice anything—it just returns a shorter list. Don't assume that a variable which formerly held the argument *list* now has fewer elements, or that it still holds the original list! Instead, save the result of `delq` and use that. Most often we store the result back into the variable that held the original list:

```
(setq flowers (delq 'rose flowers))
```

In the following example, the `(list 4)` that `delq` attempts to match and the `(4)` in the `sample-list` are `equal` but not `eq`:

```
(delq (list 4) sample-list)
     ⇒ (a c (4))
```

If you want to delete elements that are `equal` to a given value, use `delete` (see below).

Function: remq *object list*

This function returns a copy of *list*, with all elements removed which are `eq` to *object*. The letter '`q`' in `remq` says that it uses `eq` to compare *object* against the elements of `list`.

```
(setq sample-list (list 'a 'b 'c 'a 'b 'c))
     ⇒ (a b c a b c)
(remq 'a sample-list)
     ⇒ (b c b c)
sample-list
     ⇒ (a b c a b c)
```

Function: memql *object list*

The function `memql` tests to see whether *object* is a member of *list*, comparing members with *object* using `eql`, so floating-point elements are compared by value. If *object* is a member, `memql` returns a list starting with its first occurrence in *list*. Otherwise, it returns `nil`.

Compare this with `memq`:

```
(memql 1.2 '(1.1 1.2 1.3))  ; 1.2 and 1.2 are eql.
     ⇒ (1.2 1.3)
(memq 1.2 '(1.1 1.2 1.3))  ; The two 1.2s need not be eq.
     ⇒ Unspecified; might be nil or (1.2 1.3).
```

The following three functions are like `memq`, `delq` and `remq`, but use `equal` rather than `eq` to compare elements. See Equality Predicates.

Function: member *object list*

The function `member` tests to see whether *object* is a member of *list*, comparing members with *object* using `equal`. If *object* is a member, `member` returns a list starting with its first occurrence in

list. Otherwise, it returns `nil`.

Compare this with `memq`:

```
(member '(2) '((1) (2)))  ; (2) and (2) are equal.
     ⇒ ((2))
(memq '(2) '((1) (2)))    ; The two (2)s need not be eq.
     ⇒ Unspecified; might be nil or (2).
;; Two strings with the same contents are equal.
(member "foo" '("foo" "bar"))
     ⇒ ("foo" "bar")
```

Function: delete *object sequence*

This function removes all elements `equal` to *object* from *sequence*, and returns the resulting sequence.

If *sequence* is a list, `delete` is to `delq` as `member` is to `memq`: it uses `equal` to compare elements with *object*, like `member`; when it finds an element that matches, it cuts the element out just as `delq` would. As with `delq`, you should typically use the return value by assigning it to the variable which held the original list.

If `sequence` is a vector or string, `delete` returns a copy of `sequence` with all elements `equal` to `object` removed.

For example:

```
(setq l (list '(2) '(1) '(2)))
(delete '(2) l)
     ⇒ ((1))
l
     ⇒ ((2) (1))
;; If you want to change l reliably,
;; write (setq l (delete '(2) l)).
(setq l (list '(2) '(1) '(2)))
(delete '(1) l)
     ⇒ ((2) (2))
l
     ⇒ ((2) (2))
;; In this case, it makes no difference whether you set l,
;; but you should do so for the sake of the other case.
(delete '(2) [(2) (1) (2)])
     ⇒ [(1)]
```

Function: remove *object sequence*

This function is the non-destructive counterpart of `delete`. It returns a copy of `sequence`, a list, vector, or string, with elements `equal` to `object` removed. For example:

```
(remove '(2) '((2) (1) (2)))
     ⇒ ((1))
(remove '(2) [(2) (1) (2)])
     ⇒ [(1)]
```

Common Lisp note: The functions `member`, `delete` and `remove` in GNU Emacs Lisp are derived from Maclisp, not Common Lisp. The Common Lisp versions do not use `equal` to compare elements.

Function: member-ignore-case *object list*

This function is like `member`, except that *object* should be a string and that it ignores differences in letter-case and text representation: upper-case and lower-case letters are treated as equal, and unibyte strings are converted to multibyte prior to comparison.

Function: delete-dups *list*

This function destructively removes all `equal` duplicates from *list*, stores the result in *list* and returns it. Of several `equal` occurrences of an element in *list*, `delete-dups` keeps the first one.

See also the function `add-to-list`, in List Variables, for a way to add an element to a list stored in a variable and used as a set.

Next: Association Lists, Previous: Modificando listas, Acima:Listas [Conteúdo][Índice]