11. Collections

Like most programming languages, Common Lisp provides standard data types that collect multiple values into a single object. Every language slices up the collection problem a little bit differently, but the basic collection types usually boil down to an integer-indexed array type and a table type that can be used to map more or less arbitrary keys to values. The former are variously called *arrays*, *lists*, or *tuples*; the latter go by the names *hash tables*, *associative arrays*, *maps*, and *dictionaries*.

Lisp is, of course, famous for its list data structure, and most Lisp books, following the ontogeny-recapitulates-phylogeny principle of language instruction, start their discussion of Lisp's collections with lists. However, that approach often leads readers to the mistaken conclusion that lists are Lisp's *only* collection type. To make matters worse, because Lisp's lists are such a flexible data structure, it *is* possible to use them for many of the things arrays and hash tables are used for in other languages. But it's a mistake to focus too much on lists; while they're a crucial data structure for representing Lisp code as Lisp data, in many situations other data structures are more appropriate.

To keep lists from stealing the show, in this chapter I'll focus on Common Lisp's other collection types: vectors and hash tables.[1] However, vectors and lists share enough characteristics that Common Lisp treats them both as subtypes of a more general abstraction, the sequence. Thus, you can use many of the functions I'll discuss in this chapter with both vectors and lists.

Vectors

Vectors are Common Lisp's basic integer-indexed collection, and they come in two flavors. Fixed-size vectors are a lot like arrays in a language such as Java: a thin veneer over a chunk of contiguous memory that holds the vector's elements.[2] Resizable vectors, on the other hand, are more like arrays in Perl or Ruby, lists in Python, or the ArrayList class in Java: they abstract the actual storage, allowing the vector to grow and shrink as elements are added and removed.

You can make fixed-size vectors containing specific values with the function **`VECTOR`**, which takes any number of arguments and returns a freshly allocated fixed-size vector containing those arguments.

```
(vector)     ==> #()
(vector 1)   ==> #(1)
(vector 1 2) ==> #(1 2)
```

The `#(...)` syntax is the literal notation for vectors used by the Lisp printer and reader. This syntax allows you to save and restore vectors by **PRINT**ing them out and **READ**ing them back in. You can use the `#(...)` syntax to include literal vectors in your code, but as the effects of modifying literal objects aren't defined, you should always use **VECTOR** or the more general function **MAKE-ARRAY** to create vectors you plan to modify.

MAKE-ARRAY is more general than **VECTOR** since you can use it to create arrays of any dimensionality as well as both fixed-size and resizable vectors. The one required argument to **MAKE-ARRAY** is a list containing the dimensions of the array. Since a vector is a one-dimensional array, this list will contain one number, the size of the vector. As a convenience, **MAKE-ARRAY** will also accept a plain number in the place of a one-item list. With no other arguments, **MAKE-ARRAY** will create a vector with uninitialized elements that must be set before they can be accessed.[3] To create a vector with the elements all set to a particular value, you can pass an `:initial-element` argument. Thus, to make a five-element vector with its elements initialized to **NIL**, you can write the following:

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

MAKE-ARRAY is also the function to use to make a resizable vector. A resizable vector is a slightly more complicated object than a fixed-size vector; in addition to keeping track of the memory used to hold the elements and the number of slots available, a resizable vector also keeps track of the number of elements actually stored in the vector. This number is stored in the vector's *fill pointer*, so called because it's the index of the next position to be filled when you add an element to the vector.

To make a vector with a fill pointer, you pass **MAKE-ARRAY** a `:fill-pointer` argument. For instance, the following call to **MAKE-ARRAY** makes a vector with room for five elements; but it looks empty because the fill pointer is zero:

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

To add an element to the end of a resizable vector, you can use the function **VECTOR-PUSH**. It adds the element at the current value of the fill pointer and then increments the fill pointer by one, returning the index where the new element was added. The function **VECTOR-POP** returns the most recently pushed item, decrementing the fill pointer in the process.

```
(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)
```

```
(vector-pop *x*)      ==> A
*x*                   ==> #()
```

However, even a vector with a fill pointer isn't completely resizable. The vector `*x*` can hold at most five elements. To make an arbitrarily resizable vector, you need to pass **`MAKE-ARRAY`** another keyword argument: `:adjustable`.

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

This call makes an *adjustable* vector whose underlying memory can be resized as needed. To add elements to an adjustable vector, you use **`VECTOR-PUSH-EXTEND`**, which works just like **`VECTOR-PUSH`** except it will automatically expand the array if you try to push an element onto a full vector--one whose fill pointer is equal to the size of the underlying storage.[4]

Subtypes of Vector

All the vectors you've dealt with so far have been *general* vectors that can hold any type of object. It's also possible to create *specialized* vectors that are restricted to holding certain types of elements. One reason to use specialized vectors is they may be stored more compactly and can provide slightly faster access to their elements than general vectors. However, for the moment let's focus on a couple kinds of specialized vectors that are important data types in their own right.

One of these you've seen already--strings are vectors specialized to hold characters. Strings are important enough to get their own read/print syntax (double quotes) and the set of string-specific functions I discussed in the previous chapter. But because they're also vectors, all the functions I'll discuss in the next few sections that take vector arguments can also be used with strings. These functions will fill out the string library with functions for things such as searching a string for a substring, finding occurrences of a character within a string, and more.

Literal strings, such as `"foo"`, are like literal vectors written with the `#()` syntax--their size is fixed, and they must not be modified. However, you can use **`MAKE-ARRAY`** to make resizable strings by adding another keyword argument, `:element-type`. This argument takes a *type* descriptor. I won't discuss all the possible type descriptors you can use here; for now it's enough to know you can create a string by passing the symbol **`CHARACTER`** as the `:element-type` argument. Note that you need to quote the symbol to prevent it from being treated as a variable name. For example, to make an initially empty but resizable string, you can write this:

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

Bit vectors--vectors whose elements are all zeros or ones--also get some special treatment. They have a special read/print syntax that looks like `#*00001111` and a fairly large library of functions, which I won't discuss, for performing bit-twiddling operations such as "anding" together two bit arrays. The type descriptor to pass as the `:element-type` to create a bit vector is the symbol **`BIT`**.

Vectors As Sequences

As mentioned earlier, vectors and lists are the two concrete subtypes of the abstract type *sequence*. All the functions I'll discuss in the next few sections are sequence functions; in addition to being applicable to vectors--both general and specialized--they can also be used with lists.

The two most basic sequence functions are **LENGTH**, which returns the length of a sequence, and **ELT**, which allows you to access individual elements via an integer index. **LENGTH** takes a sequence as its only argument and returns the number of elements it contains. For vectors with a fill pointer, this will be the value of the fill pointer. **ELT**, short for *element*, takes a sequence and an integer index between zero (inclusive) and the length of the sequence (exclusive) and returns the corresponding element. **ELT** will signal an error if the index is out of bounds. Like **LENGTH**, **ELT** treats a vector with a fill pointer as having the length specified by the fill pointer.

```
(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 is also a **SETF**able place, so you can set the value of a particular element like this:

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

*x* ==> #(10 2 3)
```

Sequence Iterating Functions

While in theory all operations on sequences boil down to some combination of **LENGTH**, **ELT**, and **SETF** of **ELT** operations, Common Lisp provides a large library of sequence functions.

One group of sequence functions allows you to express certain operations on sequences such as finding or filtering specific elements without writing explicit loops. Table 11-1 summarizes them.

Table 11-1.Basic Sequence Functions

Name	Required Arguments	Returns
COUNT	Item and sequence	Number of times item appears in sequence
FIND	Item and sequence	Item or **NIL**
POSITION	Item and sequence	Index into sequence or **NIL**
REMOVE	Item and sequence	Sequence with instances of item removed
SUBSTITUTE	New item, item, and sequence	Sequence with instances of item replaced with new item

Here are some simple examples of how to use these functions:

```
(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
```

Note how **REMOVE** and **SUBSTITUTE** always return a sequence of the same type as their sequence argument.

You can modify the behavior of these five functions in a variety of ways using keyword arguments. For instance, these functions, by default, look for elements in the sequence that are the same object as the item argument. You can change this in two ways: First, you can use the `:test` keyword to pass a function that accepts two arguments and returns a boolean. If provided, it will be used to compare *item* to each element instead of the default object equality test, **EQL**.[5] Second, with the `:key` keyword you can pass a one-argument function to be called on each element of the sequence to extract a *key* value, which will then be compared to the item in the place of the element itself. Note, however, that functions such as **FIND** that return elements of the sequence continue to return the actual element, not just the extracted key.

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

To limit the effects of these functions to a particular subsequence of the sequence argument, you can provide bounding indices with `:start` and `:end` arguments. Passing **NIL** for `:end` or omitting it is the same as specifying the length of the sequence.[6]

If a non-**NIL** `:from-end` argument is provided, then the elements of the sequence will be examined in reverse order. By itself `:from-end` can affect the results of only **FIND** and **POSITION**. For instance:

```
(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)
```

However, the `:from-end` argument can affect **REMOVE** and **SUBSTITUTE** in conjunction with another keyword parameter, `:count`, that's used to specify how many elements to remove or substitute. If you specify a `:count` lower than the number of matching elements, then it obviously matters which end you start from:

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

And while `:from-end` can't change the results of the **COUNT** function, it does affect the order the elements are passed to any `:test` and `:key` functions, which could possibly have side effects. For example:

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

Table 11-2 summarizes these arguments.

Table 11-2. Standard Sequence Function Keyword Arguments

Argument	Meaning	Default
`:test`	Two-argument function used to compare item (or value extracted by `:key` function) to element.	**`EQL`**
`:key`	One-argument function to extract key value from actual sequence element. **`NIL`** means use element as is.	**`NIL`**
`:start`	Starting index (inclusive) of subsequence.	0
`:end`	Ending index (exclusive) of subsequence. **`NIL`** indicates end of sequence.	**`NIL`**
`:from-end`	If true, the sequence will be traversed in reverse order, from end to start.	**`NIL`**
`:count`	Number indicating the number of elements to remove or substitute or **`NIL`** to indicate all (**`REMOVE`** and **`SUBSTITUTE`** only).	**`NIL`**

Higher-Order Function Variants

For each of the functions just discussed, Common Lisp provides two *higher-order function* variants that, in the place of the item argument, take a function to be called on each element of the sequence. One set of variants are named the same as the basic function with an `-IF` appended. These functions count, find, remove, and substitute elements of the sequence for which the function argument returns true. The other set of variants are named with an `-IF-NOT` suffix and count, find, remove, and substitute elements for which the function argument does *not* return true.

```
(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")
```

According to the language standard, the **`-IF-NOT`** variants are deprecated. However, that deprecation is generally considered to have itself been ill-advised. If the standard is ever revised, it's more likely the deprecation will be removed than the **`-IF-NOT`** functions. For one thing, the **`REMOVE-IF-NOT`** variant is probably used more often than **`REMOVE-IF`**. Despite its negative-

sounding name, **REMOVE-IF-NOT** is actually the positive variant--it returns the elements that *do* satisfy the predicate. [7]

The `-IF` and `-IF-NOT` variants accept all the same keyword arguments as their vanilla counterparts except for `:test`, which isn't needed since the main argument is already a function.[8] With a `:key` argument, the value extracted by the `:key` function is passed to the function instead of the actual element.

```
(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")
```

The **REMOVE** family of functions also support a fourth variant, **REMOVE-DUPLICATES**, that has only one required argument, a sequence, from which it removes all but one instance of each duplicated element. It takes the same keyword arguments as **REMOVE**, except for `:count`, since it always removes all duplicates.

```
(remove-duplicates #(1 2 1 2 3 1 2 3 4)) ==> #(1 2 3 4)
```

Whole Sequence Manipulations

A handful of functions perform operations on a whole sequence (or sequences) at a time. These tend to be simpler than the other functions I've described so far. For instance, **COPY-SEQ** and **REVERSE** each take a single argument, a sequence, and each returns a new sequence of the same type. The sequence returned by **COPY-SEQ** contains the same elements as its argument while the sequence returned by **REVERSE** contains the same elements but in reverse order. Note that neither function copies the elements themselves--only the returned sequence is a new object.

The **CONCATENATE** function creates a new sequence containing the concatenation of any number of sequences. However, unlike **REVERSE** and **COPY-SEQ**, which simply return a sequence of the same type as their single argument, **CONCATENATE** must be told explicitly what kind of sequence to produce in case the arguments are of different types. Its first argument is a type descriptor, like the `:element-type` argument to **MAKE-ARRAY**. In this case, the type descriptors you'll most likely use are the symbols **VECTOR**, **LIST**, or **STRING**.[9] For example:

```
(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"
```

Sorting and Merging

The functions **SORT** and **STABLE-SORT** provide two ways of sorting a sequence. They both take a sequence and a two-argument predicate and return a sorted version of the sequence.

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

The difference is that **STABLE-SORT** is guaranteed to not reorder any elements considered equivalent by the predicate while **SORT** guarantees only that the result is sorted and may reorder equivalent elements.

Both these functions are examples of what are called *destructive* functions. Destructive functions are allowed--typically for reasons of efficiency--to modify their arguments in more or less arbitrary ways. This has two implications: one, you should always do something with the return value of these functions (such as assign it to a variable or pass it to another function), and, two, unless you're done with the object you're passing to the destructive function, you should pass a copy instead. I'll say more about destructive functions in the next chapter.

Typically you won't care about the unsorted version of a sequence after you've sorted it, so it makes sense to allow **SORT** and **STABLE-SORT** to destroy the sequence in the course of sorting it. But it does mean you need to remember to write the following:[10]

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

rather than just this:

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

Both these functions also take a keyword argument, `:key`, which, like the `:key` argument in other sequence functions, should be a function and will be used to extract the values to be passed to the sorting predicate in the place of the actual elements. The extracted keys are used only to determine the ordering of elements; the sequence returned will contain the actual elements of the argument sequence.

The **MERGE** function takes two sequences and a predicate and returns a sequence produced by merging the two sequences, according to the predicate. It's related to the two sorting functions in that if each sequence is already sorted by the same predicate, then the sequence returned by **MERGE** will also be sorted. Like the sorting functions, **MERGE** takes a `:key` argument. Like **CONCATENATE**, and for the same reason, the first argument to **MERGE** must be a type descriptor specifying the type of sequence to produce.

```
(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)
```

Subsequence Manipulations

Another set of functions allows you to manipulate subsequences of existing sequences. The most basic of these is **SUBSEQ**, which extracts a subsequence starting at a particular index and continuing to a particular ending index or the end of the sequence. For instance:

```
(subseq "foobarbaz" 3)   ==> "barbaz"
(subseq "foobarbaz" 3 6) ==> "bar"
```

SUBSEQ is also **SETF**able, but it won't extend or shrink a sequence; if the new value and the subsequence to be replaced are different lengths, the shorter of the two determines how many characters are actually changed.

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

(setf (subseq *x* 3 6) "xxx")  ; subsequence and new value are same length
*x* ==> "fooxxxbaz"

(setf (subseq *x* 3 6) "abcd") ; new value too long, extra character ignored.
*x* ==> "fooabcbaz"

(setf (subseq *x* 3 6) "xx")   ; new value too short, only two characters changed
*x* ==> "fooxxcbaz"
```

You can use the **FILL** function to set multiple elements of a sequence to a single value. The required arguments are a sequence and the value with which to fill it. By default every element of the sequence is set to the value; `:start` and `:end` keyword arguments can limit the effects to a given subsequence.

If you need to find a subsequence within a sequence, the **SEARCH** function works like **POSITION** except the first argument is a sequence rather than a single item.

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

On the other hand, to find where two sequences with a common prefix first diverge, you can use the **MISMATCH** function. It takes two sequences and returns the index of the first pair of mismatched elements.

```
(mismatch "foobarbaz" "foom") ==> 3
```

It returns **NIL** if the strings match. **MISMATCH** also takes many of the standard keyword arguments: a `:key` argument for specifying a function to use to extract the values to be compared; a `:test` argument to specify the comparison function; and `:start1`, `:end1`, `:start2`, and `:end2` arguments to specify subsequences within the two sequences. And a `:from-end` argument of **T** specifies the sequences should be searched in reverse order, causing **MISMATCH** to return the index, in the first sequence, where whatever common suffix the two sequences share begins.

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

Sequence Predicates

Four other handy functions are **EVERY**, **SOME**, **NOTANY**, and **NOTEVERY**, which iterate over sequences testing a boolean predicate. The first argument to all these functions is the predicate, and the remaining arguments are sequences. The predicate should take as many arguments as the number of sequences passed. The elements of the sequences are passed to the predicate--one element from each sequence--until one of the sequences runs out of elements or the overall

termination test is met: **EVERY** terminates, returning false, as soon as the predicate fails. If the predicate is always satisfied, it returns true. **SOME** returns the first non-**NIL** value returned by the predicate or returns false if the predicate is never satisfied. **NOTANY** returns false as soon as the predicate is satisfied or true if it never is. And **NOTEVERY** returns true as soon as the predicate fails or false if the predicate is always satisfied. Here are some examples of testing just one sequence:

```
(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
```

These calls compare elements of two sequences pairwise:

```
(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
```

Sequence Mapping Functions

Finally, the last of the sequence functions are the generic mapping functions. **MAP**, like the sequence predicate functions, takes a *n*-argument function and *n* sequences. But instead of a boolean value, **MAP** returns a new sequence containing the result of applying the function to subsequent elements of the sequences. Like **CONCATENATE** and **MERGE**, **MAP** needs to be told what kind of sequence to create.

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

MAP-INTO is like **MAP** except instead of producing a new sequence of a given type, it places the results into a sequence passed as the first argument. This sequence can be the same as one of the sequences providing values for the function. For instance, to sum several vectors--`a`, `b`, and `c`--into one, you could write this:

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

If the sequences are different lengths, **MAP-INTO** affects only as many elements as are present in the shortest sequence, including the sequence being mapped into. However, if the sequence being mapped into is a vector with a fill pointer, the number of elements affected isn't limited by the fill pointer but rather by the actual size of the vector. After a call to **MAP-INTO**, the fill pointer will be set to the number of elements mapped. **MAP-INTO** won't, however, extend an adjustable vector.

The last sequence function is **REDUCE**, which does another kind of mapping: it maps over a single sequence, applying a two-argument function first to the first two elements of the sequence and then to the value returned by the function and subsequent elements of the sequence. Thus, the following expression sums the numbers from one to ten:

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

REDUCE is a surprisingly useful function--whenever you need to distill a sequence down to a single value, chances are you can write it with **REDUCE**, and it will often be quite a concise way to express what you want. For instance, to find the maximum value in a sequence of numbers, you can write `(reduce #'max numbers)`. **REDUCE** also takes a full complement of keyword arguments (`:key`, `:from-end`, `:start`, and `:end`) and one unique to **REDUCE** (`:initial-value`). The latter specifies a value that's logically placed before the first element of the sequence (or after the last if you also specify a true `:from-end` argument).

Hash Tables

The other general-purpose collection provided by Common Lisp is the hash table. Where vectors provide an integer-indexed data structure, hash tables allow you to use arbitrary objects as the indexes, or keys. When you add a value to a hash table, you store it under a particular key. Later you can use the same key to retrieve the value. Or you can associate a new value with the same key--each key maps to a single value.

With no arguments **MAKE-HASH-TABLE** makes a hash table that considers two keys equivalent if they're the same object according to **EQL**. This is a good default unless you want to use strings as keys, since two strings with the same contents aren't necessarily **EQL**. In that case you'll want a so-called **EQUAL** hash table, which you can get by passing the symbol **EQUAL** as the `:test` keyword argument to **MAKE-HASH-TABLE**. Two other possible values for the `:test` argument are the symbols **EQ** and **EQUALP**. These are, of course, the names of the standard object comparison functions, which I discussed in Chapter 4. However, unlike the `:test` argument passed to sequence functions, **MAKE-HASH-TABLE**'s `:test` can't be used to specify an arbitrary function--only the values **EQ**, **EQL**, **EQUAL**, and **EQUALP**. This is because hash tables actually need two functions, an equivalence function and a *hash* function that computes a numerical hash code from the key in a way compatible with how the equivalence function will ultimately compare two keys. However, although the language standard provides only for hash tables that use the standard equivalence functions, most implementations provide some mechanism for defining custom hash tables.

The **GETHASH** function provides access to the elements of a hash table. It takes two arguments--a key and the hash table--and returns the value, if any, stored in the hash table under that key or **NIL**.[11] For example:

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

(gethash 'foo *h*) ==> NIL

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

(gethash 'foo *h*) ==> QUUX
```

Since **GETHASH** returns **NIL** if the key isn't present in the table, there's no way to tell from the return value the difference between a key not being in a hash table at all and being in the table with the value **NIL**. **GETHASH** solves this problem with a feature I haven't discussed yet--multiple return values. **GETHASH** actually returns two values; the primary value is the value stored under the given key or **NIL**. The secondary value is a boolean indicating whether the key is present in the hash table. Because of the way multiple values work, the extra return value is silently discarded unless the caller explicitly handles it with a form that can "see" multiple values.

I'll discuss multiple return values in greater detail in Chapter 20, but for now I'll give you a sneak preview of how to use the **MULTIPLE-VALUE-BIND** macro to take advantage of **GETHASH**'s extra return value. **MULTIPLE-VALUE-BIND** creates variable bindings like **LET** does, filling them with the multiple values returned by a form.

The following function shows how you might use **MULTIPLE-VALUE-BIND**; the variables it binds are `value` and `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

(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."
```

Since setting the value under a key to **NIL** leaves the key in the table, you'll need another function to completely remove a key/value pair. **REMHASH** takes the same arguments as **GETHASH** and removes the specified entry. You can also completely clear a hash table of all its key/value pairs with **CLRHASH**.

Hash Table Iteration

Common Lisp provides a couple ways to iterate over the entries in a hash table. The simplest of these is via the function **MAPHASH**. Analogous to the **MAP** function, **MAPHASH** takes a two-argument function and a hash table and invokes the function once for each key/value pair in the hash table. For instance, to print all the key/value pairs in a hash table, you could use **MAPHASH** like this:

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

The consequences of adding or removing elements from a hash table while iterating over it aren't specified (and are likely to be bad) with two exceptions: you can use **SETF** with **GETHASH** to change the value of the current entry, and you can use **REMHASH** to remove the current entry. For instance, to remove all the entries whose value is less than ten, you could write this:

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

The other way to iterate over a hash table is with the extended **LOOP** macro, which I'll discuss in Chapter 22.[12] The **LOOP** equivalent of the first **MAPHASH** expression would look like this:

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

I could say a lot more about the nonlist collections supported by Common Lisp. For instance, I haven't discussed multidimensional arrays at all or the library of functions for manipulating bit arrays. However, what I've covered in this chapter should suffice for most of your general-purpose programming needs. Now it's finally time to look at Lisp's eponymous data structure: lists.

---

[1]Once you're familiar with all the data types Common Lisp offers, you'll also see that lists can be useful for prototyping data structures that will later be replaced with something more efficient once it becomes clear how exactly the data is to be used.

[2]Vectors are called *vectors*, not *arrays* as their analogs in other languages are, because Common Lisp supports true multidimensional arrays. It's equally correct, though more cumbersome, to refer to them as *one-dimensional arrays*.

[3]Array elements "must" be set before they're accessed in the sense that the behavior is undefined; Lisp won't necessarily stop you.

[4]While frequently used together, the `:fill-pointer` and `:adjustable` arguments are independent--you can make an adjustable array without a fill pointer. However, you can use **VECTOR-PUSH** and **VECTOR-POP** only with vectors that have a fill pointer and **VECTOR-PUSH-EXTEND** only with vectors that have a fill pointer and are adjustable. You can also use the function **ADJUST-ARRAY** to modify adjustable arrays in a variety of ways beyond just extending the length of a vector.

[5]Another parameter, `:test-not` parameter, specifies a two-argument predicate to be used like a `:test` argument except with the boolean result logically reversed. This parameter is deprecated, however, in preference for using the **COMPLEMENT** function. **COMPLEMENT** takes a function argu-ment and returns a function that takes the same number of arguments as the original and returns the logical complement of the original function. Thus, you can, and should, write this:

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

rather than the following:

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

[6]Note, however, that the effect of `:start` and `:end` on **REMOVE** and **SUBSTITUTE** is only to limit the elements they consider for removal or substitution; elements before `:start` and after `:end` will be passed through untouched.

[7]This same functionality goes by the name `grep` in Perl and `filter` in Python.

[8]The difference between the predicates passed as `:test` arguments and as the function arguments to the `-IF` and `-IF-NOT` functions is that the `:test` predicates are two-argument predicates used to compare the elements of the sequence to the specific item while the `-IF` and `-IF-NOT` predicates are one-argument functions that simply test the individual elements of the sequence. If the vanilla variants didn't exist, you could implement them in terms of the -IF versions by embedding a specific item in the test function.

```
(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)
```

[9]If you tell **CONCATENATE** to return a specialized vector, such as a string, all the elements of the argument sequences must be instances of the vector's element type.

[10]When the sequence passed to the sorting functions is a vector, the "destruction" is actually guaranteed to entail permuting the elements in place, so you could get away without saving the returned value. However, it's good style to always do something with the return value since the sorting functions can modify lists in much more arbitrary ways.

[11]By an accident of history, the order of arguments to **GETHASH** is the opposite of **ELT**--**ELT** takes the collection first and then the index while **GETHASH** takes the key first and then the collection.

[12]**LOOP**'s hash table iteration is typically implemented on top of a more primitive form, **WITH-HASH-TABLE-ITERATOR**, that you don't need to worry about; it was added to the language specifically to support implementing things such as **LOOP** and is of little use unless you need to write completely new control constructs for iterating over hash tables.