# Alexandria Manual

draft version

# Table of Contents

## 0.1 Hash Table Utilities

`alexandria:copy-hash-table` *table* **&key** *test size rehash-size rehash-threshold* [Function]

Returns a shallow copy of hash table `table`, with the same keys and values as the `table`. The copy has the same properties as the original, unless overridden by the keyword arguments.

`alexandria:maphash-keys` *function table* [Function]

Like `maphash`, but calls `function` with each key in the hash table `table`.

`alexandria:maphash-values` *function table* [Function]

Like `maphash`, but calls `function` with each value in the hash table `table`.

`alexandria:hash-table-keys` *table* [Function]

Returns a list containing the keys of hash table `table`.

`alexandria:hash-table-values` *table* [Function]

Returns a list containing the values of hash table `table`.

`alexandria:hash-table-alist` *table* [Function]

Returns an association list containing the keys and values of hash table `table`.

`alexandria:hash-table-plist` *table* [Function]

Returns a property list containing the keys and values of hash table `table`.

`alexandria:alist-hash-table` *alist* **&rest** *hash-table-initargs* [Function]

Returns a hash table containing the keys and values of the association list `alist`. Hash table is initialized using the `hash-table-initargs`.

`alexandria:plist-hash-table` *plist* **&rest** *hash-table-initargs* [Function]

Returns a hash table containing the keys and values of the property list `plist`. Hash table is initialized using the `hash-table-initargs`.

## 0.2 Higher Order Functions

`alexandria:disjoin` *predicate* **&rest** *more-predicates* [Function]

Returns a function that applies each of `predicate` and `more-predicate` functions in turn to its arguments, returning the primary value of the first predicate that returns true, without calling the remaining predicates. If none of the predicates returns true, `nil` is returned.

`alexandria:conjoin` *predicate* **&rest** *more-predicates* [Function]

Returns a function that applies each of `predicate` and `more-predicate` functions in turn to its arguments, returning `nil` if any of the predicates returns false, without calling the remaining predicated. If none of the predicates returns false, returns the primary value of the last predicate.

`alexandria:compose` *function* **&rest** *more-functions* [Function]
Returns a function composed of `function` and `more-functions` that applies its arguments to to each in turn, starting from the rightmost of `more-functions`, and then calling the next one with the primary value of the last.

`alexandria:multiple-value-compose` *function* **&rest** *more-functions* [Function]
Returns a function composed of `function` and `more-functions` that applies its arguments to to each in turn, starting from the rightmost of `more-functions`, and then calling the next one with all the return values of the last.

`alexandria:curry` *function* **&rest** *arguments* [Function]
Returns a function that applies `arguments` and the arguments it is called with to `function`.

`alexandria:rcurry` *function* **&rest** *arguments* [Function]
Returns a function that applies the arguments it is called with and `arguments` to `function`.

## 0.3 List Manipulation

`alexandria:proper-list` [Type]
Type designator for proper lists. Implemented as a `satisfies` type, hence not recommended for performance intensive use. Main usefullness as a type designator of the expexted type in a `type-error`.

`alexandria:circular-list` [Type]
Type designator for circular lists. Implemented as a `satisfies` type, so not recommended for performance intensive use. Main usefullness as the expected-type designator of a `type-error`.

`alexandria:appendf` *g1* **&rest** *lists* **&environment** *g0* [Macro]
Modify-macro for `append`. Appends `lists` to the place designated by the first argument.

`alexandria:circular-list` **&rest** *elements* [Function]
Creates a circular list of `elements`.

`alexandria:circular-list-p` *object* [Function]
Returns true if `object` is a circular list, `nil` otherwise.

`alexandria:circular-tree-p` *object* [Function]
Returns true if `object` is a circular tree, `nil` otherwise.

`alexandria:proper-list-p` *object* [Function]
Returns true if `object` is a proper list.

`alexandria:lastcar` *list* [Function]
Returns the last element of `list`. Signals a type-error if `list` is not a proper list.

`alexandria:make-circular-list` *length* **&key** *initial-element* [Function]
Creates a circular list of `length` with the given `initial-element`.

`alexandria:ensure-list` *list* [Function]
If `list` is a list, it is returned. Otherwise returns the list designated by `list`.

`alexandria:sans` *plist* **&rest** *keys* [Function]
Returns a propery-list with same keys and values as `plist`, except that keys in the list designated by `keys` and values corresponding to them are removed. The returned property-list may share structure with the `plist`, but `plist` is not destructively modified.

`alexandria:mappend` *function* **&rest** *lists* [Function]
Applies `function` to respective element(s) of each `list`, appending all the all the result list to a single list. `function` must return a list.

`alexandria:map-product` *function list* **&rest** *more-lists* [Function]
Returns a list containing the results of calling `function` with one argument from `list`, and one from each of `more-lists` for each combination of arguments. In other words, returns the product of `list` and `more-lists` using `function`.

Example:

```
(map-product 'list '(1 2) '(3 4) '(5 6)) => ((1 3 5) (1 3 6) (1 4 5) (1 4 6)
                                             (2 3 5) (2 3 6) (2 4 5) (2 4 6))
```

`alexandria:set-equal` *list1 list2* **&key** *test key* [Function]
Returns true if every element of LIST1 matches some element of LIST2 and every element of LIST2 matches some element of LIST1. Otherwise returns false.

`alexandria:setp` *object* **&key** *test key* [Function]
Returns true if `object` is a list that denotes a set, `nil` otherwise. A list denotes a set if each element of the list is unique under `key` and `test`.

`alexandria:flatten` *tree* [Function]
Traverses the tree in order, collecting non-null leaves into a list.

## 0.4 Sequence Manipulation

`alexandria:proper-sequence` [Type]
Type designator for proper sequences, that is proper lists and sequences that are not lists.

`alexandria:deletef` *g134 item* **&rest** *remove-keywords* **&environment** *g133* [Macro]

Modify-macro for `delete`. Sets place designated by the first argument to the result of calling `delete` with `item`, place, and the `remove-keywords`.

`alexandria:removef` *g114 item* **&rest** *remove-keywords* **&environment** *g113* [Macro]

Modify-macro for `remove`. Sets place designated by the first argument to the result of calling `remove` with `item`, place, and the `remove-keywords`.

`alexandria:rotate` *sequence* **&optional** *n* [Function]

Returns a sequence of the same type as `sequence`, with the elements of `sequence` rotated by `n`: `n` elements are moved from the end of the sequence to the front if `n` is positive, and `-n` elements moved from the front to the end if `n` is negative. `sequence` must be a proper sequence. `n` must be an integer, defaulting to 1. If absolute value of `n` is greater then the length of the sequence, the results are identical to calling `rotate` with (* (SIGNUM N) (MOD `n` (LENGTH SEQUENCE))). The original sequence may be destructively altered, and result sequence may share structure with it.

`alexandria:suffle` *sequence* **&key** *start end* [Function]

Returns a radom permutation of `sequence` bounded by `start` and `end`. Permuted sequence may share storage with the original one. Signals an error if `sequence` is not a proper sequence.

`alexandria:random-elt` *sequence* **&key** *start end* [Function]

Returns a random element from `sequence` bounded by `start` and `end`. Signals an error if the `sequence` is not a proper sequence.

`alexandria:emptyp` *sequence* [Function]

Returns true if `sequence` is an empty sequence. Signals an error if `sequence` is not a sequence

`alexandria:sequence-of-length-p` *sequence length* [Function]

Return true if `sequence` is a sequence of length `length`. Signals an error if `sequence` is not a sequence. Returns `false` for circular lists.

`alexandria:copy-sequence` *type sequence* [Function]

Returns a fresh sequence of `type`, which has the same elements as `sequence`.

`alexandria:first-elt` *sequence* [Function]

Returns the first element of `sequence`. Signals a type-error if `sequence` is not a sequence, or is an empty sequence.

`alexandria:last-elt` *sequence* [Function]

Returns the last element of `sequence`. Signals a type-error if `sequence` is not a proper sequence, or is an empty sequence.

`alexandria:starts-with` *object sequence* [Function]
Returns true if `sequence` is a sequence whose first element is `eql` to `object`. Returns `nil` if the `sequence` is not a sequence or is an empty sequence.

`alexandria:ends-with` *object sequence* [Function]
Returns true if `sequence` is a sequence whose last element is `eql` to `object`. Returns `nil` if the `sequence` is not a sequence or is an empty sequence. Signals an error if `sequence` is an improper list.

## 0.5 Macro Writing Utilities

`alexandria:with-unique-names` *names* **&body** *forms* [Macro]
Binds each variable named by `names` to a unique symbol.

`alexandria:once-only` *names* **&body** *forms* [Macro]
Evaluates `forms` with `names` rebound to temporary variables, ensuring that each is evaluated only once.

Example: (defmacro cons1 (x) (once-only (x) '(cons ,x ,x))) (let ((y 0)) (cons1 (incf y))) => (1 . 1)

## 0.6 Symbol Utilities

`alexandria:ensure-symbol` *name* **&optional** *package* [Function]
Returns a symbol with name designated by `name`, accessible in package designated by `package`. If symbol is not already accessible in `package`, it is interned there.

Example: (ENSURE-SYMBOL `:cons` :CL) => `cl:cons`

`alexandria:format-symbol` *package control* **&rest** *arguments* [Function]
Constructs a string by applying `arguments` to `control` as if by `format`, and then creates a symbol named by that string. If `package` is `nil`, returns an uninterned symbol, if package is `t`, returns a symbol interned in the current package, and otherwise returns a symbol interned in the package designated by `package`.

`alexandria:make-keyword` *name* [Function]
Interns the string designated by `name` in the `keyword` package.

`alexandria:make-gensym-list` *length* **&optional** *x* [Function]
Returns a list of `length` gensyms, each generated with a call to `gensym` using (if provided) as the argument.

## 0.7 Array Utilities

`alexandria:array-index` [Type]
Type designator for an array of `length`: an integer between 0 (inclusive) and `length` (exclusive). `length` defaults to `array-dimension-limit`.

`alexandria:copy-array` *array* **&key** *element-type fill-pointer adjustable* [Function]
Returns an undisplaced copy of `array`, with same fill-pointer and adjustability (if any) as the original, unless overridden by the keyword arguments.

## 0.8 Type Designator Manipulation

`alexandria:of-type` *type* [Function]
Returns a function of one argument, which returns true when its argument is of `type`.

`alexandria:type=` *type1 type2* [Function]
Returns a primary value of `t` is TYPE1 and TYPE2 are the same type, and a secondary value that is true is the type equality could be reliably determined: primary value of `nil` and secondary value of `t` indicates that the types are not equivalent.

## 0.9 Mathematical Utilities

`alexandria:maxf` *g172* **&rest** *numbers* **&environment** *g171* [Macro]
Modify-macro for `max`. Sets place designated by the first argument to the maximum of its original value and `numbers`.

`alexandria:minf` *g192* **&rest** *numbers* **&environment** *g191* [Macro]
Modify-macro for `min`. Sets place designated by the first argument to the minimum of its original value and `numbers`.

`alexandria:clamp` *number min max* [Function]
Clamps the `number` into [MIN, MAX] range. Returns `min` if `number` lesser then `min` and `max` if `number` is greater then `max`, otherwise returns `number`.

`alexandria:lerp` *v a b* [Function]
Returns the result of linear interpolation between A and `b`, using the interpolation coefficient `v`.

`alexandria:gaussian-random` **&optional** *min max* [Function]
Returns two gaussian random double floats as the primary and secondary value, optionally constrained by `min` and `max`. Gaussian random numbers form a standard normal distribution around 0.0d0.

`alexandria:iota` *n* **&key** *start step* [Function]
Return a list of n numbers, starting from `start` (with numeric contagion from `step` applied), each consequtive number being the sum of the previous one and `step`. `start` defaults to 0 and `step` to 0.

Examples:

```
(iota 4)                       => (0 1 2 3 4)
(iota 3 :start 1 :step 1.0)    => (1.0 2.0 3.0)
(iota 3 :start -1 :step -1/2) => (-1 -3/2 -2)
```

`alexandria:mean` *sample* [Function]

Returns the mean of `sample`. `sample` must be a sequence of numbers.

`alexandria:median` *sample* [Function]

Returns median of `sample`. `sample` must be a sequence of real numbers.

`alexandria:variance` *sample* **&key** *biased* [Function]

Variance of `sample`. Returns the biased variance if `biased` is true (the default), and the unbiased estimator of variance if `biased` is false. `sample` must be a sequence of numbers.

`alexandria:standard-deviation` *sample* **&key** *biased* [Function]

Standard deviation of `sample`. Returns the biased standard deviation if `biased` is true (the default), and the square root of the unbiased estimator for variance if `biased` is false (which is not the same as the unbiased estimator for standard deviation). `sample` must be a sequence of numbers.