alexandria Manual

draft version

Alexandria software and associated documentation are in the public domain:

> Authors dedicate this work to public domain, for the benefit of the public at large and to the detriment of the authors' heirs and successors. Authors intends this dedication to be an overt act of relinquishment in perpetuity of all present and future rights under copyright law, whether vested or contingent, in the work. Authors understands that such relinquishment of all rights includes the relinquishment of all rights to enforce (by lawsuit or otherwise) those copyrights in the work.
>
> Authors recognize that, once placed in the public domain, the work may be freely reproduced, distributed, transmitted, used, modified, built upon, or otherwise exploited by anyone for any purpose, commercial or non-commercial, and in any way, including by methods that have not yet been invented or conceived.

In those legislations where public domain dedications are not recognized or possible, Alexandria is distributed under the following terms and conditions:

> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Unless otherwise noted, the symbols are exported from the `"ALEXANDRIA"` package; only newer symbols that require `"ALEXANDRIA-2"` are fully qualified.

The package `"ALEXANDRIA-2"` includes all the symbols from `"ALEXANDRIA-1"`.

Table of Contents

1 Hash Tables

`ensure-gethash` *key hash-table &optional default* [Macro]
Like `gethash`, but if `key` is not found in the `hash-table` saves the `default` under key before returning it. Secondary return value is true if key was already in the table.

`copy-hash-table` *table &key key test size rehash-size rehash-threshold* [Function]
Returns a 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.

Before each of the original values is set into the new hash-table, `key` is invoked on the value. As `key` defaults to `cl:identity`, a shallow copy is returned by default.

`maphash-keys` *function table* [Function]
Like `maphash`, but calls `function` with each key in the hash table `table`.

`maphash-values` *function table* [Function]
Like `maphash`, but calls `function` with each value in the hash table `table`.

`hash-table-keys` *table* [Function]
Returns a list containing the keys of hash table `table`.

`hash-table-values` *table* [Function]
Returns a list containing the values of hash table `table`.

`hash-table-alist` *table* [Function]
Returns an association list containing the keys and values of hash table `table`.

`hash-table-plist` *table* [Function]
Returns a property list containing the keys and values of hash table `table`.

`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`.

`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`.

2 Data and Control Flow

`define-constant` *name initial-value &key test documentation* [Macro]

Ensures that the global variable named by `name` is a constant with a value that is equal under `test` to the result of evaluating `initial-value`. `test` is a /function designator/ that defaults to `eql`. If `documentation` is given, it becomes the documentation string of the constant.

Signals an error if `name` is already a bound non-constant variable.

Signals an error if `name` is already a constant variable whose value is not equal under `test` to result of evaluating `initial-value`.

`destructuring-case` *keyform &body clauses* [Macro]

`destructuring-case`, `-ccase`, and `-ecase` are a combination of `case` and `destructuring-bind`. `keyform` must evaluate to a `cons`.

Clauses are of the form:

```
((CASE-KEYS . DESTRUCTURING-LAMBDA-LIST) FORM*)
```

The clause whose `case-keys` matches `car` of `key`, as if by `case`, `ccase`, or `ecase`, is selected, and FORMs are then executed with `cdr` of `key` is destructured and bound by the `destructuring-lambda-list`.

Example:

```
(defun dcase (x)
  (destructuring-case x
    ((:foo a b)
     (format nil "foo: ~S, ~S" a b))
    ((:bar &key a b)
     (format nil "bar: ~S, ~S" a b))
    (((:alt1 :alt2) a)
     (format nil "alt: ~S" a))
    ((t &rest rest)
     (format nil "unknown: ~S" rest))))

(dcase (list :foo 1 2))        ; => "foo: 1, 2"
(dcase (list :bar :a 1 :b 2))  ; => "bar: 1, 2"
(dcase (list :alt1 1))         ; => "alt: 1"
(dcase (list :alt2 2))         ; => "alt: 2"
(dcase (list :quux 1 2 3))     ; => "unknown: 1, 2, 3"

(defun decase (x)
  (destructuring-case x
    ((:foo a b)
     (format nil "foo: ~S, ~S" a b))
    ((:bar &key a b)
     (format nil "bar: ~S, ~S" a b))
    (((:alt1 :alt2) a)
     (format nil "alt: ~S" a))))

(decase (list :foo 1 2))        ; => "foo: 1, 2"
```

```
(decase (list :bar :a 1 :b 2))  ; => "bar: 1, 2"
(decase (list :alt1 1))         ; => "alt: 1"
(decase (list :alt2 2))         ; => "alt: 2"
(decase (list :quux 1 2 3))     ; =| error
```

`ensure-functionf` *&rest places* [Macro]

Multiple-place modify macro for `ensure-function`: ensures that each of `places` contains a function.

`multiple-value-prog2` *first-form second-form &body forms* [Macro]

Evaluates `first-form`, then `second-form`, and then `forms`. Yields as its value all the value returned by `second-form`.

`named-lambda` *name lambda-list &body body* [Macro]

Expands into a lambda-expression within whose `body` `name` denotes the corresponding function.

`nth-value-or` *nth-value &body forms* [Macro]

Evaluates `form` arguments one at a time, until the `nth-value` returned by one of the forms is true. It then returns all the values returned by evaluating that form. If none of the forms return a true nth value, this form returns `nil`.

`if-let` *bindings &body* (*then-form* **&optional** *else-form*) [Macro]

Creates new variable bindings, and conditionally executes either `then-form` or `else-form`. `else-form` defaults to `nil`.

`bindings` must be either single binding of the form:

```
(variable initial-form)
```

or a list of bindings of the form:

```
((variable-1 initial-form-1)
 (variable-2 initial-form-2)
 ...
 (variable-n initial-form-n))
```

All initial-forms are executed sequentially in the specified order. Then all the variables are bound to the corresponding values.

If all variables were bound to true values, the `then-form` is executed with the bindings in effect, otherwise the `else-form` is executed with the bindings in effect.

`when-let` *bindings &body forms* [Macro]

Creates new variable bindings, and conditionally executes `forms`.

`bindings` must be either single binding of the form:

```
(variable initial-form)
```

or a list of bindings of the form:

```
((variable-1 initial-form-1)
 (variable-2 initial-form-2)
 ...
 (variable-n initial-form-n))
```

All initial-forms are executed sequentially in the specified order. Then all the variables are bound to the corresponding values.

If all variables were bound to true values, then `forms` are executed as an implicit `progn`.

`when-let*` *bindings &body body* [Macro]

Creates new variable bindings, and conditionally executes `body`.

`bindings` must be either single binding of the form:

```
(variable initial-form)
```

or a list of bindings of the form:

```
((variable-1 initial-form-1)
 (variable-2 initial-form-2)
 ...
 (variable-n initial-form-n))
```

Each `initial-form` is executed in turn, and the variable bound to the corresponding value. `initial-form` expressions can refer to variables previously bound by the `when-let*`.

Execution of `when-let*` stops immediately if any `initial-form` evaluates to `nil`. If all INITIAL-FORMs evaluate to true, then `body` is executed as an implicit `progn`.

`switch` (*object* **&key** *test key*) *&body clauses* [Macro]

Evaluates first matching clause, returning its values, or evaluates and returns the values of `t` or `otherwise` if no keys match.

`cswitch` (*object* **&key** *test key*) *&body clauses* [Macro]

Like `switch`, but signals a continuable error if no key matches.

`eswitch` (*object* **&key** *test key*) *&body clauses* [Macro]

Like `switch`, but signals an error if no key matches.

`whichever` *&rest possibilities* [Macro]

Evaluates exactly one of `possibilities`, chosen at random.

`xor` *&rest datums* [Macro]

Evaluates its arguments one at a time, from left to right. If more than one argument evaluates to a true value no further `datums` are evaluated, and `nil` is returned as both primary and secondary value. If exactly one argument evaluates to true, its value is returned as the primary value after all the arguments have been evaluated, and `t` is returned as the secondary value. If no arguments evaluate to true `nil` is returned as primary, and `t` as secondary value.

`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.

`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 predicates. If none of the predicates returns false, returns the primary value of the last predicate.

`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.

`ensure-function` *function-designator* [Function]
Returns the function designated by `function-designator:` if `function-designator` is a function, it is returned, otherwise it must be a function name and its `fdefinition` is returned.

`multiple-value-compose` *function &rest more-functions* [Function]
Returns a function composed of `function` and `more-functions` that applies its arguments 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.

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

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

`alexandria-2:line-up-first` *&rest forms* [Macro]
Lines up `forms` elements as the first argument of their successor. Example:

```
(thread-first
  5
  (+ 20)
  /
  (+ 40))
```

is equivalent to:

```
(+ (/ (+ 5 20)) 40)
```

Note how the single '/ got converted into a list before threading.

`alexandria-2:line-up-last` *&rest forms* [Macro]
Lines up `forms` elements as the last argument of their successor. Example:

```
(thread-last
  5
  (+ 20)
  /
  (+ 40))
```

is equivalent to:

```
(+ 40 (/ (+ 20 5)))
```

Note how the single '/ got converted into a list before threading.

3 Conses

`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 expected type in a `type-error`.

`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`.

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

`nconcf` *place &rest lists* [Macro]
: Modify-macro for `nconc`. Concatenates `lists` to place designated by the first argument.

`remove-from-plistf` *place &rest keys* [Macro]
: Modify macro for `remove-from-plist`.

`delete-from-plistf` *place &rest keys* [Macro]
: Modify macro for `delete-from-plist`.

`reversef` *place* [Macro]
: Modify-macro for `reverse`. Copies and reverses the list stored in the given place and saves back the result into the place.

`nreversef` *place* [Macro]
: Modify-macro for `nreverse`. Reverses the list stored in the given place by destructively modifying it and saves back the result into the place.

`unionf` *place list &rest args* [Macro]
: Modify-macro for `union`. Saves the union of `list` and the contents of the place designated by the first argument to the designated place.

`nunionf` *place list &rest args* [Macro]
: Modify-macro for `nunion`. Saves the union of `list` and the contents of the place designated by the first argument to the designated place. May modify either argument.

`doplist` (*key val plist* **&optional** *values*) *&body body* [Macro]
: Iterates over elements of `plist`. `body` can be preceded by declarations, and is like a `tagbody`. `return` may be used to terminate the iteration early. If `return` is not used, returns `values`.

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

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

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

`alist-plist` *alist* [Function]
Returns a property list containing the same keys and values as the association list `alist` in the same order.

`plist-alist` *plist* [Function]
Returns an association list containing the same keys and values as the property list `plist` in the same order.

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

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

`ensure-car` *thing* [Function]
If `thing` is a `cons`, its `car` is returned. Otherwise `thing` is returned.

`ensure-cons` *cons* [Function]
If `cons` is a cons, it is returned. Otherwise returns a fresh cons with `cons` in the car, and `nil` in the cdr.

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

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

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

`(setf lastcar)` [Function]
Sets the last element of `list`. Signals a type-error if `list` is not a proper list.

`proper-list-length` *list* [Function]
Returns length of `list`, signalling an error if it is not a proper list.

`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.

`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))
```

`remove-from-plist` *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. Keys are compared using `eq`.

`delete-from-plist` *plist &rest keys* [Function]
Just like `remove-from-plist`, but this version may destructively modify the provided `plist`.

`alexandria-2:delete-from-plist*` *plist &rest keys* [Function]
Just like `remove-from-plist`, but this version may destructively modify the provided `plist`. The second return value is an alist of the removed items, in unspecified order.

`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.

`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`.

4 Sequences

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

`deletef` *place item &rest keyword-arguments* [Macro]
Modify-macro for `delete`. Sets place designated by the first argument to the result of calling `delete` with `item`, place, and the `keyword-arguments`.

`removef` *place item &rest keyword-arguments* [Macro]
Modify-macro for `remove`. Sets place designated by the first argument to the result of calling `remove` with `item`, place, and the `keyword-arguments`.

`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))).
```

Note: the original sequence may be destructively altered, and result sequence may share structure with it.

`shuffle` *sequence &key start end* [Function]
Returns a random permutation of `sequence` bounded by `start` and `end`. Original sequence may be destructively modified, and (if it contains `cons` or lists themselv) share storage with the original one. Signals an error if `sequence` is not a proper sequence.

`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 non-empty sequence, or if `end` and `start` are not proper bounding index designators for `sequence`.

`emptyp` *sequence* [Generic Function]
Returns `t` if `sequence` is an empty sequence and `nil` otherwise. Signals an error if `sequence` is not a sequence.

`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.

`length=` *&rest sequences* [Function]
Takes any number of sequences or integers in any order. Returns true iff the length of all the sequences and the integers are equal. Hint: there's a compiler macro that expands into more efficient code if the first argument is a literal integer.

`copy-sequence` *type sequence* [Function]
Returns a fresh sequence of `type`, which has the same elements as `sequence`.

`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.

`(setf first-elt)` [Function]
Sets the first element of `sequence`. Signals a type-error if `sequence` is not a sequence, is an empty sequence, or if `object` cannot be stored in `sequence`.

`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.

`(setf last-elt)` [Function]
Sets the last element of `sequence`. Signals a type-error if `sequence` is not a proper sequence, is an empty sequence, or if `object` cannot be stored in `sequence`.

`starts-with` *object sequence &key test key* [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.

`starts-with-subseq` *prefix sequence &rest args &key return-suffix &allow-other-keys* [Function]
Test whether the first elements of `sequence` are the same (as per TEST) as the elements of `prefix`.

If `return-suffix` is `t` the function returns, as a second value, a sub-sequence or displaced array pointing to the sequence after `prefix`.

`ends-with` *object sequence &key test key* [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.

`ends-with-subseq` *suffix sequence &key test* [Function]
Test whether `sequence` ends with `suffix`. In other words: return true if the last (length SUFFIX) elements of `sequence` are equal to `suffix`.

`map-combinations` *function sequence &key start end length copy* [Function]
Calls `function` with each combination of `length` constructable from the elements of the subsequence of `sequence` delimited by `start` and `end`. `start` defaults to `0`, `end` to length of `sequence`, and `length` to the length of the delimited subsequence. (So unless `length` is specified there is only a single combination, which has the same elements as the delimited subsequence.) If `copy` is true (the default) each combination is freshly allocated. If `copy` is false all combinations are `eq` to each other, in which case consequences are unspecified if a combination is modified by `function`.

`map-derangements` *function sequence &key start end copy* [Function]
Calls `function` with each derangement of the subsequence of `sequence` denoted by the bounding index designators `start` and `end`. Derangement is a permutation of the sequence where no element remains in place. `sequence` is not modified, but individual derangements are `eq` to each other. Consequences are unspecified if calling `function` modifies either the derangement or `sequence`.

`map-permutations` *function sequence &key start end length copy* [Function]
Calls function with each permutation of `length` constructable from the subsequence of `sequence` delimited by `start` and `end`. `start` defaults to `0`, `end` to length of the sequence, and `length` to the length of the delimited subsequence.

5 IO

`read-stream-content-into-string` *stream &key buffer-size* [Function]
Return the "content" of `stream` as a fresh string.

`read-file-into-string` *pathname &key buffer-size external-format* [Function]
Return the contents of the file denoted by `pathname` as a fresh string.

The `external-format` parameter will be passed directly to `with-open-file` unless it's `nil`, which means the system default.

`read-stream-content-into-byte-vector` *stream &key %length initial-size* [Function]
Return "content" of `stream` as freshly allocated (unsigned-byte 8) vector.

`read-file-into-byte-vector` *pathname* [Function]
Read `pathname` into a freshly allocated (unsigned-byte 8) vector.

6 Macro Writing

`once-only` *specs &body forms* [Macro]

Constructs code whose primary goal is to help automate the handling of multiple evaluation within macros. Multiple evaluation is handled by introducing intermediate variables, in order to reuse the result of an expression.

The returned value is a list of the form

```
(let ((<gensym-1> <expr-1>)
      ...
      (<gensym-n> <expr-n>))
  <res>)
```

where `gensym-1`, ..., `gensym-n` are the intermediate variables introduced in order to evaluate `expr-1`, ..., `expr-n` once, only. `res` is code that is the result of evaluating the implicit progn `forms` within a special context determined by `specs`. `res` should make use of (reference) the intermediate variables.

Each element within `specs` is either a symbol `symbol` or a pair (SYMBOL INITFORM). Bare symbols are equivalent to the pair (SYMBOL SYMBOL).

Each pair (SYMBOL INITFORM) specifies a single intermediate variable:

- `initform` is an expression evaluated to produce EXPR-i
- `symbol` is the name of the variable that will be bound around `forms` to the corresponding gensym GENSYM-i, in order for `forms` to generate `res` that references the intermediate variable

The evaluation of INITFORMs and binding of SYMBOLs resembles `let`. INITFORMs of all the pairs are evaluated before binding SYMBOLs and evaluating `forms`.

Example:

The following expression

```
(let ((x '(incf y)))
  (once-only (x)
    `(cons ,x ,x)))
;;; =>
;;; (let ((#1=#:X123 (incf y)))
;;;   (cons #1# #1#))
```

could be used within a macro to avoid multiple evaluation like so

```
(defmacro cons1 (x)
  (once-only (x)
    `(cons ,x ,x)))
(let ((y 0))
  (cons1 (incf y)))
;;; => (1 . 1)
```

Example:

The following expression demonstrates the usage of the `initform` field

```
(let ((expr '(incf y)))
```

```
  (once-only ((var `(1+ ,expr)))
    `(list ',expr ,var ,var)))
;;; =>
;;; (let ((#1=#:VAR123 (1+ (incf y))))
;;;   (list '(incf y) #1# #1))
```

which could be used like so

```
(defmacro print-succ-twice (expr)
  (once-only ((var `(1+ ,expr)))
    `(format t "Expr: ~s, Once: ~s, Twice: ~s~%" ',expr ,var ,var)))
(let ((y 10))
  (print-succ-twice (incf y)))
;;; >>
;;; Expr: (INCF Y), Once: 12, Twice: 12
```

`with-gensyms` *names &body forms* [Macro]

Binds a set of variables to gensyms and evaluates the implicit progn `forms`.

Each element within `names` is either a symbol `symbol` or a pair (SYMBOL STRING-DESIGNATOR). Bare symbols are equivalent to the pair (SYMBOL SYMBOL).

Each pair (SYMBOL STRING-DESIGNATOR) specifies that the variable named by `symbol` should be bound to a symbol constructed using `gensym` with the string designated by `string-designator` being its first argument.

`with-unique-names` *names &body forms* [Macro]

Alias for `with-gensyms`.

`featurep` *feature-expression* [Function]

Returns `t` if the argument matches the state of the `*features*` list and `nil` if it does not. `feature-expression` can be any atom or list acceptable to the reader macros `#+` and `#-`.

`parse-body` *body &key documentation whole* [Function]

Parses `body` into (values remaining-forms declarations doc-string). Documentation strings are recognized only if `documentation` is true. Syntax errors in body are signalled and `whole` is used in the signal arguments when given.

`parse-ordinary-lambda-list` *lambda-list &key normalize allow-specializers normalize-optional normalize-keyword normalize-auxilary* [Function]

Parses an ordinary lambda-list, returning as multiple values:

1. Required parameters.
2. Optional parameter specifications, normalized into form:

   ```
   (name init suppliedp)
   ```

3. Name of the rest parameter, or `nil`.
4. Keyword parameter specifications, normalized into form:

   ```
   ((keyword-name name) init suppliedp)
   ```

5. Boolean indicating `&allow-other-keys` presence.
6. `&aux` parameter specifications, normalized into form
   `(name init)`.
7. Existence of `&key` in the lambda-list.

Signals a `program-error` is the lambda-list is malformed.

7 Symbols

`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. Returns a secondary value reflecting the status of the symbol in the package, which matches the secondary return value of `intern`.

Example:

```
(ensure-symbol :cons :cl) => cl:cons, :external
```

`format-symbol` *package control &rest arguments* [Function]

Constructs a string by applying `arguments` to string designator `control` as if by `format` within `with-standard-io-syntax`, 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`.

`make-keyword` *name* [Function]

Interns the string designated by `name` in the `keyword` package.

`make-gensym` *name* [Function]

If `name` is a non-negative integer, calls `gensym` using it. Otherwise `name` must be a string designator, in which case calls `gensym` using the designated string as the argument.

`make-gensym-list` *length &optional x* [Function]

Returns a list of `length` gensyms, each generated as if with a call to `make-gensym`, using the second (optional, defaulting to `"G"`) argument.

`symbolicate` *&rest things* [Function]

Concatenate together the names of some strings and symbols, producing a symbol in the current package.

8 Arrays

`array-index` [Type]

Type designator for an index into array of `length:` an integer between `0` (inclusive) and `length` (exclusive). `length` defaults to one less than `array-dimension-limit`.

`array-length` [Type]

Type designator for a dimension of an array of `length:` an integer between `0` (inclusive) and `length` (inclusive). `length` defaults to one less than `array-dimension-limit`.

`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.

9 Types

`string-designator` [Type]

A string designator type. A string designator is either a string, a symbol, or a character.

`coercef` *place type-spec* [Macro]

Modify-macro for `coerce`.

`of-type` *type* [Function]

Returns a function of one argument, which returns true when its argument is of `type`.

`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.

10 Numbers

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

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

`binomial-coefficient` *n k* [Function]
Binomial coefficient of `n` and `k`, also expressed as `n` choose `k`. This is the number of `k` element combinations given `n` choises. `n` must be equal to or greater then `k`.

`count-permutations` *n &optional k* [Function]
Number of `k` element permutations for a sequence of `n` objects. `k` defaults to `n`

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

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

`factorial` *n* [Function]
Factorial of non-negative integer `n`.

`subfactorial` *n* [Function]
Subfactorial of the non-negative integer `n`.

`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`.

Sufficiently positive `min` or negative `max` will cause the algorithm used to take a very long time. If `min` is positive it should be close to zero, and similarly if `max` is negative it should be close to zero.

`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 `1`.

Examples:

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

`map-iota` *function n &key start step* [Function]

Calls `function` with `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 `1`. Returns `n`.

Examples:

```
(map-iota #'print 3 :start 1 :step 1.0) => 3
  ;;; 1.0
  ;;; 2.0
  ;;; 3.0
```

`mean` *sample* [Function]

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

`median` *sample* [Function]

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

`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.

`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.