第11章 集 合

和多数编程语言一样，Common Lisp也提供了能将多个值收集到单一对象的标准数据类型。每一种语言在处理集合问题上都稍有不同，但基本的集合类型通常都可归结为一个整数索引的数组类型，以及一个可将或多或少的任意关键字映射到值上的表类型。前者分别称为*数组*（array）、*列表*（list）或*元组*（tuple），后者命名为*哈希表*（hash table）、*关联数组*（associative array）、*映射表*（map）和*字典*（dictionary）。

当然，Lisp以其列表数据结构闻名于世，而多数遵循了语言用法的进化重演（ontogeny-recapitulates-phylogeny）原则的Lisp教材也都从讨论基于列表的Lisp集合开始。尽管如此，这一观点通常导致读者错误地推论出列表是Lisp的唯一集合类型。更糟糕的是，因为Lisp的列表是如此灵活的数据结构，它可用于许多其他语言使用数组和哈希表的场合。但是将注意力过于集中在列表上是错误的。尽管它是一种将Lisp代码作为Lisp数据来表达的关键数据结构，但在许多场合其他数据结构更合适。

为了避免让列表过于出风头，在本章里我将集中在Common Lisp的其他集合类型上：向量和哈希表。[1]尽管如此，向量和列表共享了许多特征，因此Common Lisp将它们都视为更一般抽象的子类型（即序列）。因此，你可以将本章我所讨论的许多函数同时用在向量和列表上。

11.1 向量

向量是Common Lisp基本的整数索引集合，它们分为两大类。定长向量与Java等语言里的数组非常相似：一块数据头以及一段保存向量元素的连续内存区域。[2]另一方面，变长向量更像是Perl或Ruby中的数组、Python中的列表以及Java中的ArrayList类：它们抽象了实际存储，允许向量随着元素的增加和移除而增大和减小。

你可以用函数VECTOR来生成含有特定值的定长向量，该函数接受任意数量的参数并返回一个新分配的含有那些参数的定长向量。

---

① 一旦你熟悉了Common Lisp提供的所有数据类型，会发现列表可以作为原型数据结构来使用，并且以后可以替换成其他更高效的东西，只要你清楚了使用数据的确切方式。

② 向量被称为向量，而不是像其他语言里那样被称为数组，是因为Common Lisp支持真正的多维数组。将它称为一维数组应该更加确切，但过于冗繁。

```
(vector)      → #()
(vector 1)    → #(1)
(vector 1 2) → #(1 2)
```

语法#(...)是Lisp打印器和读取器使用的向量的字面表示形式，该语法可使你用**PRINT**打印并用**READ**读取，以此来保存并恢复向量。可以使用#(...)语法在代码中添加字面向量，但修改字面对象的后果并不明确，因此应当总是使用**VECTOR**或更为通用的函数**MAKE-ARRAY**来创建打算修改的向量。

MAKE-ARRAY比**VECTOR**更加通用，因为它可以用来创建任何维度的数组以及定长和变长向量。**MAKE-ARRAY**的一个必要参数是一个含有数组维数的列表。由于向量是一维数组，所以该列表将含有一个数字，也就是向量的大小。出于方便的考量，**MAKE-ARRAY**也会用一个简单的数字来代替只含有一项的列表。如果没有其他参数，**MAKE-ARRAY**就将创建一个带有未初始化元素的向量，它们必须在被访问之前设置其值。[①]为了创建所有元素都设置到一个特定值上的向量，你可以传递一个:initial-element参数。因此，为了生成一个元素初始化到**NIL**的五元素向量，你可以写成下面这样：

```
(make-array 5 :initial-element nil) → #(NIL NIL NIL NIL NIL)
```

MAKE-ARRAY也是用来创建变长向量的函数。变长向量是比定长向量稍微复杂的向量。除了跟踪其用来保存元素的内存和可访问的槽位数量，变长向量还要跟踪实际存储在向量中的元素数量。这个数字存放在向量的*填充指针*里，这样称呼是因为它是当为向量添加一个元素时下一个被填充位置的索引。

为了创建带有填充指针的向量，你可以向**MAKE-ARRAY**传递一个:fill-pointer实参。例如，下面的**MAKE-ARRAY**调用生成了一个带有五元素空间的向量，它看起来是空的，因为填充指针是零：

```
(make-array 5 :fill-pointer 0) → #()
```

为了向可变向量的尾部添加一个元素，你可以使用函数**VECTOR-PUSH**。它在填充指针的当前值上添加一个元素并将填充指针递增一次，并返回新元素被添加位置的索引。函数**VECTOR-POP**返回最近推入的项，并在该过程中递减填充指针。

```
(defparameter *x* (make-array 5 :fill-pointer 0))

(vector-push 'a *x*) → 0
*x*                  → #(A)
(vector-push 'b *x*) → 1
*x*                  → #(A B)
(vector-push 'c *x*) → 2
*x*                  → #(A B C)
(vector-pop *x*)     → C
*x*                  → #(A B)
(vector-pop *x*)     → B
*x*                  → #(A)
```

① 数组元素在其被访问前“必须”被赋值，如果不这样做，其行为将是未定义的。Lisp不一定会报错。

```
(vector-pop *x*)      → A
*x*                   → #()
```

尽管如此，甚至一个带有填充指针的向量也不是完全变长的。向量*x*只能保存最多五个元素。为了创建一个可任意变长的向量，你需要向**MAKE-ARRAY**传递另外一个关键字参数:adjustable。

```
(make-array 5 :fill-pointer 0 :adjustable t) → #()
```

这个调用生成了一个可调整的向量，其底层内存可以按需调整大小。为了向一个可调整向量添加元素，你可以使用**VECTOR-PUSH-EXTEND**，它就像**VECTOR-PUSH**那样工作，只是在你试图向一个已满的向量（其填充指针等于底层存储的大小）中推入元素时，它能自动扩展该数组。[①]

11.2 向量的子类型

目前为止，你处理的所有向量都是可以保存任意类型对象的通用向量。你也可以创建特化的向量使其仅限于保存特定类型的元素。使用特化向量的理由之一是，它们可以更加紧凑地存储，并且可以比通用向量提供对其元素更快速的访问。不过目前我们将集中介绍几类特化向量，它们本身就是重要的数据类型。

其中一类你已经见过了，就是字符串，它是特定用来保存字符的向量。字符串特别重要，以至于它们有自己的读写语法（双引号）和一组特定于字符串的函数，前一章已讨论过。但因为它们也是向量，所有接下来几节里所讨论的接受向量实参的函数也可以用在字符串上。这些函数将为字符串函数库带来新的功能，例如用一个子串来搜索字符串，查找一个字符在字符串中出现的次数，等等。

诸如“foo”这样的字面字符串，和那些用#()语法写成的字面向量一样，其大小都是固定的，并且根本不能修改它们。但你可以用**MAKE-ARRAY**通过添加另一个关键字参数:element-type来创建变长字符串。该参数接受一个类型描述符。我将不会介绍你可以在这里使用的所有可能的类型描述符，目前只需知道，你可以通过传递符号**CHARACTER**作为:element-type来创建字符串。注意，你需要引用该符号以避免它被视为一个变量名。例如，创建一个初始为空但却变长的字符串，如下所示：

```
(make-array 5 :fill-pointer 0 :adjustable t :element-type 'character) → ""
```

位向量是元素全部由0或1所组成的向量，它也得到一些特殊对待。它们有一个特别的读/写语法，看起来像#*00001111，另外还有一个相对巨大的函数库。这些函数可用于按位操作，例如将两个位数组“与”在一起（不会介绍）。用来创建一个位向量传递给:element-type的类型描述符是符号BIT。

① 尽管经常一起使用，但:file-pointer和:adjustable是无关的——你可以生成一个不带有填充指针的可调整数组。不过，你只能在带有填充指针的向量上使用**VECTOR-PUSH**和**VECTOR-POP**，并只能在带有填充指针且可调整的向量上使用**VECTOR-PUSH-EXTEND**。你还可以使用函数**ADJUST-ARRAY**以超出扩展向量长度的多种方式来修改可调整数组。

11.3　作为序列的向量

正如早先所提到的，向量和列表是抽象类型序列的两种具体子类型。接下来几节里讨论的所有函数都是序列函数：除了可以应用于向量（无论是通用还是特化的）之外，它们还可应用于列表。

两个最基本的序列函数是**LENGTH**，其返回一个序列的长度；**ELT**，其允许通过一个整数索引来访问个别元素。**LENGTH**接受序列作为其唯一的参数并返回它含有的元素数量。对于带有填充指针的向量，这些是填充指针的值。**ELT**是元素（element）的简称，它接受序列和从0到序列长度（左闭右开区间）的整数索引，并返回对应的元素。**ELT**将在索引超出边界时报错。和**LENGTH**一样，**ELT**也将把一个带有填充指针的向量视为其具有该填充指针所指定的长度。

```
(defparameter *x* (vector 1 2 3))

(length *x*) → 3
(elt *x* 0)  → 1
(elt *x* 1)  → 2
(elt *x* 2)  → 3
(elt *x* 3)  → error
```

ELT也是一个支持**SETF**的位置，因此可以像这样来设置一个特定元素的值：

```
(setf (elt *x* 0) 10)

*x* → #(10 2 3)
```

11.4　序列迭代函数

尽管理论上所有的序列操作都可归结于**LENGTH**、**ELT**和**ELT**的**SETF**操作的某种组合，但Common Lisp还是提供了一个庞大的序列函数库。

一组序列函数允许你无需编写显式循环就可以表达一些特定的序列操作，比如说查找或过滤指定元素等。表11-1总结如下。

表11-1　基本序列函数

名　称	所需参数	返　回
COUNT	项和序列	序列中出现某项的次数
FIND	项和序列	项或NIL
POSITION	项和序列	序列中的索引NIL
REMOVE	项和序列	项的实例被移除后的序列
SUBSTITUTE	新项、项和序列	项的实项被新项替换后的序列

下面是一些关于如何使用这些函数的简单例子：

```
(count 1 #(1 2 1 2 3 1 2 3 4))          → 3
(remove 1 #(1 2 1 2 3 1 2 3 4))         → #(2 2 3 2 3 4)
(remove 1 '(1 2 1 2 3 1 2 3 4))         → (2 2 3 2 3 4)
(remove #\a "foobarbaz")                → "foobrbz"
```

```
(substitute 10 1 #(1 2 1 2 3 1 2 3 4)) → #(10 2 10 2 3 10 2 3 4)
(substitute 10 1 '(1 2 1 2 3 1 2 3 4)) → (10 2 10 2 3 10 2 3 4)
(substitute #\x #\b "foobarbaz")       → "fooxarxaz"
(find 1 #(1 2 1 2 3 1 2 3 4))          → 1
(find 10 #(1 2 1 2 3 1 2 3 4))         → NIL
(position 1 #(1 2 1 2 3 1 2 3 4))      → 0
```

注意，**REMOVE**和**SUBSTITUTE**总是返回与其序列实参相同类型的序列。

可以使用关键字参数以多种方式修改这五个函数的行为。例如，在默认情况下，这些函数会查看序列中与其项参数相同的对象。你可以用两种方式改变这一行为。首先，你可以使用:test关键字来传递一个接受两个参数并返回一个布尔值的函数。如果有了这一函数，它将使用该函数代替默认的对象等价性测试**EQL**来比较序列中的每个元素。[①]其次，使用:key关键字可以传递单参数函数，其被调用在序列的每个元素上以抽取出一个关键值，该值随后会和替代元素自身的项进行比对。但请注意，诸如**FIND**这类返回序列元素的函数将仍然返回实际的元素而不只是被抽取出的关键值。

```
(count "foo" #("foo" "bar" "baz") :test #'string=)    → 1
(find 'c #((a 10) (b 20) (c 30) (d 40)) :key #'first) → (C 30)
```

为了将这些函数的效果限制在序列实参的特定子序列上，你可以用:start和:end参数提供边界指示。为:end传递**NIL**或是省略它与指定该序列的长度具有相同的效果。[②]

如果你使用非**NIL**的:from-end参数，那些序列的元素将以相反的顺序被检查。:from-end单独使用只能影响**FIND**和**POSITION**的结果。例如：

```
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first)              → (A 10)
(find 'a #((a 10) (b 20) (a 30) (b 40)) :key #'first :from-end t) → (A 30)
```

而:from-end参数和另一个关键字参数:count用于指定有多少个元素被移除或替换，这两个参数一起使用时可能影响**REMOVE**和**SUBSTITUTE**的行为。如果指定了一个低于匹配元素数量的:count，那么从哪一端开始显然至关重要：

```
(remove #\a "foobarbaz" :count 1)              → "foobrbaz"
(remove #\a "foobarbaz" :count 1 :from-end t) →"foobarbz"
```

尽管:from-end无法改变**COUNT**函数的结果，但它确实可以影响传递给任何:test和:key函数的元素的顺序，这些函数可能带有副作用。例如：

```
CL-USER> (defparameter *v* #((a 10) (b 20) (a 30) (b 40)))
*V*
```

① 另一个形参:test-not指定了一个两参数谓词，它可以像:test参数那样使用除了带有逻辑上相反的布尔结果。这个参数已经过时，而目前推荐使用**COMPLEMENT**函数。**COMPLEMENT**接受一个函数参数，然后返回一个带有相同数量参数的函数，其返回与原先函数逻辑上相反的结果。因此你可以而且也应该写成这样：

```
(count x sequence :test (complement #'some-test))
```

而不是这样：

```
(count x sequence :test-not #'some-test)
```

② 注意，尽管如此，:start和:end在**REMOVE**和**SUBSTITUTE**的效果仅限于它们所考虑移除或替换的元素，在:start之前和:end之后的元素将原封不动地传递。

```
CL-USER> (defun verbose-first (x) (format t "Looking at ~s~%" x) (first x))
VERBOSE-FIRST
CL-USER> (count 'a *v* :key #'verbose-first)
Looking at (A 10)
Looking at (B 20)
Looking at (A 30)
Looking at (B 40)
2
CL-USER> (count 'a *v* :key #'verbose-first :from-end t)
Looking at (B 40)
Looking at (A 30)
Looking at (B 20)
Looking at (A 10)
2
```

表11-2总结了这些参数。

表11-2　标准序列函数关键字参数

参　数	含　义	默认值
:test	两参数函数用来比较元素（或由:key函数解出的值）和项	EQL
:key	单参数函数用来从实际的序列元素中解出用于比较的关键字值 NIL表示原样采用序列元素	NIL
:start	子序列的起始索引（含）	0
:end	子序列的终止索引（不含）。NIL表示到序列的结尾	NIL
:from-end	如果为真，序列将以相反的顺序遍历，从尾到头	NIL
:count	数字代表需要移除或替换的元素个数，NIL代表全部。（仅用于REMOVE和SUBSTITETE）	NIL

11.5　高阶函数变体

对于每个刚刚讨论过的函数，Common Lisp都提供了两种高阶函数变体，它们接受一个将在每个序列元素上调用的函数，以此来代替项参数。一组变体被命名为与基本函数相同的名字并带有一个追加的-IF。这些函数用于计数、查找、移除以及替换序列中那些函数参数返回真的元素。另一类变体以-IF-NOT后缀命名并计数、查找、移除以及替换函数参数不返回真的元素。

```
(count-if #'evenp #(1 2 3 4 5))          → 2

(count-if-not #'evenp #(1 2 3 4 5))      → 3

(position-if #'digit-char-p "abcd0001") → 4

(remove-if-not #'(lambda (x) (char= (elt x 0) #\f))
  #("foo" "bar" "baz" "foom")) → #("foo" "foom")
```

根据语言标准，这些-IF-NOT变体已经过时了。但这种过时通常被认为是由于标准本身欠考虑。不过，如果再次修订标准，更有可能被去掉的是-IF而非-IF-NOT系列。比如说，有个叫

REMOVE-IF-NOT的变体就比**REMOVE-IF**更经常使用。尽管它有一个听起来具有否定意义的名字，但**REMOVE-IF-NOT**实际上是一个具有肯定意义的变体——它返回满足谓词的那些元素。[1]

除了:test，这些-IF和-IF-NOT变体都接受和它们的原始版本相同的关键字参数，:test不再被需要是因为主参数已经是一个函数了。[2]通过使用:key参数，由:key函数所抽取出的值将代替实际元素传递给该函数。

```
(count-if #'evenp #((1 a) (2 b) (3 c) (4 d) (5 e)) :key #'first)     → 2

(count-if-not #'evenp #((1 a) (2 b) (3 c) (4 d) (5 e)) :key #'first) → 3

(remove-if-not #'alpha-char-p
  #("foo" "bar" "1baz") :key #'(lambda (x) (elt x 0))) → #("foo" "bar")
```

REMOVE函数家族还支持第四个变体**REMOVE-DUPLICATES**，它接受序列作为仅需的必要参数，并将其中每个重复的元素移除到只剩下一个实例。除:count外，它与**REMOVE**有相同的关键字参数，因为它总是移除所有重复的元素。

```
(remove-duplicates #(1 2 1 2 3 1 2 3 4)) → #(1 2 3 4)
```

11.6 整个序列上的操作

有一些函数每次在整个序列（或多个序列）上进行操作。这些函数比目前已描述的其他函数简单一些。例如，**COPY-SEQ**和**REVERSE**都接受单一的序列参数并返回一个相同类型的新序列。**COPY-SEQ**返回的序列包含与其参数相同的元素，而**REVERSE**返回的序列则含有顺序相反的相同元素。注意，这两个函数都不会复制元素本身，只有返回的序列是一个新对象。

函数**CONCATENATE**创建一个将任意数量序列连接在一起的新序列。不过，跟**REVERSE**和**COPY-SEQ**简单地返回与其单一参数相同类型序列有所不同的是，**CONCATENATE**必须被显式指定产生何种类型的序列，因为其参数可能是不同类型的。它的第一个参数是类型描述符，就像是**MAKE-ARRAY**的:element-type参数。这里最常用到的类型描述符是符号**VECTOR**、**LIST**和**STRING**。[3]例如：

```
(concatenate 'vector #(1 2 3) '(4 5 6))     → #(1 2 3 4 5 6)
(concatenate 'list #(1 2 3) '(4 5 6))       → (1 2 3 4 5 6)
(concatenate 'string "abc" '(#\d #\e #\f)) → "abcdef"
```

① 同样的功能由Perl中的grep和Python中的filter所实现。

② 作为:test参数传递的谓词与作为函数参数传递给-IF和-IF-NOT函数的谓词之间的区别在于：:test谓词是用来将序列元素与特定项相比较的两参数谓词，而-IF和-IF-NOT谓词是简单测试序列元素的单参数函数。如果原始变体不存在，你可以通过将一个特定的项嵌入到测试函数中，从而用-IF版本来实现它们。

```
(count char string ≡)
  (count-if #'(lambda (c) (eql char c)) string)
(count char string :test #'CHAR-EQUAL) ≡
  (count-if #'(lambda (c) (char-equal char c)) string)
```

③ 如果让**CONCATENATE**返回一个特化的向量，例如一个字符串，那么参数序列的所有元素都必须是该向量元素类型的实例。

11.7　排序与合并

函数**SORT**和**STABLE-SORT**提供了两种序列排序方式。它们都接受一个序列和一个由两个实参组成的谓词，返回该序列排序后的版本。

```
(sort (vector "foo" "bar" "baz") #'string<) → #("bar" "baz" "foo")
```

它们的区别在于，**STABLE-SORT**可以保证不会重排任何被该谓词视为等价的元素，而**SORT**只保证结果是已排序的并可能重排等价元素。

这两个函数都是所谓的*破坏性*（destructive）函数。通常出于效率的原因，破坏性函数都会或多或少地修改它们的参数。这有两层含义：第一，你应该总是对这些函数的返回值做一些事情（比如给它赋值一个变量或将它传递给另一个函数）；第二，除非你不再需要传给破坏性函数的那个对象，否则应该传递一个副本。下一章里将会讨论更多有关破坏性函数的内容。

在排序以后，你通常不会再关心那个序列的未排序版本，因此在排序的过程中，允许**SORT**和**STABLE-SORT**破坏序列是合理的。但这意味着需要记得要这样来写：[①]

```
(setf my-sequence (sort my-sequence #'string<))
```

而不只是这样：

```
(sort my-sequence #'string<)
```

这两个函数也接受关键字参数`:key`，它和其他序列函数的`:key`参数一样，应当是一个将被用来从序列元素中抽取出传给排序谓词的值的函数。被抽取出的关键字仅用于确定元素顺序，返回的序列将含有参数序列的实际元素。

函数**MERGE**接受两个序列和一个谓词，并返回按照该谓词合并这两个序列所产生的序列。它和两个排序函数之间的关系在于，如果每个序列已经被同样的谓词排序过了，那么由**MERGE**返回的序列也将是有序的。和排序函数一样，**MERGE**也接受一个`:key`参数。和**CONCATENATE**一样，出于同样原因，**MERGE**的第一个参数必须是用来指定所生成序列类型的类型描述符。

```
(merge 'vector #(1 3 5) #(2 4 6) #'<) → #(1 2 3 4 5 6)
(merge 'list #(1 3 5) #(2 4 6) #'<)   → (1 2 3 4 5 6)
```

11.8　子序列操作

另一类函数允许你对已有序列的子序列进行操作，其中最基本的是**SUBSEQ**，它解出序列中从一个特定索引开始并延续到一个特定终止索引或结尾处的子序列。例如：

```
(subseq "foobarbaz" 3)   → "barbaz"
(subseq "foobarbaz" 3 6) → "bar"
```

SUBSEQ也支持**SETF**，但不会扩大或缩小一个序列。如果新的值和将被替换的子序列具有不

① 当传递给排序函数的序列是一个向量时，其破坏性实际上可以确保进行元素的就地交换，因此你可以无需保存返回值而得到正确的效果。尽管如此，总是对返回值做一些事情是好的编程风格，因为排序函数可以以更灵活的方式来修改列表。

同的长度，那么两者中较短的那一个将决定有多少个字符被实际改变。

```
(defparameter *x* (copy-seq "foobarbaz"))

(setf (subseq *x* 3 6) "xxx")  ; 子序列和新值具有相同的长度。
*x* → "fooxxxbaz"

(setf (subseq *x* 3 6) "abcd") ; 新值太长，其他字符被忽略。
*x* → "fooabcbaz"

(setf (subseq *x* 3 6) "xx")   ; 新值太短，只有两个字符被替换。
*x* → "fooxxcbaz"
```

你可以使用**FILL**函数来将一个序列的多个元素设置到单个值上。所需的参数是一个序列以及所填充的值。在默认情况下，该序列的每个元素都设置到该值上。:start和:end关键字参数可以将效果限制在一个给定的子序列上。

如果你需要在一个序列中查找一个子序列，**SEARCH**函数可以像**POSITION**那样工作，不过第一个参数是一个序列而不是一个单独的项。

```
(position #\b "foobarbaz") → 3
(search "bar" "foobarbaz") → 3
```

另一方面，为了找出两个带有相同前缀的序列首次分岔的位置，可以使用**MISMATCH**函数。它接受两个序列并返回第一对不相匹配的元素的索引。

```
(mismatch "foobarbaz" "foom") → 3
```

如果字符串匹配，它将返回**NIL**。**MISMATCH**也接受许多标准关键字参数：:key参数可以指定一个函数用来抽取出被比较的值；:test参数用于指定比较函数；而:start1、:end1、:start2和:end2参数则用来指定两个序列中的子序列。另外，一个设置为**T**的:from-end参数可以指定以相反的顺序搜索序列，从而导致**MISMATCH**返回索引值，这个索引值表示两个序列的相同后缀在第一个序列中的开始位置。

```
(mismatch "foobar" "bar" :from-end t) → 3
```

11.9 序列谓词

另外四个常见的函数是**EVERY**、**SOME**、**NOTANY**和**NOTEVERY**，它们在序列上迭代并测试一个布尔谓词。所有这些函数的第一参数是谓词，其余的参数都是序列。这个谓词应当接受与所传递序列相同数量的参数。序列的元素传递给该谓词，每个序列中各取出一个元素，直到某个序列用完所有的元素或满足了整体终止测试条件。**EVERY**在谓词失败时返回假；如果谓词总被满足，它返回真。**SOME**返回由谓词所返回的第一个非**NIL**值，或者在谓词永远得不到满足时返回假。**NOTANY**将在谓词满足时返回假，或者在从未满足时返回真。而**NOTEVERY**在谓词失败时返回真，或是在谓词总是满足时返回假。下面是一些仅在一个序列上测试的例子：

```
(every #'evenp #(1 2 3 4 5))     → NIL
(some #'evenp #(1 2 3 4 5))      → T
```

```
(notany #'evenp #(1 2 3 4 5))    → NIL
(notevery #'evenp #(1 2 3 4 5)) → T
```

下面的调用比较成对的两个序列中的元素：

```
(every #'> #(1 2 3 4) #(5 4 3 2))     → NIL
(some #'> #(1 2 3 4) #(5 4 3 2))      → T
(notany #'> #(1 2 3 4) #(5 4 3 2))    → NIL
(notevery #'> #(1 2 3 4) #(5 4 3 2)) → T
```

11.10　序列映射函数

最后的序列函数是通用映射函数。**MAP**和序列谓词函数一样，接受一个*n*-参数函数和*n*个序列。**MAP**不返回布尔值，而是返回一个新序列，它由那些将函数应用在序列的相继元素上所得到的结果组成。与**CONCATENATE**和**MERGE**相似，**MAP**需要被告知其所创建序列的类型。

```
(map 'vector #'* #(1 2 3 4 5) #(10 9 8 7 6)) → #(10 18 24 28 30)
```

MAP-INTO和**MAP**相似，但它并不产生给定类型的新序列，而是将结果放置在一个作为第一个参数传递的序列中。这个序列可以是为函数提供值的序列中的一个。例如，为了将几个向量a、b和c相加到其中一个向量里，可以写成这样：

```
(map-into a #'+ a b c)
```

如果这些序列的长度不同，那么**MAP-INTO**将只影响与最短序列元素数量相当的那些元素，其中也包括那个将被映射到的序列。不过，如果序列被映射到一个带有填充指针的向量里，受影响元素的数量将不限于填充指针而是该向量的实际大小。在对**MAP-INTO**的调用之后，填充指针将被设置成被映射元素的数量。尽管如此，**MAP-INTO**将不会扩展一个可调整大小的向量。

最后一个序列函数是**REDUCE**，它可以做另一种类型的映射：映射在单个序列上，先将一个两参数函数应用到序列的最初两个元素上，再将函数返回值和序列后续元素继续用于该函数。这样，下面的表达式将对从1到10的整数求和：

```
(reduce #'+ #(1 2 3 4 5 6 7 8 9 10)) → 55
```

REDUCE函数非常有用，无论何时，当需要将一个序列提炼成一个单独的值时，你都有机会用**REDUCE**来写它，而这通常是一种相当简洁的表达意图的方法。例如，为了找出一个数字序列中的最大值，可以写成`(reduce #'max numbers)`。**REDUCE**也接受完整的关键字参数（`:key`、`:from-end`、`:start`和`:end`）以及一个**REDUCE**专用的`:intial-value`。后者可以指定一个值，在逻辑上被放置在序列的第一个元素之前（或者，如果你同时指定了一个为真的`:from-end`参数，那么该值被放置在序列的最后一个元素之后）。

11.11　哈希表

Common Lisp提供的另一个通用集合类型是哈希表。与提供整数索引的数据结构的向量有所不同的是，哈希表允许你使用任意对象作为索引或是键（key）。当向哈希表添加值时，可以把它

保存在一个特定的键下。以后就可以使用相同的键来获取该值，或者可以将同一个键关联到一个新值上——每个键映射到单一值上。

不带参数的**MAKE-HASH-TABLE**将创建一个哈希表，其认定两个键等价，当且仅当它们在**EQL**的意义上是相同的对象。这是一个好的默认值，除非你想要使用字符串作为健，因为两个带有相同内容的字符串不一定是**EQL**等价的。在这种情况下，你需要一个所谓的**EQUAL**哈希表，它可以通过将符号**EQUAL**作为:test关键字参数传递给**MAKE-HASH-TABLE**来获得。:test参数的另外两个可能的值是符号**EQ**和**EQUALP**。这些都是第4章里讨论过的标准对象比较函数的名字。不过，和传递给序列函数的:test参数不同的是，**MAKE-HASH-TABLE**的:test不能用来指定一个任意函数——只能是值**EQ**、**EQL**、**EQUAL**和**EQUALP**。这是因为哈希表实际上需要两个函数：一个等价性函数；一个以一种和等价函数最终比较两个键时相兼容的方式，用来从键中计算出一个数值的哈希码的哈希函数。不过，尽管语言标准仅提供了使用标准等价函数的哈希表，但多数实现都提供了一些自定义哈希表的方法。

函数**GETHASH**提供了对哈希表元素的访问。它接受两个参数，即键和哈希表，并返回保存在哈希表中相应键下的值（如果有的话）或是**NIL**。[①]例如：

```
(defparameter *h* (make-hash-table))

(gethash 'foo *h*) → NIL

(setf (gethash 'foo *h*) 'quux)

(gethash 'foo *h*) → QUUX
```

由于当键在表中不存在时**GETHASH**返回**NIL**，所以无法从返回值中看出，究竟是键在哈希表中不存在还是键在表中存在却带有值**NIL**。**GETHASH**用一个我尚未讨论到的特性解决了这一问题，即通过多重返回值。**GETHASH**实际上返回两个值：主值是保存在给定键下的值或**NIL**；从值是一个布尔值，用来指示该键在哈希表中是否存在。由于多重返回值的工作方式，除非调用者用一个可以看见多值的形式显式地处理它，否则额外的返回值将被偷偷地丢掉。

我将在第20章里讨论更多关于多重返回值的细节，但目前我将概要地介绍一下如何使用**MULTIPLE-VALUE-BIND**宏来利用**GETHASH**额外返回值。**MULTIPLE-VALUE-BIND**创建类似于**LET**所做的变量绑定，并用一个形式返回的多个值来填充它们。

下面的函数显示了怎样使用**MULTIPLE-VALUE-BIND**，它绑定的变量是value和present：

```
(defun show-value (key hash-table)
  (multiple-value-bind (value present) (gethash key hash-table)
    (if present
      (format nil "Value ~a actually present." value)
      (format nil "Value ~a because key not found." value))))

(setf (gethash 'bar *h*) nil) ; provide an explicit value of NIL
```

① 由于历史上的意外，**GETHASH**的参数顺序与**ELT**相反。**ELT**将集合作为第一个参数，然后是索引，而**GETHASH**将键作为第一个参数，然后是集合。

```
(show-value 'foo *h*) → "Value QUUX actually present."
(show-value 'bar *h*) → "Value NIL actually present."
(show-value 'baz *h*) → "Value NIL because key not found."
```

由于将一个键下面的值设置成NIL会造成把键留在表中，因而你需要另一个函数来完全移除一个键值对。REMHASH接受和GETHASH相同的参数并移除指定的项。也可以使用CLRHASH来完全清除哈希表中的所有键值对。

11.12　哈希表迭代

Common Lisp提供了几种在哈希表项上迭代的方式，其中最简单的方式是通过函数MAPHASH。和MAP函数相似，MAPHASH接受一个两参数函数和一个哈希表，并在哈希表的每一个键值对上调用一次该函数。例如，为了打印哈希表中所有的键值对，可以像这样来使用MAPHASH：

```
(maphash #'(lambda (k v) (format t "~a => ~a~%" k v)) *h*)
```

在迭代一个哈希表的过程中，向其中添加或移除元素的后果没有被指定（并且可能会很坏），但有两个例外：可以将SETF与GETHASH一起使用来改变当前项的值，并且可以使用REMHASH来移除当前项。例如，为了移除所有其值小于10的项，可以写成下面这样：

```
(maphash #'(lambda (k v) (when (< v 10) (remhash k *h*))) *h*)
```

另一种在哈希表上迭代的方式是使用扩展的LOOP宏，我将在第22章里讨论它。[1]第一个MAPHASH表达式的等价LOOP形式如下所示：

```
(loop for k being the hash-keys in *h* using (hash-value v)
  do (format t "~a => ~a~%" k v))
```

关于Common Lisp所支持的非列表集合，我还可以讲更多的内容，例如多维数组以及处理位数组的函数库。但本章中涉及的内容已能满足多数通用编程场合的需要。现在，可以介绍列表这个让Lisp因此得名的数据结构了。

① LOOP的哈希表迭代通常是用更基本的形式WITH-HASH-TABLE-ITERATOR来实现的，你不需要担心这点。它被添加到语言里特定用来支持实现诸如LOOP这样的东西，并且除非你需要编写迭代在哈希表之上的全新控制构造，否则几乎不会用到它。