

# **Programming Language—Common Lisp**

## **18. Hash Tables**

Version 15.17R, X3J13/94-101R.  
Fri 12-Aug-1994 6:35pm EDT

---

## 18.1 Hash Table Concepts

### 18.1.1 Hash-Table Operations

Figure 18–1 lists some *defined names* that are applicable to *hash tables*. The following rules apply to *hash tables*.

- A *hash table* can only associate one value with a given key. If an attempt is made to add a second value for a given key, the second value will replace the first. Thus, adding a value to a *hash table* is a destructive operation; the *hash table* is modified.
- There are four kinds of *hash tables*: those whose keys are compared with **eq**, those whose keys are compared with **eql**, those whose keys are compared with **equal**, and those whose keys are compared with **equalp**.
- *Hash tables* are created by **make-hash-table**. **gethash** is used to look up a key and find the associated value. New entries are added to *hash tables* using **setf** with **gethash**. **remhash** is used to remove an entry. For example:

```
(setq a (make-hash-table)) → #<HASH-TABLE EQL 0/120 32536573>
(setf (gethash 'color a) 'brown) → BROWN
(setf (gethash 'name a) 'fred) → FRED
(gethash 'color a) → BROWN, true
(gethash 'name a) → FRED, true
(gethash 'pointy a) → NIL, false
```

In this example, the symbols **color** and **name** are being used as keys, and the symbols **brown** and **fred** are being used as the associated values. The *hash table* has two items in it, one of which associates from **color** to **brown**, and the other of which associates from **name** to **fred**.

- A key or a value may be any *object*.
- The existence of an entry in the *hash table* can be determined from the *secondary value* returned by **gethash**.

<b>clrhash</b>	<b>hash-table-p</b>	<b>remhash</b>
<b>gethash</b>	<b>make-hash-table</b>	<b>sxhash</b>
<b>hash-table-count</b>	<b>maphash</b>	

Figure 18–1. Hash-table defined names

## 18.1.2 Modifying Hash Table Keys

The function supplied as the `:test` argument to **make-hash-table** specifies the ‘equivalence test’ for the *hash table* it creates.

An *object* is ‘visibly modified’ with regard to an equivalence test if there exists some set of *objects* (or potential *objects*) which are equivalent to the *object* before the modification but are no longer equivalent afterwards.

If an *object*  $O_1$  is used as a key in a *hash table*  $H$  and is then visibly modified with regard to the equivalence test of  $H$ , then the consequences are unspecified if  $O_1$ , or any *object*  $O_2$  equivalent to  $O_1$  under the equivalence test (either before or after the modification), is used as a key in further operations on  $H$ . The consequences of using  $O_1$  as a key are unspecified even if  $O_1$  is visibly modified and then later modified again in such a way as to undo the visible modification.

Following are specifications of the modifications which are visible to the equivalence tests which must be supported by *hash tables*. The modifications are described in terms of modification of components, and are defined recursively. Visible modifications of components of the *object* are visible modifications of the *object*.

### 18.1.2.1 Visible Modification of Objects with respect to EQ and EQL

No *standardized function* is provided that is capable of visibly modifying an *object* with regard to **eq** or **eql**.

### 18.1.2.2 Visible Modification of Objects with respect to EQUAL

As a consequence of the behavior for **equal**, the rules for visible modification of *objects* not explicitly mentioned in this section are inherited from those in Section 18.1.2.1 (Visible Modification of Objects with respect to EQ and EQL).

#### 18.1.2.2.1 Visible Modification of Conses with respect to EQUAL

Any visible change to the *car* or the *cdr* of a *cons* is considered a visible modification with regard to **equal**.

#### 18.1.2.2.2 Visible Modification of Bit Vectors and Strings with respect to EQUAL

For a *vector* of *type* **bit-vector** or of *type* **string**, any visible change to an *active element* of the *vector*, or to the *length* of the *vector* (if it is *actually adjustable* or has a *fill pointer*) is considered a visible modification with regard to **equal**.

### 18.1.2.3 Visible Modification of Objects with respect to EQUALP

As a consequence of the behavior for **equalp**, the rules for visible modification of *objects* not explicitly mentioned in this section are inherited from those in Section 18.1.2.2 (Visible Modification of Objects with respect to EQUAL).

#### 18.1.2.3.1 Visible Modification of Structures with respect to EQUALP

Any visible change to a *slot* of a *structure* is considered a visible modification with regard to **equalp**.

#### 18.1.2.3.2 Visible Modification of Arrays with respect to EQUALP

In an *array*, any visible change to an *active element*, to the *fill pointer* (if the *array* can and does have one), or to the *dimensions* (if the *array* is *actually adjustable*) is considered a visible modification with regard to **equalp**.

#### 18.1.2.3.3 Visible Modification of Hash Tables with respect to EQUALP

In a *hash table*, any visible change to the count of entries in the *hash table*, to the keys, or to the values associated with the keys is considered a visible modification with regard to **equalp**.

Note that the visibility of modifications to the keys depends on the equivalence test of the *hash table*, not on the specification of **equalp**.

#### 18.1.2.4 Visible Modifications by Language Extensions

*Implementations* that extend the language by providing additional mutator functions (or additional behavior for existing mutator functions) must document how the use of these extensions interacts with equivalence tests and *hash table* searches.

*Implementations* that extend the language by defining additional acceptable equivalence tests for *hash tables* (allowing additional values for the **:test** argument to **make-hash-table**) must document the visible components of these tests.

---

## hash-table

*System Class*

---

### Class Precedence List:

hash-table, t

### Description:

*Hash tables* provide a way of mapping any *object* (a *key*) to an associated *object* (a *value*).

### See Also:

Section 18.1 (Hash Table Concepts), Section 22.1.3.13 (Printing Other Objects)

### Notes:

The intent is that this mapping be implemented by a hashing mechanism, such as that described in Section 6.4 “Hashing” of *The Art of Computer Programming, Volume 3* (pp506-549). In spite of this intent, no *conforming implementation* is required to use any particular technique to implement the mapping.

---

## make-hash-table

*Function*

---

### Syntax:

`make-hash-table &key test size rehash-size rehash-threshold` → *hash-table*

### Arguments and Values:

*test*—a *designator* for one of the functions `eq`, `eql`, `equal`, or `equalp`. The default is `eql`.

*size*—a non-negative *integer*. The default is *implementation-dependent*.

*rehash-size*—a *real* of *type* (or (integer 1 \*) (float (1.0) \*)). The default is *implementation-dependent*.

*rehash-threshold*—a *real* of *type* (real 0 1). The default is *implementation-dependent*.

*hash-table*—a *hash table*.

### Description:

Creates and returns a new *hash table*.

*test* determines how *keys* are compared. An *object* is said to be present in the *hash-table* if that *object* is the *same* under the *test* as the *key* for some entry in the *hash-table*.

*size* is a hint to the *implementation* about how much initial space to allocate in the *hash-table*. This information, taken together with the *rehash-threshold*, controls the approximate number of entries which it should be possible to insert before the table has to grow. The actual size might

---

be rounded up from *size* to the next ‘good’ size; for example, some *implementations* might round to the next prime number.

*rehash-size* specifies a minimum amount to increase the size of the *hash-table* when it becomes full enough to require rehashing; see *rehash-threshold* below. If *rehash-size* is an *integer*, the expected growth rate for the table is additive and the *integer* is the number of entries to add; if it is a *float*, the expected growth rate for the table is multiplicative and the *float* is the ratio of the new size to the old size. As with *size*, the actual size of the increase might be rounded up.

*rehash-threshold* specifies how full the *hash-table* can get before it must grow. It specifies the maximum desired hash-table occupancy level.

The *values* of *rehash-size* and *rehash-threshold* do not constrain the *implementation* to use any particular method for computing when and by how much the size of *hash-table* should be enlarged. Such decisions are *implementation-dependent*, and these *values* only hints from the *programmer* to the *implementation*, and the *implementation* is permitted to ignore them.

### Examples:

```
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 46142754>
(setf (gethash "one" table) 1) → 1
(gethash "one" table) → NIL, false
(setq table (make-hash-table :test 'equal)) → #<HASH-TABLE EQUAL 0/139 46145547>
(setf (gethash "one" table) 1) → 1
(gethash "one" table) → 1, T
(make-hash-table :rehash-size 1.5 :rehash-threshold 0.7)
→ #<HASH-TABLE EQL 0/120 46156620>
```

### See Also:

gethash, hash-table

---

## hash-table-p

*Function*

---

### Syntax:

hash-table-p *object* → *generalized-boolean*

### Arguments and Values:

*object*—an *object*.

*generalized-boolean*—a *generalized boolean*.

### Description:

Returns *true* if *object* is of *type* **hash-table**; otherwise, returns *false*.

---

## Examples:

```
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32511220>
(hash-table-p table) → true
(hash-table-p 37) → false
(hash-table-p '((a . 1) (b . 2))) → false
```

## Notes:

```
(hash-table-p object) ≡ (typep object 'hash-table)
```

---

# hash-table-count

*Function*

---

## Syntax:

```
hash-table-count hash-table → count
```

## Arguments and Values:

*hash-table*—a *hash table*.

*count*—a non-negative *integer*.

## Description:

Returns the number of entries in the *hash-table*. If *hash-table* has just been created or newly cleared (see `clrhash`) the entry count is 0.

## Examples:

```
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32115135>
(hash-table-count table) → 0
(setf (gethash 57 table) "fifty-seven") → "fifty-seven"
(hash-table-count table) → 1
(dotimes (i 100) (setf (gethash i table) i)) → NIL
(hash-table-count table) → 100
```

## Affected By:

`clrhash`, `remhash`, `setf` of `gethash`

## See Also:

`hash-table-size`



---

## Notes:

The following relationships are functionally correct, although in practice using **hash-table-count** is probably much faster:

```
(hash-table-count table) ≡  
(loop for value being the hash-values of table count t) ≡  
(let ((total 0))  
  (maphash #'(lambda (key value)  
                (declare (ignore key value))  
                (incf total))  
    table)  
  total)
```

---

## hash-table-rehash-size

*Function*

---

### Syntax:

**hash-table-rehash-size** *hash-table* → *rehash-size*

### Arguments and Values:

*hash-table*—a *hash table*.

*rehash-size*—a *real* of *type* (or (integer 1 \*) (float (1.0 \*))).

### Description:

Returns the current rehash size of *hash-table*, suitable for use in a call to **make-hash-table** in order to produce a *hash table* with state corresponding to the current state of the *hash-table*.

### Examples:

```
(setq table (make-hash-table :size 100 :rehash-size 1.4))  
→ #<HASH-TABLE EQL 0/100 2556371>  
(hash-table-rehash-size table) → 1.4
```

### Exceptional Situations:

Should signal an error of *type* **type-error** if *hash-table* is not a *hash table*.

### See Also:

**make-hash-table**, **hash-table-rehash-threshold**

### Notes:

If the hash table was created with an *integer* rehash size, the result is an *integer*, indicating that the rate of growth of the *hash-table* when rehashed is intended to be additive; otherwise, the

---

result is a *float*, indicating that the rate of growth of the *hash-table* when rehashed is intended to be multiplicative. However, this value is only advice to the *implementation*; the actual amount by which the *hash-table* will grow upon rehash is *implementation-dependent*.

---

## hash-table-rehash-threshold

*Function*

---

### Syntax:

`hash-table-rehash-threshold hash-table → rehash-threshold`

### Arguments and Values:

*hash-table*—a *hash table*.

*rehash-threshold*—a *real* of *type* `(real 0 1)`.

### Description:

Returns the current rehash threshold of *hash-table*, which is suitable for use in a call to **make-hash-table** in order to produce a *hash table* with state corresponding to the current state of the *hash-table*.

### Examples:

```
(setq table (make-hash-table :size 100 :rehash-threshold 0.5))  
→ #<HASH-TABLE EQL 0/100 2562446>  
(hash-table-rehash-threshold table) → 0.5
```

### Exceptional Situations:

Should signal an error of *type* **type-error** if *hash-table* is not a *hash table*.

### See Also:

**make-hash-table**, **hash-table-rehash-size**

---

---

## hash-table-size

---

*Function*

### Syntax:

`hash-table-size hash-table`  $\rightarrow$  *size*

### Arguments and Values:

*hash-table*—a *hash table*.

*size*—a non-negative *integer*.

### Description:

Returns the current size of *hash-table*, which is suitable for use in a call to **make-hash-table** in order to produce a *hash table* with state corresponding to the current state of the *hash-table*.

### Exceptional Situations:

Should signal an error of type **type-error** if *hash-table* is not a *hash table*.

### See Also:

`hash-table-count`, `make-hash-table`

---

## hash-table-test

---

*Function*

### Syntax:

`hash-table-test hash-table`  $\rightarrow$  *test*

### Arguments and Values:

*hash-table*—a *hash table*.

*test*—a *function designator*. For the four *standardized hash table test functions* (see **make-hash-table**), the *test* value returned is always a *symbol*. If an *implementation* permits additional tests, it is *implementation-dependent* whether such tests are returned as *function objects* or *function names*.

### Description:

Returns the test used for comparing *keys* in *hash-table*.

### Exceptional Situations:

Should signal an error of type **type-error** if *hash-table* is not a *hash table*.

### See Also:

`make-hash-table`

---

## gethash

---

*Accessor*

### Syntax:

`gethash key hash-table &optional default` → *value*, *present-p*  
(`setf (gethash key hash-table &optional default) new-value`)

### Arguments and Values:

*key*—an *object*.

*hash-table*—a *hash table*.

*default*—an *object*. The default is `nil`.

*value*—an *object*.

*present-p*—a *generalized boolean*.

### Description:

*Value* is the *object* in *hash-table* whose *key* is the *same* as *key* under the *hash-table*'s equivalence test. If there is no such entry, *value* is the *default*.

*Present-p* is *true* if an entry is found; otherwise, it is *false*.

`setf` may be used with `gethash` to modify the *value* associated with a given *key*, or to add a new entry. When a `gethash form` is used as a `setf place`, any *default* which is supplied is evaluated according to normal left-to-right evaluation rules, but its *value* is ignored.

### Examples:

```
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32206334>
(gethash 1 table) → NIL, false
(gethash 1 table 2) → 2, false
(setf (gethash 1 table) "one") → "one"
(setf (gethash 2 table "two") "two") → "two"
(gethash 1 table) → "one", true
(gethash 2 table) → "two", true
(gethash nil table) → NIL, false
(setf (gethash nil table) nil) → NIL
(gethash nil table) → NIL, true
(defvar *counters* (make-hash-table)) → *COUNTERS*
(gethash 'foo *counters*) → NIL, false
(gethash 'foo *counters* 0) → 0, false
```

---

```
(defmacro how-many (obj) `(values (gethash ,obj *counters* 0))) → HOW-MANY
(defun count-it (obj) (incf (how-many obj))) → COUNT-IT
(dolist (x '(bar foo foo bar bar baz)) (count-it x))
(how-many 'foo) → 2
(how-many 'bar) → 3
(how-many 'quux) → 0
```

**See Also:**

remhash

**Notes:**

The *secondary value*, *present-p*, can be used to distinguish the absence of an entry from the presence of an entry that has a value of *default*.

---

## remhash

*Function*

---

**Syntax:**

remhash *key hash-table* → *generalized-boolean*

**Arguments and Values:**

*key*—an *object*.

*hash-table*—a *hash table*.

*generalized-boolean*—a *generalized boolean*.

**Description:**

Removes the entry for *key* in *hash-table*, if any. Returns *true* if there was such an entry, or *false* otherwise.

**Examples:**

```
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32115666>
(setf (gethash 100 table) "C") → "C"
(gethash 100 table) → "C", true
(remhash 100 table) → true
(gethash 100 table) → NIL, false
(remhash 100 table) → false
```

**Side Effects:**

The *hash-table* is modified.

---

## maphash

---

### maphash

*Function*

---

#### Syntax:

`maphash function hash-table` → nil

#### Arguments and Values:

*function*—a *designator* for a *function* of two *arguments*, the *key* and the *value*.

*hash-table*—a *hash table*.

#### Description:

Iterates over all entries in the *hash-table*. For each entry, the *function* is called with two *arguments*—the *key* and the *value* of that entry.

The consequences are unspecified if any attempt is made to add or remove an entry from the *hash-table* while a **maphash** is in progress, with two exceptions: the *function* can use **setf** of **gethash** to change the *value* part of the entry currently being processed, or it can use **remhash** to remove that entry.

#### Examples:

```
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32304110>
(dotimes (i 10) (setf (gethash i table) i)) → NIL
(let ((sum-of-squares 0))
  (maphash #'(lambda (key val)
               (let ((square (* val val)))
                 (incf sum-of-squares square)
                 (setf (gethash key table) square)))
    table)
  sum-of-squares) → 285
(hash-table-count table) → 10
(maphash #'(lambda (key val)
              (when (oddp val) (remhash key table)))
  table) → NIL
(hash-table-count table) → 5
(maphash #'(lambda (k v) (print (list k v))) table)
(0 0)
(8 64)
(2 4)
(6 36)
(4 16)
→ NIL
```

#### Side Effects:

None, other than any which might be done by the *function*.

---

**See Also:**

`loop`, `with-hash-table-iterator`, Section 3.6 (Traversal Rules and Side Effects)

---

## **with-hash-table-iterator**

*Macro*

---

**Syntax:**

`with-hash-table-iterator` (*name hash-table*) {*declaration*}\* {*form*}\*  $\rightarrow$  {*result*}\*

**Arguments and Values:**

*name*—a name suitable for the first argument to `macrolet`.

*hash-table*—a *form*, evaluated once, that should produce a *hash table*.

*declaration*—a `declare expression`; not evaluated.

*forms*—an *implicit progn*.

*results*—the *values* returned by *forms*.

**Description:**

Within the lexical scope of the body, *name* is defined via `macrolet` such that successive invocations of (*name*) return the items, one by one, from the *hash table* that is obtained by evaluating *hash-table* only once.

An invocation (*name*) returns three values as follows:

1. A *generalized boolean* that is *true* if an entry is returned.
2. The key from the *hash-table* entry.
3. The value from the *hash-table* entry.

After all entries have been returned by successive invocations of (*name*), then only one value is returned, namely `nil`.

It is unspecified what happens if any of the implicit interior state of an iteration is returned outside the dynamic extent of the `with-hash-table-iterator` *form* such as by returning some *closure* over the invocation *form*.

Any number of invocations of `with-hash-table-iterator` can be nested, and the body of the innermost one can invoke all of the locally *established macros*, provided all of those *macros* have *distinct* names.

**Examples:**

The following function should return `t` on any *hash table*, and signal an error if the usage of `with-hash-table-iterator` does not agree with the corresponding usage of `maphash`.

```
(defun test-hash-table-iterator (hash-table)
  (let ((all-entries '())
        (generated-entries '())
        (unique (list nil)))
    (maphash #'(lambda (key value) (push (list key value) all-entries))
              hash-table)
    (with-hash-table-iterator (generator-fn hash-table)
      (loop
        (multiple-value-bind (more? key value) (generator-fn)
          (unless more? (return))
          (unless (eql value (gethash key hash-table unique))
            (error "Key ~S not found for value ~S" key value))
          (push (list key value) generated-entries))))
    (unless (= (length all-entries)
               (length generated-entries)
               (length (union all-entries generated-entries
                              :key #'car :test (hash-table-test hash-table))))
      (error "Generated entries and Maphash entries don't correspond"))
    t))
```

The following could be an acceptable definition of **maphash**, implemented by **with-hash-table-iterator**.

```
(defun maphash (function hash-table)
  (with-hash-table-iterator (next-entry hash-table)
    (loop (multiple-value-bind (more key value) (next-entry)
          (unless more (return nil))
          (funcall function key value)))))
```

### Exceptional Situations:

The consequences are undefined if the local function named *name established* by **with-hash-table-iterator** is called after it has returned *false* as its *primary value*.

### See Also:

Section 3.6 (Traversal Rules and Side Effects)

---

## clrhash

*Function*

---

### Syntax:

`clrhash hash-table`  $\rightarrow$  *hash-table*

### Arguments and Values:

*hash-table*—a *hash table*.



---

## Description:

Removes all entries from *hash-table*, and then returns that empty *hash table*.

## Examples:

```
(setq table (make-hash-table)) → #<HASH-TABLE EQL 0/120 32004073>
(dotimes (i 100) (setf (gethash i table) (format nil "~R" i))) → NIL
(hash-table-count table) → 100
(gethash 57 table) → "fifty-seven", true
(clearhash table) → #<HASH-TABLE EQL 0/120 32004073>
(hash-table-count table) → 0
(gethash 57 table) → NIL, false
```

## Side Effects:

The *hash-table* is modified.

---

# sxhash

*Function*

---

## Syntax:

`sxhash object` → *hash-code*

## Arguments and Values:

*object*—an *object*.

*hash-code*—a non-negative *fixnum*.

## Description:

`sxhash` returns a hash code for *object*.

The manner in which the hash code is computed is *implementation-dependent*, but subject to certain constraints:

1. `(equal x y)` implies `(= (sxhash x) (sxhash y))`.
2. For any two *objects*, *x* and *y*, both of which are *bit vectors*, *characters*, *conses*, *numbers*, *pathnames*, *strings*, or *symbols*, and which are *similar*, `(sxhash x)` and `(sxhash y)` yield the same mathematical value even if *x* and *y* exist in different *Lisp images* of the same *implementation*. See Section 3.2.4 (Literal Objects in Compiled Files).
3. The *hash-code* for an *object* is always the *same* within a single *session* provided that the *object* is not visibly modified with regard to the equivalence test **equal**. See Section 18.1.2 (Modifying Hash Table Keys).

## sxhash

---

4. The *hash-code* is intended for hashing. This places no verifiable constraint on a *conforming implementation*, but the intent is that an *implementation* should make a good-faith effort to produce *hash-codes* that are well distributed within the range of non-negative *fixnums*.
5. Computation of the *hash-code* must terminate, even if the *object* contains circularities.

### Examples:

```
(= (sxhash (list 'list "ab")) (sxhash (list 'list "ab"))) → true
(= (sxhash "a") (sxhash (make-string 1 :initial-element #\a))) → true
(let ((r (make-random-state)))
  (= (sxhash r) (sxhash (make-random-state r))))
→ implementation-dependent
```

### Affected By:

The *implementation*.

### Notes:

Many common hashing needs are satisfied by **make-hash-table** and the related functions on *hash tables*. **sxhash** is intended for use where the pre-defined abstractions are insufficient. Its main intent is to allow the user a convenient means of implementing more complicated hashing paradigms than are provided through *hash tables*.

The hash codes returned by **sxhash** are not necessarily related to any hashing strategy used by any other *function* in Common Lisp.

For *objects* of *types* that **equal** compares with **eq**, item 3 requires that the *hash-code* be based on some immutable quality of the identity of the object. Another legitimate implementation technique would be to have **sxhash** assign (and cache) a random hash code for these *objects*, since there is no requirement that *similar* but non-**eq** objects have the same hash code.

Although *similarity* is defined for *symbols* in terms of both the *symbol's name* and the *packages* in which the *symbol* is *accessible*, item 3 disallows using *package* information to compute the hash code, since changes to the package status of a symbol are not visible to *equal*.

---