User Guide for Common Lisp Package `make-hash`

Christopher Genovese (`genovese@cmu.edu`)

30 Jun 2012

Contents

1 Motivation and Overview

Two common (and arguably apt) criticisms of hash tables in Common Lisp are that hash table initialization is bulky and awkward and that the representation of hash tables is not as integrated into the language as are the representations of lists and (to a degree) vectors.

The `make-hash` package addresses these issues by supplying three useful, related mechanisms:

1. A hash table constructor `make-hash` with initialization that is concise, flexible, and extensible.

   See `make-hash`, `initialize-hash`, `hash-initializer-default-format`, and `make-hash-transformer` below.

2. Methods for defining hash-table factories with a customized set of initialization options, either as a globally or locally defined function.

   See `define-hash-factory`, `make-hash-factory`, and `*hash-factory-defaults*` below.

3. Readtable installers for defining a portable reader interface to the hash-table factories, either as (raw) delimited or dispatched reader macros.

   See `install-hash-reader` below.

In particular, the function `make-hash` is a wrapper around the standard CL function `make-hash-table` with some additional keyword arguments that allow one to specify initial contents and format.

As an illustation, consider the example on page 440 of the venerable *Common Lisp the Language, Second Edition* by Guy Steele [CLtL2].

```
(setq turtles (make-hash-table :size 9 :test 'eq))
(setf (gethash 'howard-kaylan turtles) '(musician lead-singer))
(setf (gethash 'john-barbata turtles) '(musician drummer))
(setf (gethash 'leonardo turtles) '(ninja leader blue))
(setf (gethash 'donatello turtles) '(ninja machines purple))
(setf (gethash 'al-nichol turtles) '(musician guitarist))
(setf (gethash 'mark-volman turtles) '(musician great-hair))
(setf (gethash 'raphael turtles) '(ninja cool rude red))
(setf (gethash 'michaelangelo turtles) '(ninja party-dude orange))
(setf (gethash 'jim-pons turtles) '(musician bassist))
```

This is not horrible by any means, but the repeated `setf`'s force an assignment-oriented block of statements and visually obscure the relationships in the table. And in practice, even more syntactic infrastructure is usually required (e.g., another level of let for a local definition, a loop for a larger hash table). While it is certainly a matter of taste which form one prefers, the goal of `make-hash` is to allow a more convenient, functional-style hash table construction that is consistent with constructors for lists, vectors, and arrays. Compare the above with any of the following:

```
(make-hash :size 9 :test 'eq
           :initial-contents '(howard-kaylan (musician lead-singer)
                               jon-barbata   (musician drummer)
                               leonardo      (ninja leader blue)
                               donatello     (ninja machines purple)
                               al-nichol     (musician guitarist)
                               mark-volman   (musician great-hair)
                               raphael       (ninja cool rude red)
                               michaelangelo (ninja party-dude orange)
                               jim-pons      (musician bassist)))

(make-hash :size 9 :test 'eq :init-format :lists
           :initial-contents '((howard-kaylan (musician lead-singer))
                               (jon-barbata   (musician drummer))
                               (leonardo      (ninja leader blue))
                               (donatello     (ninja machines purple))
                               (al-nichol     (musician guitarist))
                               (mark-volman   (musician great-hair))
                               (raphael       (ninja cool rude red))
                               (michaelangelo (ninja party-dude orange))
                               (jim-pons      (musician bassist))))

(make-hash :size 9 :test 'eq :init-format :keychain
           :initial-contents
           '(howard-kaylan jon-barbata leonardo
             donatello al-nichol mark-volman
             raphael michaelangelo jim-pons)
           :init-data
           '((musician lead-singer) (musician drummer) (ninja leader blue)
             (ninja machines purple) (musician guitarist) (musician great-hair)
             (ninja cool rude red) (ninja party-dude orange) (musician bassist)))
```

```
#{ howard-kaylan (musician lead-singer)
   jon-barbata   (musician drummer)
   leonardo      (ninja leader blue)
   donatello     (ninja machines purple)
   al-nichol     (musician guitarist)
   mark-volman   (musician great-hair)
   raphael       (ninja cool rude red)
   michaelangelo (ninja party-dude orange)
   jim-pons      (musician bassist) }
```

There are many other formats for the initial contents that would be convenient to use in other contexts, and make-hash supports a wide variety of them. Moreover, custom formats can be supported easily by defining a method for a single generic function, and default formats can be adjusted similarly. See below for more detail and examples.

2 Installation

The simplest approach is to use quicklisp (www.quicklisp.org). With quicklisp installed, simply call `(ql:quickload "make-hash")` and quicklisp will do the rest.

Otherwise, obtain the code from http://github.com/genovese/make-hash, cloning the repository or downloading and unpacking the tar/zip archive. Either load it directly or put the `make-hash` subdirectory where ASDF (www.cliki.net/asdf) can find the `.asd` file. With ASDF, call `(asdf:load-system "make-hash")` to load the package.

For both quicklisp and ASDF, you may want to call `(use-package :make-hash)` to import the main functions. If you want to run the tests, which are in the package `make-hash-tests`, do the following.

- In quicklisp:

```
(ql:quickload "make-hash")
(ql:quickload "make-hash-tests")
(asdf:test-system "make-hash-tests")
```

- With ASDF alone:

```
(asdf:load-system "make-hash")
(asdf:load-system "make-hash-tests")
(asdf:test-system "make-hash-tests")
```

3 Examples

The use of `make-hash` is pretty straightforward, and I think it will be clearer to see some examples before looking at the detailed specifications. It might help to scan these examples quickly on first read through and then come back after reading the specification in the ensuing sections. Here, I will assume that the predefined formats and defaults are in effect, although these can be overridden if desired.

1. No Initialization

   Use exactly like `make-hash-table`, with all standard or implementation-dependent keyword arguments.

   ```
   (make-hash)
   (make-hash :test #'equal)
   (make-hash :size 128 :rehash-size 1.75)
   ```

2. (Shallow) Copying an existing hash table

   ```
   (make-hash :initial-contents eql-hash-table)
   (make-hash :test (hash-table-test other-hash-table)
              :initial-contents other-hash-table)
   ```

3. Initializing from simple sequences containing keys and values

   ```
   (make-hash :initial-contents '(a 1 b 2 c 3 d 1 e 2 f 3 g 4))
   (make-hash :init-format :flat
              :initial-contents '(a 1 b 2 c 3 d 1 e 2 f 3 g 4))
   (make-hash :init-format :pairs
              :initial-contents '((a . 1) (b . 2) (c . 3)
                                  (d . 1) (e . 2) (f . 3) (g . 4)))
   (make-hash :init-format :lists
              :initial-contents '((a 1) (b 2) (c 3)
                                  (d 1) (e 2) (f 3) (g 4)))
   (make-hash :init-format :vectors
              :initial-contents '(#(a 1) #(b 2) #(c 3)
                                  #(d 1) #(e 2) #(f 3) #(g 4)))
   (make-hash :init-format :seqs
              :initial-contents '((a 1) #(b 2) (c 3)
                                  #(d 1) (e 2) #(f 3) #(g 4)))
   ```

Here `:flat` is the default format, and the result in all these cases maps `a` → 1, `b` → 2, `c` → 3, `d` → 1, `e` → 2, `f` → 3, and `g` → 4.

4. Initializing from separate sequences of keys and values

```
(make-hash :init-format :keychain
           :initial-contents '(a b c d e f g)
           :init-data        '(1 2 3 1 2 3 4))
(make-hash :init-format :keychain
           :initial-contents '(a b c d e f g)
           :init-data        #(1 2 3 1 2 3 4))
```

   The resulting tables are the same as in the last example.

5. Creating a hash table of keys and counts

   Given a sequence of objects, create a hash table with the unique objects as keys and the frequency counts in the sequence as values.

```
(make-hash :init-format :keybag
           :initial-contents '(a b d d e c b a a e c c d a a c c e c c))
(make-hash :init-format :keybag
           :initial-contents #(a b d d e c b a a e c c d a a c c e c c))
```

   The results map `a` → 5, `b` → 2, `c` → 7, `d` → 3, and `e` → 3.

6. Building a hash from selected keys in another associative map or database

   Here, the `:initial-contents` is a sequence of keys, and the corresponding values are the values for those keys in the map given as `:init-data`, or the `:init-default` if none exists.

   Let `turtles` be the hash table above from CLtL2. Suppose `turtles-alist` is an associative list with the same data and that `turtles-database-reader` is a function that reads an associated record from a database. We can extract a "sub-hash" whose keys are those corresponding to mutant, ninja turtles as follows.

```
(make-hash :init-format :keys
           :initial-contents '(leonardo donatello raphael michaelangelo)
           :init-data turtles)
```

```
(make-hash :init-format :keys
           :initial-contents '(leonardo donatello raphael michaelangelo)
           :init-data turtles-alist)
(make-hash :init-format :keys
           :initial-contents '(leonardo donatello raphael michaelangelo)
           :init-data turtles-database-reader)
```

7. Initializing from repeated calls to a function

   The following initializes the hash table from a *simple* CSV (comma-separated value) file, with no commas within fields, using the first field as the key and the list of remaining fields as the value. The function `parse-csv-line` acts on one line at a time, skipping and either initializes or skips using the return value convention described below.

```
(use-package :cl-ppcre)

(defun parse-csv-line (stream)
  (let ((line (read-line stream nil)))
    (cond
      ((null line)
       (values nil nil nil))
      ((scan "^\\s*$" line)
       (values t t t))
      (t
       (let ((fields
              (split "\\s*,\\s*"  line :limit most-positive-fixnum)))
         (values (first fields) (rest fields) nil))))))

(with-open-file (s "data.csv" :direction :input :if-does-not-exist nil)
  (make-hash :test #'equal :init-format :function
             :initial-contents #'parse-csv-line :init-data (list s)))
```

   The following initializes the hash table from the key-value pairs in an INI file. The function `parse-ini-line` is acts on one line at a time and either initializes or skips using the return value convention described below.

```
(use-package :cl-ppcre)
```

```
(let ((ini-line-re
       (create-scanner
        "^\\s*(?:|;.*|\\[([^]]+)\\]|(\\w+)\\s*=\\s*(.*?))?\\s*$"))
      (current-section-name ""))
  (defun parse-ini-line (stream)
    (let ((line (read-line stream nil)))
      (unless line
        (setf current-section-name "")
        (return-from parse-ini (values nil nil nil)))
      (multiple-value-bind (beg end reg-begs reg-ends)
          (scan ini-line-re line)
        (declare (ignorable end))
        (unless beg
          (error "Improperly formatted INI line: ~A" line))
        (if (and (> (length reg-begs) 2) (aref reg-begs 1))
            (values
             (concatenate 'string
                          current-section-name "/"
                          (subseq line (aref reg-begs 1) (aref reg-ends 1)))
             (subseq line (aref reg-begs 2) (aref reg-ends 2))
             nil)
            (progn
              (when (and (> (length reg-begs) 0) (aref reg-begs 0))
                (setf current-section-name
                      (subseq line (aref reg-begs 0) (aref reg-ends 0))))
              (values t t t)))))))

(with-open-file (s "config.ini" :direction :input :if-does-not-exist nil)
  (make-hash :test #'equal :init-format :function
             :initial-contents #'parse-ini-line :init-data (list s)))
```

8. Transforming a hash built from a sequence of keys and values

   Passing a function as `:init-data` can be used to transform the initial contents as the hash is being initialized.

```
(make-hash :init-format :flat
           :initial-contents '(a 1 b 2 c 3 d 1 e 2 f 3 g 4)
           :init-data (lambda (k v) (values k (* v v) nil)))
```

```
(make-hash :init-format :pairs
           :initial-contents '((a . 1) (b . 2) (c . 3)
                               (d . 1) (e . 2) (f . 3) (g . 4))
           :init-data (lambda (k v)
                        (values (intern (symbol-name k) :keyword)
                                (* v v))))
(let ((scratch (make-hash)))
  (make-hash :init-format :lists
             :initial-contents '((a 1) (b 2) (c 3)
                                 (d 1) (e 2) (f 3) (g 4))
             :init-data (lambda (k v)
                          (values v
                                  (setf (gethash v scratch)
                                        (cons k (gethash v scratch nil)))
                                  nil))))
```

The first is a hash that maps `a` and `d` to 1, `b` and `e` to 4, `c` and `f` to 9, and `g` to 16. The second is the same except that the keys are the keywords with the same symbol-name (e.g., :a, :b). The third reverses the given alist, accumulated repeated values in a list: 1 → `(d a)`, 2 → `(e b)`, 3 → `(f c)`, and 4 → `(g)`.

9. Transforming an existing hash table or alist

```
(defun lastcar (list)
  (car (last list)))

(defvar *pet-hash*
  (make-hash :initial-contents
             '(dog  (mammal pet loyal 3) cat (mammal pet independent 1)
               eagle 0 cobra 0
               goldfish (fish pet flushed 1) hamster (mammal pet injured sad 2)
               corn-snake (reptile pet dog-like 1) crab (crustacean quiet 4)
               grasshopper (insect methusala 1) black-widow 0)))

(make-hash :initial-contents *pet-hash*
           :init-data (make-hash-transformer :value #'lastcar #'atom))
```

The result maps `dog` → 3, `cat` → 1, `goldfish` → 1, `hamster` → 2, `corn-snake` → 1, `grasshopper` → 1, and `crab` → 4. If `*pet-hash*`

had been an alist instead of a hash table, the call to `make-hash` would be unchanged. Note that lastcar is not called on an entry unless atom returns `nil`.

10. Transforming a keybag

    Create a hash recording counts for each key (see example 7) but filter on some constraint. A function for `:init-data` takes the key and count and sets the values according to the return convention described below. With a vector for `:init-data`, the count is an index into the vector for the new value. With a hash table, the count is used as key to lookup the new value.

```
(make-hash :init-format :keybag
           :initial-contents #(a b d d e c b a a e c c d a a c c e c c)
           :init-data (lambda (key count) (values key count (<= count 3))))
(make-hash :init-format :keybag
           :initial-contents #(a b d d e c b a a e c c d a a c c e c c)
           :init-data #(zero one two three four)
           :init-default 'more-than-four)
(make-hash :init-format :keybag
           :initial-contents #(a b d d e c b a a e c c d a a c c e c c)
           :init-data (make-hash :initial-contents '(3 "You're out!"))
           :init-default "Whatever!")
```

    The first gives a hash a → 5, b → 2, c → 7, d → 3, e → 3. The second gives a hash a → more-than-four, b → two, c → more-than-four, d → three, e → three. And the third gives a hash with a, b, and c mapping to the string "Whatever!" and d and e mapping to "You're out!".

11. Creating Hash Factories

    Hash factories are shortcuts that encapsulate a specified set of hash creation options, primarily for use with literal hash creation with sequence-style init formats. The factories are functions that package their arguments (&rest style) and use the resulting list as the `:initial-contents` argument to `make-hash` with the given options. The difference between `define-hash-factory` and `make-hash-factory` is that the former defines a toplevel function, whereas the latter returns an anonymous function.

```
(define-hash-factory qhash
```

```
    :init-format :flat
    :test #'eq :size 128
    :documentation "Construct moderate size hash tables for symbols.")

(qhash 'a 1 'b 2 'c 3 'd 4 'x 100 'y -100 'z 0)
(apply #'qhash '(a 1 b 2 c 3 d 4 x 100 y -100 z 0))

(define-hash-factory ahash
    :init-format :pairs
    :init-data (lambda (k v)
                 (if (stringp k) (intern (string-upcase k)) k))
    :documentation "Alist->hash, converting string keys to symbols.")

(ahash ("foo" 10) ("bar" 20) ("zap" 30))
(apply #'ahash '((a . 1) (b . 2) (c . 3) ("d" . 4) ("foo" . "bar")))

(let ((h (make-hash-factory :init-format :keys :init-data *big-hash*)))
  (apply h key1 key2 key3 key4)) ; quick subhash of *big-hash*
```

12. Portable Reader Factories

    It may be desirable to use reader macros to stand-in for particular hash table constructors. These are hash factories that are installed in a readtable using `install-hash-reader` at toplevel. Both dispatched and raw delimited forms are supported, and the installer can accept a list of options or an existing factory.

    Here are three separate uses yielding `:a`→ 1, `:b` → 2, `:c` → 3, `:d` → 4.

```
(install-hash-reader ())  ; default settings and options
#{:a 1 :b 2 :c 3 :d 4}

(install-hash-reader '(:init-format :pairs)
                     :use-dispatch t :open-char #\[ :close-char #\])
#['(:a . 1) '(:b . 2) '(:c . 3) '(:d . 4)]

(install-hash-reader '(:init-format :lists)
                     :use-dispatch nil :open-char #\{ :close-char #\})
{'(:a 1) '(:b 2) '(:c 3) '(:d 4)}
```

This accepts a readtable to modify (current readtable by default) and works well with the :named-readtables package.

4 Creating Hash Tables

The function `make-hash` is an interface to the CL standard function `make-hash-table` that also allows flexible initialization. It accepts all the standard and implementation-dependent keyword arguments that the standard `make-hash-table` does but also accepts a few additional keyword arguments that can specify the initial contents of the table (analogously to the CL standard function `make-array`). The operation of the make-hash initializer is designed to handle all the common cases easily while enabling powerful abstractions where needed. See the Examples section below for examples.

The new keyword arguments are:

- `:initial-contents` *object*

  If the supplied object is non-nil, the object is used to initialize the created hash table in accordance with the `:init-format` argument. For some formats, the `:init-data` argument may also be needed to supply supplementary information for the initializer. The built-in formats support the cases where object is either a hash table or sequence from which the keys and values can be extracted. See the subsection below for a detailed description of the possibilities.

- `:init-format` *keyword*

  A keyword specifying the structure of the initialization contents and auxilliary data given by the `:initial-contents` and `:init-data` arguments. Built-in support is provided for :hash, :flat, :pairs, :lists, :vectors, :seqs, :keys, :keychain, :keybag, and :function. These are described in detail in the subsection below.

  When an initializer format is not supplied, it is computed by calling the generic function `hash-initializer-default-format` on the given `:initial-contents` object. A methods for this function should be defined whenever the function `initialize-hash` is extended to handle a new class of `:initial-contents` objects. Methods can be overridden to change the default used in existing cases.

- `:init-data` *object*

  Auxilliary data used for initialization with some formats. Its structure and meaning depends on the value of `:init-format`; as described in the subsection below.

- `:init-default` *value*

Default value to use in indirect initialization when the value for the given key cannot be determined from the `:initial-contents` and `:init-data` for the particular `:init-format` supplied.

If no :initial-contents argument is supplied, the hash table is not initialized, and `make-hash` behaves exactly like the standard function `make-hash-table`. For many formats, initialization only requires an :initial-contents argument. See Examples for more.

4.1 Functions as `:init-data` (or `:initial-contents`)

For most of the pre-defined formats, a function can be passed as the `:init-data`, and with the `:function` format, a can be passed as the `:initial-contents` as well. These functions are expected to return three values *KEY VALUE [BAD-VALUE]* that are used (under some conditions) to create a new key-value entry in the hash table being initialized. Here, BAD-VALUE is a **ternary** value: nil (or missing) means to use KEY and VALUE as is; t means to skip creating this entry entirely, and any other non-nil value means to associate KEY to the specified `:init-default` value *instead* of VALUE.

In the description of the predefined formats below, such function arguments are used in one of three ways:

1. Entry transformation: *INIT-KEY INIT-VALUE -> KEY VALUE [BAD-VALUE]*

   The key and value specified by `:initial-contents` (*INIT-KEY INIT-VALUE*) are passed to the function and the return values used as described above. (Formats `:hash`, `:flat`, `:pairs`, `:lists`, `:vectors`, `:seqs`.)

2. Key transformation: *INIT-KEY -> KEY VALUE [BAD-VALUE]*

   With format `:keys`, the key specified by `:initial-contents` is passed to the function and the return values used as described above.

3. Entry generation: *rest ARGS -> KEY VALUE [BAD-VALUE]*

   With format `:function`, the `:initial-contents` argument is a function. This function is applied repeatedly to *ARGS* and the return values used as described above. However, in this case, the first time that KEY is nil, initialization stops.

See also the documentation for the function `make-hash-transformer` which creates a function suitable for use in this way from a simpler function on keys or entries.

4.2 Predefined Initialization Formats

The `:init-format` argument is a keyword that determines how the keyword arguments `:initial-contents` and `:init-data` are interpreted. If `:init-format` is not supplied, the default format is determined by the type of `:initial-contents`.

There are four basic cases in the pre-defined initialization support:

1. Initializing from an existing hash table

   When `:init-format` is `:hash` or by default if `:initial-contents` is a hash-table, the new hash table is initialized by a shallow copy of the initial contents table, with shared structure in keys and values. If `:init-data` is a function, that function is used for entry transformation of the hash table given in `:initial-contents`.

2. Initializing from a sequence (or sequences) specifying key-value pairs.

   When `:init-format` is `:flat`, `:pairs`, `:lists`, `:vectors`, or `:seqs`, the `:initial-contents` should be a sequence that specifies a collection of key-value pairs. The only difference among these formats is the expected structure of the sequence's elements. For `:flat`, the keys and values alternate; for `:pairs`, it is a sequence of cons pairs (e.g., an alist); for `:lists`, `:vectors`, and `:seqs`, it is a sequence of lists, vectors, or arbitrary sequences respectively of which the first two elements of each give the corresponding key and value. In these cases, if `:init-data` is nil or missing, the key-value pairs are used as is; if `:init-data` is a function, the function is used for entry transformation, as described above, for each pair.

   When `:init-format` is `:keychain`, the `:initial-contents` should be a sequence of keys and `:init-data` should be a sequence of corresponding values *in the same order*. The table is initialized with the resultant key-value pairs.

   When `:init-format` is `:keys`, the `:initial-contents` should be a sequence of keys. The corresponding value is obtained by looking up the key in the hash table, alist, or function (via key mapping, see above) that is passed as `:init-data`, which in this case is required.

3. Initializing from a bag/multiset of keys.

   When `:init-format` is `:keybag`, the `:initial-contents` should be a sequence representing a *multiset* (a collection with possibly repeated elements) of keys. The hash table is initialized to map the unique elements from that multiset (as keys) to the number of times that element appears in the multiset (as values).

   In this case, if `:init-data` is a vector, hash table, or function, the count is used to find the corresponding value by indexing into the vector, looking up the value associated with count in the data hash-table, or calling the function with the key and count. When a value cannot be found, the default is used instead, subject to the value of BAD-VALUE in the function case.

4. Initializing from a function.

   When `:init-format` is `:function` or `:initial-contents` is a function, the hash table is initialized by using the function for entry generation as described above.

See also the documentation for `make-hash` for a relatively succinct table describing these options. Keep in mind that the interpretation of the formats is specified by methods of the `initialize-hash` generic function, and the default formats for different `:initial-contents` types by methods of the `hash-initializer-default-format`.

5 Defining Custom Initialization Formats

Initialization by `make-hash` is controlled by the generic function `initialize-hash`. Defining new methods for this function, or overriding existing methods, makes it easy to extend the hash table initialization, to add or modify formats, change behaviors, and so forth.

The function `initialize-hash` takes five arguments: the hash table being initialized, the format specifier, the initial contents source object, the auxilliary data (`:init-data`) object, and the default value (`:init-default`). The format is usually a keyword with eql specialization. The contents source and data object are specialized on type.

6 Specifying Default Formats

When no `:init-format` argument is given to `make-hash`, the default format is determined by calling a suitable method of the generic function `hash-initializer-default-format`, passing the `:initial-contents` argument. The predefined methods use format `:hash` given a hash table, `:flat` given a sequence, and `:function` given a function. More flexibility may be desired in particular applications.

7 Hash Table Factories

When specific patterns of hash table construction options are used repeatedly, it can be helpful to encapsulate those patterns in a simple way. Hash table factories are shortcut functions that create a hash table using prespecified construction options. Any of the keyword arguments to `make-hash`, except for `:initial-contents`, can be passed to the factory constructor and will be used for creating the hash table when the factory is called. The arguments in the factory call are packaged `&rest`-style in a list and used as the `:initial-contents`. There are two factory constructors: `define-hash-factory` creates a toplevel function of a given name and `make-hash-factory` creates an anonymous function.

8 Reader Representations

Similarly, it might be desirable for the hash factories to be represented by syntax at read time via reader macros. The macro `install-hash-reader` updates a given readtable (the current readtable by default) so that a dispatched or raw delimited form creates a hash table. The effect is identical to the use of the hash table factories, except syntactically. Indeed, a factory can be passed directly to the `install-hash-reader`.

Calls to this macro must occur at toplevel to have effect. It is designed to be as portable as possible and to work well with the named-readtables package. Common examples would be the use of #{ or { to represent hash tables.

9 Dictionary

9.1 make-hash [Function]

make-hash *&key initial-contents init-format init-data init-default ... → hash-table*

Creates, initializes if requested, and returns a new hash table.

Keyword options include all those of the standard `make-hash-table`, any extension options allowed by the given implementation, and the additional keyword options to control initialization: `:initial-contents`, the main source for information filling the table; `:init-format`, a keyword specifying how the initialization options are interpreted; `:init-data`, auxilliary data needed for initialization in some formats; and `:init-default`, a default value used when the value for a key cannot be initialized. See the description above in Creating Hash Tables. Users can support other types/configurations (or alter the default handling) by extending the generic function `initialize-hash` in this package; see Defining Custom Initialization Formats.

9.2 make-hash-transformer [Function]

make-hash-transformer *domain function &optional badp → function*

Transform FUNCTION to be suitable for use as the `:init-data` (or `:initial-contents`) argument to `make-hash`. DOMAIN specifies the signature of FUNCTION and is one of the keywords `:key`, `:value`, or `:entry`, indicating that FUNCTION takes a key, a value, or a key and a value, repectively. BADP is a function with the same argument signature as FUNCTION that follows the return convention described above. Specifically, it returns a ternary value: nil means that the transformed entry should be used as is, t means that the entry should be skipped, and any other non-nil value means that the key should be used with a default. Note that FUNCTION is *not* called for an entry if BADP returns a non-nil value.

The returned function accepts a key and a value (the value is optional with DOMAIN :key) and returns three values: the key, the value, and the bad-value ternary for that entry.

9.3 initialize-hash [Generic Function]

initialize-hash *table form source data default*

Creates and adds an entry to TABLE using info of format FORM in SOURCE and DATA. SOURCE contains the main contents, and DATA (op-

tionally) contains auxilliary information or objects required for initialization for some formats. DEFAULT is the value that should be stored in the table when an appropriate value associated to a key cannot be found. Adding or redefining methods for this function allows extension or modification of the initialization mechanism.

Note the convention, used by the predefined methods, that functions passed as either SOURCE or DATA are expected to return three values, using the convention described above.

9.4 hash-initializer-default-format [Generic Function]

hash-initializer-default-format *source → keyword or error*

Selects an initializer format based on the given initial contents SOURCE. For example, the default format for sequence contents is `:flat`; to change it to `:pairs` so that an alist is expected as `:initial-contents` by default, do the following:

```
(defmethod hash-initializer-default-format ((source list))
  :pairs)
```

9.5 `*hash-factory-defaults*` [Special Variable]

Hash table creation options used as defaults by hash factory constructors. These option specifications are passed last to make-hash by the hash factories and so are overridden by options passed as explicit arguments to the factory constructor.

Changing this variable affects the options used by every hash factory that does not fully specify its options. This includes default calls to the reader constructors. Of particular note are the `:test` and `:init-format` options.

9.6 define-hash-factory [Macro]

define-hash-factory *name &key . . . hash-options. . .*

Create a hash-table factory NAME that calls `make-hash` with options specified by given by the hash-options arguments. The defined function packages its arguments as a list, which it passes as the `:initial-contents` argument to `make-hash`.

The hash-options are alternating keyword-value pairs. The supplied keyword arguments precede and thus override the options in `*hash-factory-defaults*`, which is intended to allow one to use short names or customized policies in

simple calling patterns. Complex initialization patterns may need the full power of 'make-hash' itself.

9.7 make-hash-factory [Function]

make-hash-factory *key ... hash-options... → factory-function*

Like define-hash-factory but creates and returns an anonymous factory function.

9.8 install-hash-reader [Macro]

install-hash-reader *options &key readtable use-dispatch allow-numbered-dispatch open-char close-char dispatch-char*

Creates a hash table factory specified by OPTIONS and installs it in READTABLE (the current readtable by default). To have effect, this must be called at toplevel.

OPTIONS is either a list of keyword-value pairs (as would be passed to `make-hash` or `make-hash-factory`) or a hash factory function. READTABLE is a readtable object, `*readtable*` by default.

The keyword arguments control how the reader is modified as follows:

- USE-DISPATCH (t by default) determines whether the reader macro uses a dispatch character DISPATCH-CHAR before OPEN-CHAR. If non-nil, a dispatch character is used and is registered in READTABLE. If this is nil, then OPEN-CHAR and CLOSE-CHAR will be a raw delimited construct.

- ALLOW-NUMBERED-DISPATCH (nil by default) allows a dispatched reader macro to modify its hash test when given numeric arguments between DISPATCH-CHAR and OPEN-CHAR. This only applies when USE-DISPATCH is non-nil and when OPTIONS is a list, not a factory function. The goal here is to make it easy to reuse reader factories in several contexts.

  If nil, numbered dispatch is not supported. If t, numeric arguments 0, 1, 2, and 3 correspond to hash tests `eq`, `eql`, `equal`, and `equalp` respectively. If a sequence of symbols or functions, those functions are used for the hash test given a numeric argument from 0 below the length of the sequence. In either case, dispatch *without* a numeric argument uses the originally specified options.

  Note: This is *an experimental feature and may be discontinued in future versions* if it proves more confusing than helpful.

- OPEN-CHAR (default open-brace) is the character that delimits the beginning of the hash-table contents. If USE-DISPATCH is non-nil, this character must be preceeded by DISPATCH-CHAR, and optionally a numeric argument.
- CLOSE-CHAR (default close-brace) is the character that delimits the end of the hash-table contents.
- DISPATCH-CHAR (default #) is the character used to indicate a dispatched reader macro. When (and only when) USE-DISPATCH is non-nil. READTABLE is modified to register this as as a dispatch and a non-terminating macro character via `make-dispatch-macro-character`. Note that there can be more than one dispatch character in a read table.