



## **Z3S5 Lisp Reference Manual**

by Erich Rast and all Help system contributors

2023-9-24 14:25



For Z3S5 Lisp Version 2.4+69079b3-gui.fyne2 with installed modules (gui fyne2 oop lib kvdb zimage tasks help beep db fileio decimal ling float console base).

## 1 Introduction

This is the reference manual for Z3S5 Lisp. This manual has been automatically generated from the entries of the online help system. The reference manual is divided into two large sections. Section By Topics lists functions and symbols organized by topics. Within each topic, entries are sorted alphabetically. Section Complete Reference lists all functions and symbols alphabetically. Please consult the *User Manual* and the *Readme* document for more general information about Z3S5 Lisp, an introduction to its use, and how to embedd it into Go programs.

Incorrect documentation strings are bugs. Please report bugs using the corresponding Github issue tracker for Z3S5 Lisp and be as precise as possible. Superfluous and missing documentation entries are misfeatures and may also be reported.

## 2 Index

```
% * *colors* *error-handler* *error-printer* *help* *hooks* *last-error* *reflect
* + - / /= 10th 1st 2nd 3rd 4th 5th 6th 7th 8th 9th < <= = > >= abs action action-start
action-stop activate-menu-last-submenu activate-menu-next activate-menu-
previous add-hook add-hook-internal add-hook-once add1 alist->dict alist? and
append append-form apply apropos array array->list array->str array-append array-
copy array-exists? array-forall? array-foreach array-len array-map! array-pmap!
array-ref array-reverse array-set array-slice array-sort array-walk array?
ascii85->blob assoc assoc1 assq atom? base64->blob beep bind bitand bitclear
bitor bitshl bitshr bitxor blob->ascii85 blob->base64 blob->hex blob->str blob
-chksum blob-equal? blob-free blob? bool? bound? boxed? build-array build-list
caaar caadr caar cadar caddr cadr call-method call-super can-externalize? car
case cmp cdaar cdadr cdar cddar cdddr cddr cdec! cdr center-window-on-screen
change-action-prefix change-all-action-prefixes char->str chars chars->str
cinc! class-name class-of class? close close-gui close-window closure? collect
-garbage color cons cons? copy-record count-partitions cpunum cst! current-
error-handler current-zimage cwait darken date->epoch-ns datelist->epoch-ns
datestr datestr* datestr->datelist day+ day-of-week db.blob db.close db.close-
result db.exec db.float db.int db.open db.open* db.query db.result-column-count
```

db.result-columns db.row db.step db.str deactivate-menu-child deactivate-menu  
 -last-submenu declare-**volatile** def-custom-hook **default**-error-handler defclass  
 defmacro defmethod defstruct defun delete dequeue! dict dict->alist dict->array  
 dict->keys dict->list dict->values dict-copy dict-empty? dict-foreach dict-map  
 dict-map! dict-merge dict-protect dict-**protected**? dict-unprotect dict? dir dir?  
 disable-object div dolist dotimes dump dump-bindings enable-object enq enqueue!  
 entry-accepts-tab? epoch-ns->datelist eq? eql? equal? error error->str error?  
 eval even? exists? exit expand-macros expect expect-err expect-**false** expect-  
 ok expect-**true** expr->str externalize externalize0 fdelete feature? file-port?  
 filter find-missing-help-entries find-unneeded-help-entries fl.abs fl.acos  
 fl.asin fl.asinh fl.atan fl.atan2 fl.atanh fl.cbrt fl.ceil fl.cos fl.cosh fl.dim  
 fl.erf fl.erfc fl.erfcinv fl.erfinv fl.exp fl.exp2 fl.expm1 fl.floor fl.fma fl  
 .frexp fl.gamma fl.hypot fl.ilogb fl.inf fl.is-nan? fl.j0 fl.j1 fl.jn fl.ldexp  
 fl.lgamma fl.log fl.log10 fl.log1p fl.log2 fl.logb fl.max fl.min fl.mod fl.modf  
 fl.nan fl.next-after fl.pow fl.pow10 fl.remainder fl.round fl.round-to-even  
 fl.signbit fl.sin fl.sinh fl.sqrt fl.tan fl.tanh fl.trunc fl.y0 fl.y1 fl.yn  
 flatten **float** fmt forall? force foreach forget forget-gui-object functional-arity  
 functional-has-rest? functional? gensym get get-action get-clipboard-content  
 get-device-info get-entry-cursor get-entry-cursor-pos get-menu-item-**label** get-  
 object-min-size get-object-position get-object-size get-or-set get-partitions  
 get-progress-bar-value get-text-grid-cell get-text-grid-row get-text-grid-row  
 -text get-text-grid-tab-width get-text-grid-text get-window-canvas get-window-  
 content get-window-icon get-window-main-menu get-window-title getstacked glance  
 global-sym? has has-action-system? has-action? has-key? has-method? has-prop?  
 help help->manual-entry help-about help-entry help-strings help-topic-info  
 help-topics hex->blob hide-object hide-window hook hour+ identity **if** inchars  
 include index init-actions init-remember instr **int** intern internalize intrinsic  
 intrinsic? isa? iterate kvdb.begin kvdb.close kvdb.commit kvdb.db? kvdb.forget  
 kvdb.forget-everything kvdb.get kvdb.info kvdb.open kvdb.rollback kvdb.search  
 kvdb.set kvdb.when last lcons len let letrec lighten ling.damerau-levenshtein ling  
 .hamming ling.jaro ling.jaro-winkler ling.levenshtein ling.match-rating-codex  
 ling.match-rating-compare ling.metaphone ling.nysiis ling.porter ling.soundex  
 list list->array list->set list->str list-exists? list-forall? list-foreach  
 list-last list-ref list-reverse list-slice list? load load-zimage macro? make  
 make\* make-blob make-mutex make-queue make-set make-stack make-symbol map map-  
 pairwise mapcar max member memq memstats menu-item-checked? menu-item-disabled?  
 methods min minmax minute+ mod month+ move-object mutex-lock mutex-rlock mutex  
 -runlock mutex-unlock nconc **new** **new**-app-tabs **new**-border **new**-button **new**-button

-with-icon **new**-center-layout **new**-check **new**-choice **new**-circle **new**-combined  
 -string-validator **new**-container **new**-container-without-layout **new**-doc-tabs  
**new**-entry **new**-form **new**-form-layout **new**-grid-layout **new**-grid-wrap-layout **new**-  
 hbox-layout **new**-hyperlink **new**-icon **new**-image-from-file **new**-image-from-resource  
**new**-label **new**-line **new**-list **new**-main-menu **new**-menu **new**-menu\* **new**-menu-item **new**-  
 -menu-item-separator **new**-progress-bar **new**-raster-with-pixels **new**-rectangle  
**new**-regexp-validator **new**-shortcut **new**-slider **new**-spacer **new**-stack-layout **new**-  
 -struct **new**-tabitem **new**-tabitem-with-icon **new**-table **new**-text **new**-text-grid  
**new**-time-validator **new**-tree **new**-validator **new**-vbox-layout **new**-window nl nonce  
 not now now-ms now-ns nreverse nrgba nth nth-partition nthdef **null?** num? object-  
 disabled? object? odd? on-feature open or out outy peek permission? permissions  
 poke pop! pop-error-handler pop-finalizer popstacked prin1 princ print proc?  
 prop props protect protect-toplevel-symbols **protected?** prune-task-table prune  
 -unneeded-help-entries push! push-error-handler push-finalizer pushstacked  
 queue-empty? queue-len queue? rand random-color read read-binary read-string  
 read-zimage readall readall-str recall recall-info recall-when recollect record?  
 refresh-main-menu refresh-menu\* refresh-object register-action remember remove-  
 duplicates remove-hook remove-hook-internal remove-hooks rename-action replace-  
 hook reset-color resize-object reverse rnd rndseed rplaca run-at run-hook run-hook  
 -internal run-selftest run-zimage save-zimage sec+ semver.build semver.canonical  
 semver.compare semver.is-valid? semver.major semver.major-minor semver.max  
 semver.prerelease seq? set set\* set->list set-clipboard-content set-color set  
 -complement set-difference set-element? set-empty? set-entry-cursor-column  
 set-entry-cursor-row set-entry-min-rows-visible set-entry-on-change-callback  
 set-entry-on-cursor-change-callback set-entry-place-holder set-entry-text set-  
 entry-validator set-equal? set-help-topic-info set-intersection set-**label**-text  
 set-menu-item-checked set-menu-item-disabled set-menu-item-**label** set-object  
 -on-validation-change-callback set-permissions set-progress-bar set-slider  
 -value set-subset? set-text-alignment set-text-grid-cell set-text-grid-row  
 set-text-grid-row-style set-text-grid-rune set-text-grid-show-line-numbers  
 set-text-grid-show-whitespace set-text-grid-style set-text-grid-style-range  
 set-text-grid-tab-width set-text-grid-text set-text-size set-text-style set  
 -union set-volume set-window-content set-window-full-screen set-window-icon  
 set-window-main-menu set-window-on-close-callback set-window-padded set-  
 window-size set-window-title set? setcar setcdr setprop shorten show-object sleep  
 sleep-ns slice sort sort-symbols spaces stack-empty? stack-len stack? str+ str  
 ->array str->blob str->**char** str->chars str->expr str->expr\* str->list str->sym  
 str-count-substr str-empty? str-exists? str-forall? str-foreach str-index str

`-join str-port? str-ref str-remove-number str-remove-prefix str-remove-suffix`  
`str-replace str-replace* str-reverse str-segment str-slice str? strbuild strcase`  
`strcenter strcnt strleft strlen strless strlimit strmap stropen strright strsplit`  
`struct-index struct-instantiate struct-name struct-props struct-size struct?`  
`sub1 supers sym->str sym? synout synouty sys-key? sysmsg sysmsg* take task task`  
`-broadcast task-recv task-remove task-run task-schedule task-send task-state`  
`task? terpri testing text-grid-show-line-numbers? text-grid-show-whitespace?`  
`the-color the-color-names theme-icon time trigger-menu-last truncate try type-of`  
`type-of* unless unprotect unprotect-toplevel-symbols valid? validate-object`  
`void void? wait-for wait-for* wait-for-empty* wait-until wait-until* warn week+`  
`week-of-date when when-permission while window-fixed-size? window-full-screen?`  
`window-padded? with-colors with-error-handler with-final with-mutex-lock with-`  
`mutex-rlock write write-binary write-binary-at write-string write-zimage year+`  
`zimage-header zimage-loadable? zimage-runable?`

## 3 By Topics

### 3.1 Actions

This section concerns the action class and related functions. Actions can be used as an asynchronous interface to the host system, provided the functions `action.start`, `action.progress`, `action.result`, and `action.get-args` are defined. These functions serve as callbacks into the Go part and need to be implemented on the Go side by the user of the action system. The host system must find the action initialization code and execute it; this code should call `register-action` to register any actions provided, and then the host system may call `get-action` and `action-start` to execute the action within Lisp. Procedure `action-start` takes an action and a taskid and performs the action. To make `action-stop` work, you have listen to the 'stop message using `task-recv` and shutdown the action appropriately. While the action runs, periodically call `action.progress`. Call `action.result` once the action ends or if an error occurs that does not allow the action to complete. Use `action.get-args` to obtain the arguments of the action, which must be an array of valid Z3S5 Lisp objects. The host system might e.g. prompt the user for values, or these may depend on selected objects in a GUI interface. The host interfacing functions generally receive the action, and its name as second and the ID symbol as third argument in addition to other arguments. The second and third argument are provided for convenience, since processing a #name string and an #id symbol is much easier for a dispatch function in Go than the #action itself, which is an object instance and internally represented as a complex array.

### 3.1.1 action : class

Usage: (**new** *action* <info-clause> <name-clause> <proc-clause> ...) => *action*

The action class describes instances of actions that serve as plugins for the system hosting Z3S5 Lisp. Each action has a *name*, *prefix* and *info* string property and a unique *id*. Property *args* is an array that specifies the type of arguments of the action. This may be used by an implementation of *action.get-args*. The *proc* property must be a function taking the action and a task-id as argument and processing the action sequentially until it is completed or *task-recv* returns the 'stop signal. An action may store the result of computation in the *result* property, an error in the *error* property, and an arbitrary state in the *state* property. After processing or if an error occurs, *action.result* should be called so the host can process the result or error. The action system requires the implementation of procedures *action.start*, *action.progress*, *action.get-args*, and *action.result*. These are usually defined in the host system, for example in the Go implementation of an application using Z3S5 Lisp actions, and serve as callback functions from Lisp to Go.

See also: *action*, *action-stop*, *action.start*, *action.progress*, *action.get-args*, *action.result*. →index

### 3.1.2 action-start : method

Usage: (*action-start* *action*)

Start *action*, which runs the action's *proc* in a task with the action and a task-id as argument. The *proc* of the *action* should periodically check for the 'stop signal using *task-recv* if the action should be cancellable, should call *action.progress* to report progress, *action.error* in case of an error, and *action.result* to report the result.

See also: *action*, *action-stop*, *action-start*, *action.progress*, *action.get-args*, *action.result*. →index

### 3.1.3 action-stop : method

Usage: (*action-stop* *action*)

The stop method sends a 'stop signal to the action's running *proc*. It is up to the *proc* to check for the signal using *task-recv* and terminate the action gracefully.

See also: *action*, *action-stop*, *action-start*, *action.progress*, *action.get-args*, *action.result*. →index

### 3.1.4 `action.get-args`: procedure/3

Usage: (`action.get-args` `prefix` `name` `id` `arg-spec`)=> `array`

Used to request an array of arguments for an action with `prefix`, `name` and `id` from the host system, according to the specification given in `arg-spec`, which is usually the same as `argspec`.

See also: `action`, `action-stop`, `action.start`, `action.progress`, `action.get-args`, `action.result`. →index

### 3.1.5 `action.progress`: procedure/5

Usage: (`action.progress` `prefix` `name` `id` `taskid` `perc` `msg`)

Used to notify the host system from within a running `proc` that the action with `prefix`, `name`, `id`, and `taskid` is making progress to `perc` (a float between 0 and 1) with a message `msg`. Leave the message string empty if it is not needed. Implemented in the host system in Go, this function may, for instance, display a progress bar to inform an end-user.

See also: `action`, `action-stop`, `action.start`, `action.start`, `action.progress`, `action.get-args`, `action.result`. →index

### 3.1.6 `action.result`: procedure/5

Usage: (`action.result` `prefix` `name` `id` `taskid` `result` `error?`)

Used to notify the host system of the result of an action with `prefix`, `name`, `id`, and `taskid`. The `result` may be of any type, but `error?` needs to be a bool that indicates whether an error has occurred. If `error?` is not nil, then the host implementation should interpret `result` as an error message.

See also: `action`, `action-stop`, `action.start`, `action.progress`, `action.get-args`, `action.result`. →index

### 3.1.7 `action.start`: procedure/3

Usage: (`action.start` `prefix` `name` `id` `taskid`)

Used to notify the host system that the action with `prefix`, `name`, `id`, and `taskid` has been started.

See also: `action`, `action-stop`, `action.start`, `action.start`, `action.progress`, `action.get-args`, `action.result`. →index

### 3.1.8 `change-action-prefix` : procedure/2

Usage: (`change-action-prefix` `id` `new-prefix`)=> `bool`

Change the prefix of a registered action with given `id`, or change the prefix of action given by `id`, to `new-prefix`. If the operation succeeds, it returns true, otherwise it returns nil.

See also: `change-all-action-prefixes`, `rename-action`, `get-action`, `action?`, `action`.  
→index

### 3.1.9 `change-all-action-prefixes` : procedure/2

Usage: (`change-all-action-prefixes` `old-prefix` `new-prefix`)

Change the prefixes of all registered actions with `old-prefix` to `new-prefix`.

See also: `change-action-prefix`, `rename-action`, `get-action`, `register-action`, `action?`, `action`. →index

### 3.1.10 `get-action` : procedure/1

Usage: (`get-action` `id`)=> `action`

Return a cloned action based on `id` from the action registry. This action can be run using `action-start` and will get its own taskid.

See also: `action`, `has-action-system?`, `action-start`, `action-stop`, `register-action`.  
→index

### 3.1.11 `has-action-system?` : procedure/0

Usage: (`has-action-system?`)=> `bool`

This predicate is true if the action system is available, `false` otherwise.

See also: `action`, `init-actions`, `action-start`, `action-stop`, `registered-actions`, `register-action`. →index

### 3.1.12 `has-action?` : procedure/1

Usage: (`has-action?` `prefix` `name`)=> `bool`



Return true if an action with the given `prefix` and `name` is registered, nil otherwise. Actions are indexed by id, so this is much slower than using `get-action` to retrieve a registered action by the value of the 'id property.

See also: `get-action`, `action`, `has-action-system?`, `register-action`. →index

### 3.1.13 `init-actions:procedure/0`

Usage: (`init-actions`)

Initialize the action system, signals an error if the action system is not available.

See also: `action`, `has-action-system?`, `action-start`, `action-stop`. →index

### 3.1.14 `register-action:procedure/1`

Usage: (`register-action action`)

Register the `action` which makes it available for processing by the host system. Use `get-action` to obtain an action clone that can be started.

See also: `action`, `has-action-system?`, `action-start`, `action-stop`. →index

### 3.1.15 `rename-action:procedure/2`

Usage: (`rename-action id new-name`)=> `bool`

Rename a registered action with given `id`, or rename the action given as `id`, to `new-name`. If the operation succeeds, it returns true, otherwise it returns nil.

See also: `change-action-prefix`, `change-all-action-prefixes`, `get-action`, `has-action?`, `action`. →index

## 3.2 Arrays

This section concerns functions related to arrays, which are dynamic indexed sequences of values.

### 3.2.1 `array:procedure/0 or more`

Usage: (`array [arg1] ...`)=> `array`

Create an array containing the arguments given to it.

See also: `array?`, `build-array`. →index

### 3.2.2 `array-append` : procedure/2

Usage: `(array-append arr elem)=> array`

Append `elem` to the array `arr`. This function is destructive and mutates the array. Use `array-copy` if you need a copy.

See also: `array-ref`, `array-len`, `build-array`, `array-slice`, `array`, `array-copy`. →index

### 3.2.3 `array-copy` : procedure/1

Usage: `(array-copy arr)=> array`

Return a copy of `arr`.

See also: `array`, `array?`, `array-map!`, `array-pmap!`. →index

### 3.2.4 `array-exists?` : procedure/2

Usage: `(array-exists? arr pred)=> bool`

Return true if `pred` returns true for at least one element in array `arr`, nil otherwise.

See also: `exists?`, `forall?`, `list-exists?`, `str-exists?`, `seq?`. →index

### 3.2.5 `array-forall?` : procedure/2

Usage: `(array-forall? arr pred)=> bool`

Return true if predicate `pred` returns true for all elements of array `arr`, nil otherwise.

See also: `foreach`, `map`, `forall?`, `str-forall?`, `list-forall?`, `exists?`. →index

### 3.2.6 `array-foreach` : procedure/2

Usage: `(array-foreach arr proc)`

Apply `proc` to each element of array `arr` in order, for the side effects.

See also: `foreach`, `list-foreach`, `map`. →index

### 3.2.7 `array-len` : procedure/1

Usage: `(array-len arr)` => `int`

Return the length of array `arr`.

See also: `len`. →index

### 3.2.8 `array-map!` : procedure/2

Usage: `(array-map! arr proc)`

Traverse array `arr` in unspecified order and apply `proc` to each element. This mutates the array.

See also: `array-walk`, `array-pmap!`, `array?`, `map`, `seq?`. →index

### 3.2.9 `array-pmap!` : procedure/2

Usage: `(array-pmap! arr proc)`

Apply `proc` in unspecified order in parallel to array `arr`, mutating the array to contain the value returned by `proc` each time. Because of the calling overhead for parallel execution, for many workloads `array-map!` might be faster if `proc` is very fast. If `proc` is slow, then `array-pmap!` may be much faster for large arrays on machines with many cores.

See also: `array-map!`, `array-walk`, `array?`, `map`, `seq?`. →index

### 3.2.10 `array-ref` : procedure/1

Usage: `(array-ref arr n)` => `any`

Return the element of `arr` at index `n`. Arrays are 0-indexed.

See also: `array?`, `array`, `nth`, `seq?`. →index

### 3.2.11 `array-reverse` : procedure/1

Usage: `(array-reverse arr)` => `array`

Create a copy of `arr` that reverses the order of all of its elements.

See also: `reverse`, `list-reverse`, `str-reverse`. →index

### 3.2.12 `array-set` : procedure/3

Usage: `(array-set arr idx value)`

Set the value at index `idx` in `arr` to `value`. Arrays are 0-indexed. This mutates the array.

See also: `array?`, `array`. →index

### 3.2.13 `array-slice` : procedure/3

Usage: `(array-slice arr low high)=> array`

Slice the array `arr` starting from `low` (inclusive) and ending at `high` (exclusive) and return the slice. This function is destructive and mutates the slice. Use `array-copy` if you need a copy.

See also: `array-ref`, `array-len`, `array-append`, `build-array`, `array`, `array-copy`. →index

### 3.2.14 `array-sort` : procedure/2

Usage: `(array-sort arr proc)=> arr`

Destructively sorts array `arr` by using comparison proc `proc`, which takes two arguments and returns true if the first argument is smaller than the second argument, nil otherwise. The array is returned but it is not copied and modified in place by this procedure. The sorting algorithm is not guaranteed to be stable.

See also: `sort`. →index

### 3.2.15 `array-walk` : procedure/2

Usage: `(array-walk arr proc)`

Traverse the array `arr` from first to last element and apply `proc` to each element for side-effects. Function `proc` takes the index and the array element at that index as argument. If `proc` returns nil, then the traversal stops and the index is returned. If `proc` returns non-nil, traversal continues. If `proc` never returns nil, then the index returned is -1. This function does not mutate the array.

See also: `array-map!`, `array-pmap!`, `array?`, `map`, `seq?`. →index

### 3.2.16 `array?` : procedure/1

Usage: `(array? obj)=> bool`

Return true of `obj` is an array, nil otherwise.

See also: `seq?`, `array`. →index

### 3.2.17 `build-array`: procedure/2

Usage: (`build-array` `n` `init`)=> `array`

Create an array containing `n` elements with initial value `init`.

See also: `array`, `array?`, `array-slice`, `array-append`, `array-copy`. →index

## 3.3 Binary Manipulation

This section lists functions for manipulating binary data in memory and on disk.

### 3.3.1 `bitand`: procedure/2

Usage: (`bitand` `n` `m`)=> `int`

Return the bitwise and of integers `n` and `m`.

See also: `bitxor`, `bitor`, `bitclear`, `bitshl`, `bitshr`. →index

### 3.3.2 `bitclear`: procedure/2

Usage: (`bitclear` `n` `m`)=> `int`

Return the bitwise and-not of integers `n` and `m`.

See also: `bitxor`, `bitand`, `bitor`, `bitshl`, `bitshr`. →index

### 3.3.3 `bitor`: procedure/2

Usage: (`bitor` `n` `m`)=> `int`

Return the bitwise or of integers `n` and `m`.

See also: `bitxor`, `bitand`, `bitclear`, `bitshl`, `bitshr`. →index

### 3.3.4 `bitshl`: procedure/2

Usage: `(bitshl n m)`=> `int`

Return the bitwise left shift of `n` by `m`.

See also: `bitxor`, `bitor`, `bitand`, `bitclear`, `bitshr`. →index

### 3.3.5 `bitshr`: procedure/2

Usage: `(bitshr n m)`=> `int`

Return the bitwise right shift of `n` by `m`.

See also: `bitxor`, `bitor`, `bitand`, `bitclear`, `bitshl`. →index

### 3.3.6 `bitxor`: procedure/2

Usage: `(bitxor n m)`=> `int`

Return the bitwise exclusive or value of integers `n` and `m`.

See also: `bitand`, `bitor`, `bitclear`, `bitshl`, `bitshr`. →index

### 3.3.7 `blob-chksum`: procedure/1 or more

Usage: `(blob-chksum b [start] [end])`=> `blob`

Return the checksum of the contents of blob `b` as new blob. The checksum is cryptographically secure. If the optional `start` and `end` are provided, then only the bytes from `start` (inclusive) to `end` (exclusive) are checksummed.

See also: `fchksum`, `blob-free`. →index

### 3.3.8 `blob-equal?`: procedure/2

Usage: `(blob-equal? b1 b2)`=> `bool`

Return true if `b1` and `b2` are equal, nil otherwise. Two blobs are equal if they are either both invalid, both contain no valid data, or their contents contain exactly the same binary data.

See also: `str->blob`, `blob->str`, `blob-free`. →index

### 3.3.9 `blob-free` : procedure/1

Usage: (`blob-free` *b*)

Frees the binary data stored in blob *b* and makes the blob invalid.

See also: `make-blob`, `valid?`, `str->blob`, `blob->str`, `blob-equal?`. →index

### 3.3.10 `blob?` : procedure/1

Usage: (`blob?` *obj*)=> *bool*

Return true if *obj* is a binary blob, nil otherwise.

See also: `blob->ascii85`, `blob->base64`, `blob->hex`, `blob->str`, `blob-free`, `blob-chksum`, `blob-equal?`, `valid?`. →index

### 3.3.11 `make-blob` : procedure/1

Usage: (`make-blob` *n*)=> *blob*

Make a binary blob of size *n* initialized to zeroes.

See also: `blob-free`, `valid?`, `blob-equal?`. →index

### 3.3.12 `peek` : procedure/4

Usage: (`peek` *b* *pos* *end* *sel*)=> *num*

Read a numeric value determined by selector *sel* from binary blob *b* at position *pos* with endianness *end*. Possible values for endianness are 'little and 'big, and possible values for *sel* must be one of '(bool int8 uint8 int16 uint16 int32 uint32 int64 uint64 float32 float64).

See also: `poke`, `read-binary`. →index

### 3.3.13 `poke` : procedure/5

Usage: (`poke` *b* *pos* *end* *sel* *n*)

Write numeric value *n* as type *sel* with endianness *end* into the binary blob *b* at position *pos*. Possible values for endianness are 'little and 'big, and possible values for *sel* must be one of '(bool int8 uint8 int16 uint16 int32 uint32 int64 uint64 float32 float64).

See also: `peek`, `write-binary`. →index

### 3.4 Boxed Data Structures

Boxed values are used for dealing with foreign data structures in Lisp.

#### 3.4.1 `valid?` : procedure/1

Usage: (`valid?` `obj`)=> `bool`

Return true if `obj` is a valid object, nil otherwise. What exactly object validity means is undefined, but certain kind of objects such as graphics objects may be marked invalid when they can no longer be used because they have been disposed off by a subsystem and cannot be automatically garbage collected. Generally, invalid objects ought no longer be used and need to be discarded.

See also: `blob?`. →index

### 3.5 Concurrency and Parallel Programming

There are several mechanisms for doing parallel and concurrent programming in Z3S5 Lisp. Synchronization primitives are also listed in this section. Generally, users are advised to remain vigilant about potential race conditions.

#### 3.5.1 `ccmp` : macro/2

Usage: (`ccmp` `sym` `value`)=> `int`

Compare the integer value of `sym` with the integer `value`, return 0 if `sym` = `value`, -1 if `sym` < `value`, and 1 if `sym` > `value`. This operation is synchronized between tasks and futures.

See also: `cinc!`, `cdec!`, `cwait`, `cst!`. →index

#### 3.5.2 `cdec!` : macro/1

Usage: (`cdec!` `sym`)=> `int`

Decrease the integer value stored in top-level symbol `sym` by 1 and return the new value. This operation is synchronized between tasks and futures.

See also: `cinc!`, `cwait`, `ccmp`, `cst!`. →index



### 3.5.3 `cinc!` : macro/1

Usage: (`cinc!` `sym`)=> `int`

Increase the integer value stored in top-level symbol `sym` by 1 and return the new value. This operation is synchronized between tasks and futures.

See also: `cdec!`, `cwait`, `ccmp`, `cst!`. →index

### 3.5.4 `cpunum` : procedure/0

Usage: (`cpunum`)

Return the number of cpu cores of this machine.

See also: `sys`. →index

**Warning: This function also counts virtual cores on the emulator. The original Z3S5 machine did not have virtual cpu cores.**

### 3.5.5 `cst!` : procedure/2

Usage: (`cst!` `sym` `value`)

Set the value of `sym` to integer `value`. This operation is synchronized between tasks and futures.

See also: `cinc!`, `cdec!`, `ccmp`, `cwait`. →index

### 3.5.6 `cwait` : procedure/3

Usage: (`cwait` `sym` `value` `timeout`)

Wait until integer counter `sym` has `value` or `timeout` milliseconds have passed. If `timeout` is 0, then this routine might wait indefinitely. This operation is synchronized between tasks and futures.

See also: `cinc!`, `cdec!`, `ccmp`, `cst!`. →index

### 3.5.7 `enq` : procedure/1

Usage: (`enq` `proc`)

Put `proc` on a special internal queue for sequential execution and execute it when able. `proc` must be a procedure that takes no arguments. The queue can be used to synchronizing i/o commands but special care must be taken that `proc` terminates, or else the system might be damaged.

See also: [task](#), [future](#), [synout](#), [synouty](#). [→index](#)

**Warning: Calls to `enq` can never be nested, neither explicitly or implicitly by calling `enq` anywhere else in the call chain!**

### 3.5.8 `force` : procedure/1

Usage: (`force fut`)=> `any`

Obtain the value of the computation encapsulated by future `fut`, halting the current task until it has been obtained. If the future never ends computation, e.g. in an infinite loop, the program may halt indefinitely.

See also: [future](#), [task](#), [make-mutex](#). [→index](#)

### 3.5.9 `future` : special form

Usage: (`future ...`)=> `future`

Turn the body of this form into a promise for a future value. The body is executed in parallel and the final value can be retrieved by using (`force f`) on the future returned by this macro.

See also: [force](#), [task](#). [→index](#)

### 3.5.10 `make-mutex` : procedure/1

Usage: (`make-mutex`)=> `mutex`

Create a new mutex.

See also: [mutex-lock](#), [mutex-unlock](#), [mutex-rlock](#), [mutex-runlock](#). [→index](#)

### 3.5.11 `mutex-lock` : procedure/1

Usage: (`mutex-lock m`)

Lock the mutex `m` for writing. This may halt the current task until the mutex has been unlocked by another task.

See also: [mutex-unlock](#), [make-mutex](#), [mutex-rlock](#), [mutex-runlock](#). [→index](#)

### 3.5.12 `mutex-rlock` : procedure/1

Usage: (`mutex-rlock` `m`)

Lock the mutex `m` for reading. This will allow other tasks to read from it, too, but may block if another task is currently locking it for writing.

See also: `mutex-runlock`, `mutex-lock`, `mutex-unlock`, `make-mutex`. →index

### 3.5.13 `mutex-runlock` : procedure/1

Usage: (`mutex-runlock` `m`)

Unlock the mutex `m` from reading.

See also: `mutex-lock`, `mutex-unlock`, `mutex-rlock`, `make-mutex`. →index

### 3.5.14 `mutex-unlock` : procedure/1

Usage: (`mutex-unlock` `m`)

Unlock the mutex `m` for writing. This releases ownership of the mutex and allows other tasks to lock it for writing.

See also: `mutex-lock`, `make-mutex`, `mutex-rlock`, `mutex-runlock`. →index

### 3.5.15 `prune-task-table` : procedure/0

Usage: (`prune-task-table`)

Remove tasks that are finished from the task table. This includes tasks for which an error has occurred.

See also: `task-remove`, `task`, `task?`, `task-run`. →index

### 3.5.16 `run-at` : procedure/2

Usage: (`run-at` `date` `repeater` `proc`)=> `int`

Run procedure `proc` with no arguments as task periodically according to the specification in `spec` and return the task ID for the periodic task. Herbey, `date` is either a datetime specification or one of '(now skip next-minute next-quarter next-halfhour next-hour in-2-hours in-3-hours tomorrow next-week next-month next-year), and `repeater` is nil or a procedure that takes a task ID and unix-epoch-nanoseconds

and yields a new unix-epoch-nanoseconds value for the next time the procedure shall be run. While the other names are self-explanatory, the 'skip specification means that the task is not run immediately but rather that it is first run at (repeater -1 (now)). Timing resolution for the scheduler is about 1 minute. Consider using interrupts for periodic events with smaller time resolutions. The scheduler uses relative intervals and has 'drift'.

See also: [task](#), [task-send](#). →index

**Warning: Tasks scheduled by run-at are not persistent! They are only run until the system is shutdown.**

### 3.5.17 systask : special form

Usage: ([systask](#) [body](#) ...)

Evaluate the expressions of [body](#) in parallel in a system task, which is similar to a future but cannot be forced.

See also: [future](#), [task](#). →index

### 3.5.18 task : procedure/1

Usage: ([task](#) [sel](#) [proc](#))=> [int](#)

Create a new task for concurrently running [proc](#), a procedure that takes its own ID as argument. The [sel](#) argument must be a symbol in '(auto manual remove). If [sel](#) is 'remove, then the task is always removed from the task table after it has finished, even if an error has occurred. If [sel](#) is 'auto, then the task is removed from the task table if it ends without producing an error. If [sel](#) is 'manual then the task is not removed from the task table, its state is either 'canceled, 'finished, or 'error, and it and must be removed manually with [task-remove](#) or [prune-task-table](#). Broadcast messages are never removed. Tasks are more heavy-weight than futures and allow for message-passing.

See also: [task?](#), [task-run](#), [task-state](#), [task-broadcast](#), [task-send](#), [task-recv](#), [task-remove](#), [prune-task-table](#). →index

### 3.5.19 task-broadcast : procedure/2

Usage: ([task-broadcast](#) [id](#) [msg](#))

Send a message from task [id](#) to the blackboard. Tasks automatically send the message 'finished to the blackboard when they are finished.

See also: [task](#), [task?](#), [task-run](#), [task-state](#), [task-send](#), [task-recv](#). →index

### 3.5.20 `task-recv` : procedure/1

Usage: (`task-recv id`)=> *any*

Receive a message for task `id`, or nil if there is no message. This is typically used by the task with `id` itself to periodically check for new messages while doing other work. By convention, if a task receives the message 'end it ought to terminate at the next convenient occasion, whereas upon receiving 'cancel it ought to terminate in an expedited manner.

See also: `task-send`, `task`, `task?`, `task-run`, `task-state`, `task-broadcast`. →index

**Warning: Busy polling for new messages in a tight loop is inefficient and ought to be avoided.**

### 3.5.21 `task-remove` : procedure/1

Usage: (`task-remove id`)

Remove task `id` from the task table. The task can no longer be interacted with.

See also: `task`, `task?`, `task-state`. →index

### 3.5.22 `task-run` : procedure/1

Usage: (`task-run id`)

Run task `id`, which must have been previously created with `task`. Attempting to run a task that is already running results in an error unless `silent?` is true. If `silent?` is true, the function does never produce an error.

See also: `task`, `task?`, `task-state`, `task-send`, `task-recv`, `task-broadcast-`. →index

### 3.5.23 `task-schedule` : procedure/1

Usage: (`task-schedule sel id`)

Schedule task `id` for running, starting it as soon as other tasks have finished. The scheduler attempts to avoid running more than (`cpunum`) tasks at once.

See also: `task`, `task-run`. →index

### 3.5.24 `task-send` : procedure/2

Usage: (`task-send` `id` `msg`)

Send a message `msg` to task `id`. The task needs to cooperatively use `task-recv` to reply to the message. It is up to the receiving task what to do with the message once it has been received, or how often to check for new messages.

See also: `task-broadcast`, `task-recv`, `task`, `task?`, `task-run`, `task-state`. →index

### 3.5.25 `task-state` : procedure/1

Usage: (`task-state` `id`)=> `sym`

Return the state of the task, which is a symbol in '(finished error stopped new waiting running).

See also: `task`, `task?`, `task-run`, `task-broadcast`, `task-recv`, `task-send`. →index

### 3.5.26 `task?` : procedure/1

Usage: (`task?` `id`)=> `bool`

Check whether the given `id` is for a valid task, return true if it is valid, nil otherwise.

See also: `task`, `task-run`, `task-state`, `task-broadcast`, `task-send`, `task-recv`. →index

### 3.5.27 `wait-for` : procedure/2

Usage: (`wait-for` `dict` `key`)

Block execution until the value for `key` in `dict` is not-nil. This function may wait indefinitely if no other thread sets the value for `key` to not-nil.

See also: `wait-for*`, `future`, `force`, `wait-until`, `wait-until*`. →index

**Warning:** This cannot be used for synchronization of multiple tasks due to potential race-conditions.

### 3.5.28 `wait-for*` : procedure/3

Usage: (`wait-for*` `dict` `key` `timeout`)

Blocks execution until the value for `key` in `dict` is not-nil or `timeout` nanoseconds have passed, and returns that value or nil if waiting timed out. If `timeout` is negative, then the function waits potentially

indefinitely without any timeout. If a non-nil key is not found, the function sleeps at least *sync-wait-lower-bound* nanoseconds and up to *sync-wait-upper-bound* nanoseconds until it looks for the key again.

See also: [future](#), [force](#), [wait-for](#), [wait-until](#), [wait-until\\*](#). [→index](#)

**Warning: This cannot be used for synchronization of multiple tasks due to potential race-conditions.**

### 3.5.29 `wait-for-empty*`: procedure/3

Usage: (`wait-for-empty*` dict key timeout)

Blocks execution until the `key` is no longer present in `dict` or `timeout` nanoseconds have passed. If `timeout` is negative, then the function waits potentially indefinitely without any timeout.

See also: [future](#), [force](#), [wait-for](#), [wait-until](#), [wait-until\\*](#). [→index](#)

**Warning: This cannot be used for synchronization of multiple tasks due to potential race-conditions.**

### 3.5.30 `wait-until`: procedure/2

Usage: (`wait-until` dict key pred)

Blocks execution until the unary predicate `pred` returns true for the value at `key` in `dict`. This function may wait indefinitely if no other thread sets the value in such a way that `pred` returns true when applied to it.

See also: [wait-for](#), [future](#), [force](#), [wait-until\\*](#). [→index](#)

**Warning: This cannot be used for synchronization of multiple tasks due to potential race-conditions.**

### 3.5.31 `wait-until*`: procedure/4

Usage: (`wait-until*` dict key pred timeout)

Blocks execution until the unary predicate `pred` returns true for the value at `key` in `dict`, or `timeout` nanoseconds have passed, and returns the value or nil if waiting timed out. If `timeout` is negative, then the function waits potentially indefinitely without any timeout. If a non-nil key is not found, the function sleeps at least *sync-wait-lower-bound* nanoseconds and up to *sync-wait-upper-bound* nanoseconds until it looks for the key again.

See also: [future](#), [force](#), [wait-for](#), [wait-until\\*](#), [wait-until](#). →index

**Warning: This cannot be used for synchronization of multiple tasks due to potential race-conditions.**

### 3.5.32 with-mutex-rlock: macro/1 or more

Usage: ([with-mutex-rlock](#) *m* ...) => *any*

Execute the body with mutex *m* locked for reading and unlock the mutex afterwards.

See also: [with-mutex-lock](#), [make-mutex](#), [mutex-lock](#), [mutex-rlock](#), [mutex-unlock](#), [mutex-runlock](#). →index

## 3.6 Console Input & Output

These functions access the operating system console (terminal) mostly for string output.

### 3.6.1 nl: procedure/0

Usage: ([nl](#))

Display a newline, advancing the cursor to the next line.

See also: [out](#), [outy](#), [output-at](#). →index

### 3.6.2 prin1: procedure/1

Usage: ([prin1](#) *s*)

Print *s* to the host OS terminal, where strings are quoted.

See also: [princ](#), [terpri](#), [out](#), [outy](#). →index

### 3.6.3 princ: procedure/1

Usage: ([princ](#) *s*)

Print *s* to the host OS terminal without quoting strings.

See also: [prin1](#), [terpri](#), [out](#), [outy](#). →index



### 3.6.4 `print`: procedure/1

Usage: (`print` `x`)

Output `x` on the host OS console and end it with a newline.

See also: `prin1`, `princ`. →index

### 3.6.5 `terpri`: procedure/0

Usage: (`terpri`)

Advance the host OS terminal to the next line.

See also: `princ`, `out`, `outy`. →index

## 3.7 Data Type Conversion

This section lists various ways in which one data type can be converted to another.

### 3.7.1 `alist->dict`: procedure/1

Usage: (`alist->dict` `li`)=> `dict`

Convert an association list `li` into a dictionary. Note that the value will be the cdr of each list element, not the second element, so you need to use an alist with proper pairs '(a . b) if you want b to be a single value.

See also: `dict->alist`, `dict`, `dict->list`, `list->dict`. →index

### 3.7.2 `array->list`: procedure/1

Usage: (`array->list` `arr`)=> `li`

Convert array `arr` into a list.

See also: `list->array`, `array`. →index

### 3.7.3 `array->str` : procedure/1

Usage: `(array->str arr)=> s`

Convert an array of unicode glyphs as integer values into a string. If the given sequence is not a valid UTF-8 sequence, an error is thrown.

See also: `str->array`. [→index](#)

### 3.7.4 `ascii85->blob` : procedure/1

Usage: `(ascii85->blob str)=> blob`

Convert the ascii85 encoded string `str` to a binary blob. This will raise an error if `str` is not a valid ascii85 encoded string.

See also: `blob->ascii85`, `base64->blob`, `str->blob`, `hex->blob`. [→index](#)

### 3.7.5 `base64->blob` : procedure/1

Usage: `(base64->blob str)=> blob`

Convert the base64 encoded string `str` to a binary blob. This will raise an error if `str` is not a valid base64 encoded string.

See also: `blob->base64`, `hex->blob`, `ascii85->blob`, `str->blob`. [→index](#)

### 3.7.6 `blob->ascii85` : procedure/1 or more

Usage: `(blob->ascii85 b [start] [end])=> str`

Convert the blob `b` to an ascii85 encoded string. If the optional `start` and `end` are provided, then only bytes from `start` (inclusive) to `end` (exclusive) are converted.

See also: `blob->hex`, `blob->str`, `blob->base64`, `valid?`, `blob?`. [→index](#)

### 3.7.7 `blob->base64` : procedure/1 or more

Usage: `(blob->base64 b [start] [end])=> str`

Convert the blob `b` to a base64 encoded string. If the optional `start` and `end` are provided, then only bytes from `start` (inclusive) to `end` (exclusive) are converted.

See also: `base64->blob`, `valid?`, `blob?`, `blob->str`, `blob->hex`, `blob->ascii85`. [→index](#)

### 3.7.8 `blob->hex` : procedure/1 or more

Usage: `(blob->hex b [start] [end])=> str`

Convert the blob `b` to a hexadecimal string of byte values. If the optional `start` and `end` are provided, then only bytes from `start` (inclusive) to `end` (exclusive) are converted.

See also: `hex->blob`, `str->blob`, `valid?`, `blob?`, `blob->base64`, `blob->ascii85`. [→index](#)

### 3.7.9 `blob->str` : procedure/1 or more

Usage: `(blob->str b [start] [end])=> str`

Convert blob `b` into a string. Notice that the string may contain binary data that is not suitable for displaying and does not represent valid UTF-8 glyphs. If the optional `start` and `end` are provided, then only bytes from `start` (inclusive) to `end` (exclusive) are converted.

See also: `str->blob`, `valid?`, `blob?`. [→index](#)

### 3.7.10 `char->str` : procedure/1

Usage: `(char->str n)=> str`

Return a string containing the unicode char based on integer `n`.

See also: `str->char`. [→index](#)

### 3.7.11 `chars->str` : procedure/1

Usage: `(chars->str a)=> str`

Convert an array of UTF-8 rune integers `a` into a UTF-8 encoded string.

See also: `str->runes`, `str->char`, `char->str`. [→index](#)

### 3.7.12 `dict->alist` : procedure/1

Usage: `(dict->alist d)=> li`

Convert a dictionary into an association list. Note that the resulting alist will be a set of proper pairs of the form `'(a . b)` if the values in the dictionary are not lists.

See also: `dict`, `dict-map`, `dict->list`. [→index](#)

### 3.7.13 `dict->array`: procedure/1

Usage: `(dict->array d)=> array`

Return an array that contains all key, value pairs of `d`. A key comes directly before its value, but otherwise the order is unspecified.

See also: `dict->list`, `dict.` [→index](#)

### 3.7.14 `dict->keys`: procedure/1

Usage: `(dict->keys d)=> li`

Return the keys of dictionary `d` in arbitrary order.

See also: `dict`, `dict->values`, `dict->alist`, `dict->list`. [→index](#)

### 3.7.15 `dict->list`: procedure/1

Usage: `(dict->list d)=> li`

Return a list of the form '(key1 value1 key2 value2 ...), where the order of key, value pairs is unspecified.

See also: `dict->array`, `dict.` [→index](#)

### 3.7.16 `dict->values`: procedure/1

Usage: `(dict->values d)=> li`

Return the values of dictionary `d` in arbitrary order.

See also: `dict`, `dict->keys`, `dict->alist`, `dict->list`. [→index](#)

### 3.7.17 `expr->str`: procedure/1

Usage: `(expr->str expr)=> str`

Convert a Lisp expression `expr` into a string. Does not use a stream port.

See also: `str->expr`, `str->expr*`, `openstr`, `internalize`, `externalize`. [→index](#)

**3.7.18 hex->blob : procedure/1**

Usage: (hex->blob str)=> blob

Convert hex string `str` to a blob. This will raise an error if `str` is not a valid hex string.

See also: `blob->hex`, `base64->blob`, `ascii85->blob`, `str->blob`. →index

**3.7.19 list->array : procedure/1**

Usage: (list->array li)=> array

Convert the list `li` to an array.

See also: `list`, `array`, `string`, `nth`, `seq?`. →index

**3.7.20 list->set : procedure/1**

Usage: (list->set li)=> dict

Create a dict containing true for each element of list `li`.

See also: `make-set`, `set-element?`, `set-union`, `set-intersection`, `set-complement`, `set-difference`, `set?`, `set-empty`. →index

**3.7.21 list->str : procedure/1**

Usage: (list->str li)=> string

Return the string that is composed out of the chars in list `li`.

See also: `array->str`, `str->list`, `chars`. →index

**3.7.22 set->list : procedure/1**

Usage: (set->list s)=> li

Convert set `s` to a list of set elements.

See also: `list->set`, `make-set`, `set-element?`, `set-union`, `set-intersection`, `set-complement`, `set-difference`, `set?`, `set-empty`. →index

**3.7.23 str->array : procedure/1**

Usage: (str->array s)=> array

Return the string *s* as an array of unicode glyph integer values.

See also: array->str. →index

**3.7.24 str->blob : procedure/1**

Usage: (str->blob s)=> blob

Convert string *s* into a blob.

See also: blob->str. →index

**3.7.25 str->char : procedure/1**

Usage: (str->char s)

Return the first character of *s* as unicode integer.

See also: char->str. →index

**3.7.26 str->chars : procedure/1**

Usage: (str->chars s)=> array

Convert the UTF-8 string *s* into an array of UTF-8 rune integers. An error may occur if the string is not a valid UTF-8 string.

See also: runes->str, str->char, char->str. →index

**3.7.27 str->expr : procedure/0 or more**

Usage: (str->expr s [default])=> any

Convert a string *s* into a Lisp expression. If **default** is provided, it is returned if an error occurs, otherwise an error is raised.

See also: expr->str, str->expr\*, openstr, externalize, internalize. →index

### 3.7.28 `str->expr*` : procedure/0 or more

Usage: `(str->expr* s [default])=> li`

Convert a string `s` into a list consisting of the Lisp expressions in `s`. If `default` is provided, then this value is put in the result list whenever an error occurs. Otherwise an error is raised. Notice that it might not always be obvious what expression in `s` triggers an error, since this hinges on the way the internal expression parser works.

See also: `str->expr`, `expr->str`, `openstr`, `internalize`, `externalize`. [→index](#)

### 3.7.29 `str->list` : procedure/1

Usage: `(str->list s)=> list`

Return the sequence of numeric chars that make up string `s`.

See also: `str->array`, `list->str`, `array->str`, `chars`. [→index](#)

### 3.7.30 `str->sym` : procedure/1

Usage: `(str->sym s)=> sym`

Convert a string into a symbol.

See also: `sym->str`, `intern`, `make-symbol`. [→index](#)

### 3.7.31 `sym->str` : procedure/1

Usage: `(sym->str sym)=> str`

Convert a symbol into a string.

See also: `str->sym`, `intern`, `make-symbol`. [→index](#)

## 3.8 Special Data Structures

This section lists some more specialized data structures and helper functions for them.

### 3.8.1 chars : procedure/1

Usage: (`chars str`)=> `dict`

Return a charset based on `str`, i.e., dict with the chars of `str` as keys and true as value.

See also: `dict`, `get`, `set`, `contains`. →index

### 3.8.2 dequeue! : macro/1 or more

Usage: (`dequeue! sym [def]`)=> `any`

Get the next element from queue `sym`, which must be the unquoted name of a variable, and return it. If a default `def` is given, then this is returned if the queue is empty, otherwise nil is returned.

See also: `make-queue`, `queue?`, `enqueue!`, `glance`, `queue-empty?`, `queue-len`. →index

### 3.8.3 enqueue! : macro/2

Usage: (`enqueue! sym elem`)

Put `elem` in queue `sym`, where `sym` is the unquoted name of a variable.

See also: `make-queue`, `queue?`, `dequeue!`, `glance`, `queue-empty?`, `queue-len`. →index

### 3.8.4 glance : procedure/1

Usage: (`glance s [def]`)=> `any`

Peek the next element in a stack or queue without changing the data structure. If default `def` is provided, this is returned in case the stack or queue is empty; otherwise nil is returned.

See also: `make-queue`, `make-stack`, `queue?`, `enqueue?`, `dequeue?`, `queue-len`, `stack-len`, `pop!`, `push!`. →index

### 3.8.5 inchars : procedure/2

Usage: (`inchars char chars`)=> `bool`

Return true if char is in the charset chars, nil otherwise.

See also: `chars`, `dict`, `get`, `set`, `has`. →index



### 3.8.6 `make-queue` : procedure/0

Usage: (`make-queue`)=> `array`

Make a synchronized queue.

See also: `queue?`, `enqueue!`, `dequeue!`, `glance`, `queue-empty?`, `queue-len`. →index

**Warning: Never change the array of a synchronized data structure directly, or your warranty is void!**

### 3.8.7 `make-set` : procedure/0 or more

Usage: (`make-set` [`arg1`] ... [`argn`])=> `dict`

Create a dictionary out of arguments `arg1` to `argn` that stores true for every argument.

See also: `list->set`, `set->list`, `set-element?`, `set-union`, `set-intersection`, `set-complement`, `set-difference`, `set?`, `set-empty?`. →index

### 3.8.8 `make-stack` : procedure/0

Usage: (`make-stack`)=> `array`

Make a synchronized stack.

See also: `stack?`, `push!`, `pop!`, `stack-empty?`, `stack-len`, `glance`. →index

**Warning: Never change the array of a synchronized data structure directly, or your warranty is void!**

### 3.8.9 `pop!` : macro/1 or more

Usage: (`pop!` `sym` [`def`])=> `any`

Get the next element from stack `sym`, which must be the unquoted name of a variable, and return it. If a default `def` is given, then this is returned if the queue is empty, otherwise nil is returned.

See also: `make-stack`, `stack?`, `push!`, `stack-len`, `stack-empty?`, `glance`. →index

### 3.8.10 push! : macro/2

Usage: (push! sym elem)

Put `elem` in stack `sym`, where `sym` is the unquoted name of a variable.

See also: `make-stack`, `stack?`, `pop!`, `stack-len`, `stack-empty?`, `glance`. →index

### 3.8.11 queue-empty? : procedure/1

Usage: (queue-empty? q) => bool

Return true if the queue `q` is empty, nil otherwise.

See also: `make-queue`, `queue?`, `enqueue!`, `dequeue!`, `glance`, `queue-len`. →index

### 3.8.12 queue-len : procedure/1

Usage: (queue-len q) => int

Return the length of the queue `q`.

See also: `make-queue`, `queue?`, `enqueue!`, `dequeue!`, `glance`, `queue-len`. →index

**Warning: Be advised that this is of limited use in some concurrent contexts, since the length of the queue might have changed already once you've obtained it!**

### 3.8.13 queue? : procedure/1

Usage: (queue? q) => bool

Return true if `q` is a queue, nil otherwise.

See also: `make-queue`, `enqueue!`, `dequeue`, `glance`, `queue-empty?`, `queue-len`. →index

### 3.8.14 set-complement : procedure/2

Usage: (set-complement a domain) => set

Return all elements in `domain` that are not elements of `a`.

See also: `list->set`, `set->list`, `make-set`, `set-element?`, `set-union`, `set-difference`, `set-intersection`, `set?`, `set-empty?`, `set-subset?`, `set-equal?`. →index

### 3.8.15 `set-difference` : procedure/2

Usage: `(set-difference a b)`=> `set`

Return the set-theoretic difference of set `a` minus set `b`, i.e., all elements in `a` that are not in `b`.

See also: `list->set`, `set->list`, `make-set`, `set-element?`, `set-union`, `set-intersection`, `set-complement`, `set?`, `set-empty?`, `set-subset?`, `set-equal?`. →index

### 3.8.16 `set-element?` : procedure/2

Usage: `(set-element? s elem)`=> `bool`

Return true if set `s` has element `elem`, nil otherwise.

See also: `make-set`, `list->set`, `set->list`, `set-union`, `set-intersection`, `set-complement`, `set-difference`, `set?`, `set-empty?`. →index

### 3.8.17 `set-empty?` : procedure/1

Usage: `(set-empty? s)`=> `bool`

Return true if set `s` is empty, nil otherwise.

See also: `make-set`, `list->set`, `set->list`, `set-union`, `set-intersection`, `set-complement`, `set-difference`, `set?`. →index

### 3.8.18 `set-equal?` : procedure/2

Usage: `(set-equal? a b)`=> `bool`

Return true if `a` and `b` contain the same elements.

See also: `set-subset?`, `list->set`, `set-element?`, `set->list`, `set-union`, `set-difference`, `set-intersection`, `set-complement`, `set?`, `set-empty?`. →index

### 3.8.19 `set-intersection` : procedure/2

Usage: `(set-intersection a b)`=> `set`

Return the intersection of sets `a` and `b`, i.e., the set of elements that are both in `a` and in `b`.

See also: `list->set`, `set->list`, `make-set`, `set-element?`, `set-union`, `set-complement`, `set-difference`, `set?`, `set-empty?`, `set-subset?`, `set-equal?`. →index

### 3.8.20 `set-subset?` : procedure/2

Usage: `(set-subset? a b)`=> `bool`

Return true if `a` is a subset of `b`, nil otherwise.

See also: `set-equal?`, `list->set`, `set->list`, `make-set`, `set-element?`, `set-union`, `set-difference`, `set-intersection`, `set-complement`, `set?`, `set-empty?`. →index

### 3.8.21 `set-union` : procedure/2

Usage: `(set-union a b)`=> `set`

Return the union of sets `a` and `b` containing all elements that are in `a` or in `b` (or both).

See also: `list->set`, `set->list`, `make-set`, `set-element?`, `set-intersection`, `set-complement`, `set-difference`, `set?`, `set-empty?`. →index

### 3.8.22 `set?` : procedure/1

Usage: `(set? x)`=> `bool`

Return true if `x` can be used as a set, nil otherwise.

See also: `list->set`, `make-set`, `set->list`, `set-element?`, `set-union`, `set-intersection`, `set-complement`, `set-difference`, `set-empty?`. →index

### 3.8.23 `stack-empty?` : procedure/1

Usage: `(queue-empty? s)`=> `bool`

Return true if the stack `s` is empty, nil otherwise.

See also: `make-stack`, `stack?`, `push!`, `pop!`, `stack-len`, `glance`. →index

### 3.8.24 `stack-len` : procedure/1

Usage: `(stack-len s)`=> `int`

Return the length of the stack `s`.

See also: `make-queue`, `queue?`, `enqueue!`, `dequeue!`, `glance`, `queue-len`. →index

**Warning:** Be advised that this is of limited use in some concurrent contexts, since the length of the queue might have changed already once you've obtained it!

### 3.8.25 `stack?` : procedure/1

Usage: `(stack? q) => bool`

Return true if `q` is a stack, nil otherwise.

See also: `make-stack`, `push!`, `pop!`, `stack-empty?`, `stack-len`, `glance`. →index

## 3.9 Databases

These functions allow for SQLite3 database access. The module needs to be enabled with the “db” build tag. It also provides access to key-value databases with prefix `'kvdb` and the automated remember-recall system, both of which are implemented in Z3S5 Lisp on top of the `'db` module. To use the remember system, it needs to be initialized first by calling `(init-remember)`.

### 3.9.1 `db.blob` : procedure/2

Usage: `(db.blob db-result n) => fl`

Get the content of column `n` in `db-result` as blob. A blob is a boxed memory area holding binary data.

See also: `db.str`. →index

### 3.9.2 `db.close` : procedure/1

Usage: `(db.close db)`

Close the database `db`.

See also: `db.open`, `db.open*`, `db.exec`, `db.query`. →index

### 3.9.3 `db.close-result` : procedure/1

Usage: `(db.close-result db-result)`

Close the `db-result`. It is invalid afterwards. This should be done to avoid memory leaks after the result has been used.

See also: `db.reset`, `db.step`, `db.close`. →index

### 3.9.4 `db.exec` : procedure/2 or more

Usage: `(db.exec db stmt [args] ...)`

Execute the SQL statement `stmt` in database `db`, binding any optional `args` to the open variable slots in it. This function does not return anything, use `db.query` to execute a query that returns rows as result.

See also: `db.query`, `db.open`, `db.close`, `db.open*`. →index

### 3.9.5 `db.float` : procedure/2

Usage: `(db.float db-result n)=> fl`

Get the content of column `n` in `db-result` as float.

See also: `db.int`, `db.str`. →index

### 3.9.6 `db.int` : procedure/2

Usage: `(db.int db-result n)=> int`

Get the content of column `n` in `db-result` as integer.

See also: `db.float`, `db.str`, `db.blob`. →index

### 3.9.7 `db.open` : procedure/1

Usage: `(db.open fi)=> db`

Opens an sqlite3 DB or creates a new, empty database at file path `fi`.

See also: `db.close`, `db.exec`, `db.query`. →index

### 3.9.8 `db.open*` : procedure/1

Usage: `(db.open* sel)=> db`

Open a temporary database if `sel` is 'temp' or an in-memory database if `sel` is 'mem'.

See also: `db.open`, `db.close`, `db.exec`, `db.query`. →index

### 3.9.9 `db.query` : procedure/2 or more

Usage: `(db.query db stmt [args] ...)`=> `db-result`

Query `db` with SQL statement `stmt`, binding any optional `args` to the open variable slots in it. This function returns a `db-result` that can be used to loop through rows with `db.step` and obtain columns in them using the various accessor methods.

See also: `db.exec`, `db.step`, `db.int`, `db.cname`, `db.float`, `db.str`, `db.expr`, `db.blob`. →index

### 3.9.10 `db.result-column-count` : procedure/1

Usage: `(db.result-column-count db-result)`=> `int`

Get the number of columns in the rows of `db-result`.

See also: `db.result-columns`. →index

### 3.9.11 `db.result-columns` : procedure/1

Usage: `(db.result-columns db-result)`=> `li`

Get a list of column specifications for `db-result`, each consisting of a list with the column name and the column type as string, as these were provided to the query. Since queries support automatic type conversions, this need not reflect the column types in the database schema.

See also: `db.result-column-count`. →index

### 3.9.12 `db.row` : procedure/1

Usage: `(db.row db-result)`=> `li`

Return all columns of the current row in `db-result` as list. They have the respective base types INT, FLOAT, BLOB, and TEXT.

See also: `db.rows`. →index

### 3.9.13 `db.step` : procedure/1

Usage: `(db.step db-result)`=> `bool`

Obtain the next result row in `db-result` and return true, or return nil if there is no more row in the result.

See also: `db.query`, `db.row`, `db.rows`. →index

### 3.9.14 `db.str`: procedure/2

Usage: `(db.str db-result n)=> str`

Get the content of column `n` in `db-result` as string.

See also: `db.blob`, `db.int`, `db.float`. →index

### 3.9.15 `forget`: procedure/1

Usage: `(forget key)`

Forget the value associated with `key`. This permanently deletes the value from the persistent record.

See also: `remember`, `recall`, `recollect`, `recall-when`, `recall-info`. →index

### 3.9.16 `init-remember`: procedure/0

Usage: `(init-remember)`

Initialize the remember database. This requires the modules 'kvdb and 'db enabled. The database is located at `(str+ (sysdir 'z3s5-data) "/remembered.z3kv")`.

See also: `remember`, `recall-when`, `recall`, `forget`. →index

### 3.9.17 `kvdb.begin`: procedure/1

Usage: `(kvdb.begin db)`

Begin a key-value database transaction. This can be committed by using `kvdb.commit` and rolled back by `kvdb.rollback`.

See also: `kvdb.comit`, `kvdb.rollback`. →index

**Warning: Transactions in key-value databases cannot be nested! You have to ensure that there is only one begin...commit pair.**



**3.9.18 kvdb.close : procedure/1**

Usage: (`kvdb.close db`)

Close a key-value db.

See also: `kvdb.open`. →index

**3.9.19 kvdb.commit : procedure/1**

Usage: (`kvdb.commit db`)

Commit the current transaction, making any changes made since the transaction started permanent.

See also: `kvdb.rollback`, `kvdb.begin`. →index

**3.9.20 kvdb.db? : procedure/1**

Usage: (`kvdb.db? datum`)=> `bool`

Return true if the given datum is a key-value database, nil otherwise.

See also: `kvdb.open`. →index

**3.9.21 kvdb.forget : procedure/1**

Usage: (`kvdb.forget key`)

Forget the value for `key` if there is one.

See also: `kvdb.set`, `kvdb.get`. →index

**3.9.22 kvdb.forget-everything : procedure/1**

Usage: (`kvdb.forget-everything db`)

Erases all data from the given key-value database `db`, irrecoverably losing ALL data in it.

See also: `kvdb.forget`. →index

**Warning: This operation cannot be undone! Data for all types of keys is deleted. Permanent data loss is imminent!**

### 3.9.23 kvdb.get : procedure/2 or more

Usage: (kvdb.get db key [other])=> any

Get the value stored at `key` in the key-value database `db`. If the value is found, it is returned. If the value is not found and `other` is specified, then `other` is returned. If the value is not found and `other` is not specified, then nil is returned.

See also: kvdb.set, kvdb.when, kvdb.info, kvdb.open, kvdb.forget, kvdb.close, kvdb.search, remember, recall, forget. →index

### 3.9.24 kvdb.info : procedure/2 or more

Usage: (db key [other])=> (str str)

Return a list containing the info string and its fuzzy variant stored for `key` in `db`, `other` when the value for `key` is not found. The default for `other` is nil.

See also: kvdb.get, kvdb.when. →index

### 3.9.25 kvdb.open : procedure/1 or more

Usage: (kvdb.open path)=> kvdb-array

Create or open a key-value database at `path`.

See also: kvdb.close. →index

### 3.9.26 kvdb.rollback : procedure/1

Usage: (kvdb.rollback db)

Rollback the changes made since the last transaction has been started and return the key-value database to its previous state.

See also: kvdb.commit, kvdb.begin. →index

### 3.9.27 kvdb.search : procedure/2 or more

Usage: (kvdb.search db s [keytype] [limit] [fuzzer])=> li

Search the key-value database `db` for search expression string `s` for optional `keytype` and return a list of matching keys. The optional `keytype` may be one of '(all str sym int expr), where the default is 'all for

any kind of key. If the optional `limit` is provided, then only `limit` entries are returned. Default limit is `kvdb.default-search-limit`. If `fuzzer` is a function provided, then a fuzzy string search is performed based on applying fuzzer to the search term; default is `nil`.

See also: `kvdb.get`. →index

### 3.9.28 `kvdb.set` : procedure/3 or more

Usage: `(kvdb.set db key value [info] [fuzzer])`

Set the `value` for `key` in key-value database `db`. The optional `info` string contains searchable information about the value that may be retrieved with the search function. The optional `fuzzer` must be a function that takes a string and yields a fuzzy variant of the string that can be used for fuzzy search. If no fuzzer is specified, then the default metaphone algorithm is used. Keys for the database must be externalizable but notice that integer keys may provide faster performance.

See also: `kvdb.get`, `kvdb.forget`, `kvdb.open`, `kvdb.close`, `kvdb.search`. →index

### 3.9.29 `kvdb.when` : procedure/2 or more

Usage: `(kvdb.when db key [other])=> str`

Get the date in `db` when the entry for `key` was last modified as a date string. If there is no entry for `key`, then `other` is returned. If `other` is not specified and there is no `key`, then `nil` is returned.

See also: `datestr->datelist`, `kvdb.get`, `kvdb.info`. →index

### 3.9.30 `recall` : procedure/1 or more

Usage: `(recall key [notfound])=> any`

Obtain the value remembered for `key`, `notfound` if it doesn't exist. If `notfound` is not provided, then `nil` is returned in case the value for `key` doesn't exist.

See also: `recall-when`, `recall-info`, `recollect`, `remember`, `forget`. →index

### 3.9.31 `recall-info` : procedure/1 or more

Usage: `(recall-info key [notfound])=> (str str)`

Return a list containing the info string and its fuzzy version for a remembered value with the given `key`, `notfound` if no value for `key` was found. The default for `notfound` is `nil`.

See also: `recall-when`, `recall`, `recall-when`, `recollect`, `remember`, `forget`. →index

### 3.9.32 `recall-when` : procedure/1 or more

Usage: (`recall-when` *key* [*notfound*])=> *datestr*

Obtain the date string when the value for *key* was last modified by `remember` (`set`), *notfound* if it doesn't exist. If *notfound* is not provided, then `nil` is returned in case there is no value for *key*.

See also: `recall`, `datestr->datelist`, `recall-info`, `remember`, `forget`. →index

### 3.9.33 `recollect` : procedure/1 or more

Usage: (`recollect` *s* [*keytype*] [*limit*] [*fuzzer*])=> *li*

Search for remembered items based on search query *s* and return a list of matching keys. The optional *keytype* parameter must be one of '(all str sym int expr), where the default is 'all for all kinds of keys. Up to *limit* results are returned, default is `kvdb.default-search-limit`. The optional *fuzzer* procedure takes a word string and yields a 'fuzzy' version of it. If *fuzzer* is specified and a procedure, then a fuzzy search is performed.

See also: `kvdb.search`, `recall`, `recall-info`, `recall-when`, `remember`. →index

### 3.9.34 `remember` : procedure/2

Usage: (`remember` *key* *value* [*info*] [*fuzzer*])

Persistently remember *value* by given *key*. See `kvdb.set` for the optional *info* and *fuzzer* arguments.

See also: `recall`, `forget`, `kvdb.set`, `recall-when`, `recall-info`, `recollect`. →index

## 3.10 Dictionaries

Dictionaries are thread-safe key-value repositories held in memory. They are internally based on hash tables and have fast access.

### 3.10.1 `delete` : procedure/2

Usage: (`delete` *d* *key*)

Remove the value for *key* in dict *d*. This also removes the key.

See also: `dict?`, `get`, `set`. →index

### 3.10.2 `dict` : procedure/0 or more

Usage: `(dict [li])=> dict`

Create a dictionary. The option `li` must be a list of the form '(key1 value1 key2 value2 ...). Dictionaries are unordered, hence also not sequences. Dictionaries are safe for concurrent access.

See also: `array`, `list`. →index

### 3.10.3 `dict-copy` : procedure/1

Usage: `(dict-copy d)=> dict`

Return a copy of dict `d`.

See also: `dict`, `dict?`. →index

### 3.10.4 `dict-empty?` : procedure/1

Usage: `(dict-empty? d)=> bool`

Return true if dict `d` is empty, nil otherwise. As crazy as this may sound, this can have  $O(n)$  complexity if the dict is not empty, but it is still going to be more efficient than any other method.

See also: `dict`. →index

### 3.10.5 `dict-foreach` : procedure/2

Usage: `(dict-foreach d proc)`

Call `proc` for side-effects with the key and value for each key, value pair in dict `d`.

See also: `dict-map!`, `dict?`, `dict`. →index

### 3.10.6 `dict-map` : procedure/2

Usage: `(dict-map dict proc)=> dict`

Returns a copy of `dict` with `proc` applies to each key value pair as arguments. Keys are immutable, so `proc` must take two arguments and return the new value.

See also: `dict-map!`, `map`. →index

### 3.10.7 dict-map! : procedure/2

Usage: (dict-map! d proc)

Apply procedure `proc` which takes the key and value as arguments to each key, value pair in dict `d` and set the respective value in `d` to the result of `proc`. Keys are not changed.

See also: `dict`, `dict?`, `dict-foreach`. →index

### 3.10.8 dict-merge : procedure/2

Usage: (dict-merge a b)=> dict

Create a new dict that contains all key-value pairs from dicts `a` and `b`. Note that this function is not symmetric. If a key is in both `a` and `b`, then the key value pair in `a` is retained for this key.

See also: `dict`, `dict-map`, `dict-map!`, `dict-foreach`. →index

### 3.10.9 dict? : procedure/1

Usage: (dict? obj)=> bool

Return true if `obj` is a dict, nil otherwise.

See also: `dict`. →index

### 3.10.10 get : procedure/2 or more

Usage: (get dict key [default])=> any

Get the value for `key` in `dict`, return `default` if there is no value for `key`. If `default` is omitted, then nil is returned. Provide your own default if you want to store nil.

See also: `dict`, `dict?`, `set`. →index

### 3.10.11 get-or-set : procedure/3

Usage: (get-or-set d key value)

Get the value for `key` in dict `d` if it already exists, otherwise set it to `value`.

See also: `dict?`, `get`, `set`. →index

### 3.10.12 **getstacked** : procedure/3

Usage: (`getstacked dict key default`)

Get the topmost element from the stack stored at `key` in `dict`. If the stack is empty or no stack is stored at `key`, then `default` is returned.

See also: `pushstacked`, `popstacked`. →index

### 3.10.13 **has** : procedure/2

Usage: (`has dict key`)=> `bool`

Return true if the dict `dict` contains an entry for `key`, nil otherwise.

See also: `dict`, `get`, `set`. →index

### 3.10.14 **has-key?** : procedure/2

Usage: (`has-key? d key`)=> `bool`

Return true if `d` has key `key`, nil otherwise.

See also: `dict?`, `get`, `set`, `delete`. →index

### 3.10.15 **popstacked** : procedure/3

Usage: (`popstacked dict key default`)

Get the topmost element from the stack stored at `key` in `dict` and remove it from the stack. If the stack is empty or no stack is stored at `key`, then `default` is returned.

See also: `pushstacked`, `getstacked`. →index

### 3.10.16 **pushstacked** : procedure/3

Usage: (`pushstacked dict key datum`)

Push `datum` onto the stack maintained under `key` in the `dict`.

See also: `getstacked`, `popstacked`. →index

### 3.10.17 `set` : procedure/3

Usage: (`set` `d` `key` `value`)

Set `value` for `key` in dict `d`.

See also: `dict`, `get`, `get-or-set`. →index

### 3.10.18 `set*` : procedure/2

Usage: (`set*` `d` `li`)

Set in dict `d` the keys and values in list `li`. The list `li` must be of the form (key-1 value-1 key-2 value-2 ... key-n value-n). This function may be slightly faster than using individual `set` operations.

See also: `dict`, `set`. →index

## 3.11 File Input & Output

These functions allow direct access for reading and writing to files. This module requires the `fileio` build tag.

### 3.11.1 `close` : procedure/1

Usage: (`close` `p`)

Close the port `p`. Calling close twice on the same port should be avoided.

See also: `open`, `stropen`. →index

### 3.11.2 `dir` : procedure/1

Usage: (`dir` [`path`])=> `li`

Obtain a directory list for `path`. If `path` is not specified, the current working directory is listed.

See also: `dir?`, `open`, `close`, `read`, `write`. →index



### 3.11.3 `dir?` : procedure/1

Usage: (`dir?` `path`)=> `bool`

Check if the file at `path` is a directory and return true, nil if the file does not exist or is not a directory.

See also: `file-exists?`, `dir`, `open`, `close`, `read`, `write`. →index

### 3.11.4 `fdelete` : procedure/1

Usage: (`fdelete` `path`)

Removes the file or directory at `path`.

See also: `file-exists?`, `dir?`, `dir`. →index

**Warning: This function also deletes directories containing files and all of their subdirectories!**

### 3.11.5 `file-port?` : procedure/1

Usage: (`file-port?` `p`)=> `bool`

Return true if `p` is a file port, nil otherwise.

See also: `port?`, `str-port?`, `open`, `stropen`. →index

### 3.11.6 `open` : procedure/1 or more

Usage: (`open` `file-path` [`modes`] [`permissions`])=> `int`

Open the file at `file-path` for reading and writing, and return the stream ID. The optional `modes` argument must be a list containing one of '(read write read-write) for read, write, or read-write access respectively, and may contain any of the following symbols: 'append to append to an existing file, 'create for creating the file if it doesn't exist, 'exclusive for exclusive file access, 'truncate for truncating the file if it exists, and 'sync for attempting to sync file access. The optional `permissions` argument must be a numeric value specifying the Unix file permissions of the file. If these are omitted, then default values '(read-write append create) and 0640 are used.

See also: `stropen`, `close`, `read`, `write`. →index

### 3.11.7 read : procedure/1

Usage: (`read p`)=> `any`

Read an expression from input port `p`.

See also: `input`, `write`. →index

### 3.11.8 read-binary : procedure/3

Usage: (`read-binary p buff n`)=> `int`

Read `n` or less bytes from input port `p` into binary blob `buff`. If `buff` is smaller than `n`, then an error is raised. If less than `n` bytes are available before the end of file is reached, then the amount `k` of bytes is read into `buff` and `k` is returned. If the end of file is reached and no byte has been read, then 0 is returned. So to loop through this, read into the buffer and do something with it while the amount of bytes returned is larger than 0.

See also: `write-binary`, `read`, `close`, `open`. →index

### 3.11.9 read-string : procedure/2

Usage: (`read-string p delstr`)=> `str`

Reads a string from port `p` until the single-byte delimiter character in `delstr` is encountered, and returns the string including the delimiter. If the input ends before the delimiter is encountered, it returns the string up until EOF. Notice that if the empty string is returned then the end of file must have been encountered, since otherwise the string would contain the delimiter.

See also: `read`, `read-binary`, `write-string`, `write`, `read`, `close`, `open`. →index

### 3.11.10 str-port? : procedure/1

Usage: (`str-port? p`)=> `bool`

Return true if `p` is a string port, nil otherwise.

See also: `port?`, `file-port?`, `stropen`, `open`. →index

### 3.11.11 write : procedure/2

Usage: (`write p datum`)=> `int`

Write `datum` to output port `p` and return the number of bytes written.

See also: `write-binary`, `write-binary-at`, `read`, `close`, `open`. →index

### 3.11.12 `write-binary`: procedure/4

Usage: (`write-binary` `p` `buff` `n` `offset`)=> `int`

Write `n` bytes starting at `offset` in binary blob `buff` to the stream port `p`. This function returns the number of bytes actually written.

See also: `write-binary-at`, `read-binary`, `write`, `close`, `open`. →index

### 3.11.13 `write-binary-at`: procedure/5

Usage: (`write-binary-at` `p` `buff` `n` `offset` `fpos`)=> `int`

Write `n` bytes starting at `offset` in binary blob `buff` to the seekable stream port `p` at the stream position `fpos`. If there is not enough data in `p` to overwrite at position `fpos`, then an error is caused and only part of the data might be written. The function returns the number of bytes actually written.

See also: `read-binary`, `write-binary`, `write`, `close`, `open`. →index

### 3.11.14 `write-string`: procedure/2

Usage: (`write-string` `p` `s`)=> `int`

Write string `s` to output port `p` and return the number of bytes written. LF are *not* automatically converted to CR LF sequences on windows.

See also: `write`, `write-binary`, `write-binary-at`, `read`, `close`, `open`. →index

## 3.12 Floating Point Arithmetics Package

The package `fl` provides floating point arithmetics functions. They require the given number not to exceed a value that can be held by a 64 bit float in the range 2.2E-308 to 1.7E+308.

### 3.12.1 `fl.abs`: procedure/1

Usage: (`fl.abs` `x`)=> `fl`

Return the absolute value of `x`.

See also: `float`, `*`. →index

### 3.12.2 `fl.acos`: procedure/1

Usage: `(fl.acos x)=> fl`

Return the arc cosine of `x`.

See also: `fl.cos`. →index

### 3.12.3 `fl.asin`: procedure/1

Usage: `(fl.asin x)=> fl`

Return the arc sine of `x`.

See also: `fl.acos`. →index

### 3.12.4 `fl.asinh`: procedure/1

Usage: `(fl.asinh x)=> fl`

Return the inverse hyperbolic sine of `x`.

See also: `fl.cosh`. →index

### 3.12.5 `fl.atan`: procedure/1

Usage: `(fl.atan x)=> fl`

Return the arctangent of `x` in radians.

See also: `fl.atanh`, `fl.tan`. →index

### 3.12.6 `fl.atan2`: procedure/2

Usage: `(fl.atan2 x y)=> fl`

Atan2 returns the arc tangent of `y / x`, using the signs of the two to determine the quadrant of the return value.

See also: `fl.atan`. →index

**3.12.7 fl.atanh : procedure/1**

Usage: (fl.atanh x)=> fl

Return the inverse hyperbolic tangent of *x*.

See also: fl.atan. →index

**3.12.8 fl.cbrt : procedure/1**

Usage: (fl.cbrt x)=> fl

Return the cube root of *x*.

See also: fl.sqrt. →index

**3.12.9 fl.ceil : procedure/1**

Usage: (fl.ceil x)=> fl

Round *x* up to the nearest integer, return it as a floating point number.

See also: fl.floor, truncate, int, fl.round, fl.trunc. →index

**3.12.10 fl.cos : procedure/1**

Usage: (fl.cos x)=> fl

Return the cosine of *x*.

See also: fl.sin. →index

**3.12.11 fl.cosh : procedure/1**

Usage: (fl.cosh x)=> fl

Return the hyperbolic cosine of *x*.

See also: fl.cos. →index

**3.12.12 fl.dim: procedure/2**

Usage: (fl.dim x y)=> fl

Return the maximum of x, y or 0.

See also: [max](#). →index

**3.12.13 fl.erf: procedure/1**

Usage: (fl.erf x)=> fl

Return the result of the error function of x.

See also: [fl.erfc](#), [fl.dim](#). →index

**3.12.14 fl.erfc: procedure/1**

Usage: (fl.erfc x)=> fl

Return the result of the complementary error function of x.

See also: [fl.erfcinv](#), [fl.erf](#). →index

**3.12.15 fl.erfcinv: procedure/1**

Usage: (fl.erfcinv x)=> fl

Return the inverse of (fl.erfc x).

See also: [fl.erfc](#). →index

**3.12.16 fl.erfinv: procedure/1**

Usage: (fl.erfinv x)=> fl

Return the inverse of (fl.erf x).

See also: [fl.erf](#). →index

**3.12.17 fl.exp : procedure/1**

Usage: (fl.exp *x*)=> fl

Return  $e^x$ , the base-e exponential of *x*.

See also: fl.exp. →index

**3.12.18 fl.exp2 : procedure/2**

Usage: (fl.exp2 *x*)=> fl

Return  $2^x$ , the base-2 exponential of *x*.

See also: fl.exp. →index

**3.12.19 fl.expm1 : procedure/1**

Usage: (fl.expm1 *x*)=> fl

Return  $e^x - 1$ , the base-e exponential of (sub1 *x*). This is more accurate than (sub1 (fl.exp *x*)) when *x* is very small.

See also: fl.exp. →index

**3.12.20 fl.floor : procedure/1**

Usage: (fl.floor *x*)=> fl

Return *x* rounded to the nearest integer below as floating point number.

See also: fl.ceil, truncate, int. →index

**3.12.21 fl.fma : procedure/3**

Usage: (fl.fma *x y z*)=> fl

Return the fused multiply-add of *x*, *y*, *z*, which is  $x * y + z$ .

See also: \*, +. →index

**3.12.22 fl.frexp : procedure/1**

Usage: (fl.frexp x) => li

Break  $x$  into a normalized fraction and an integral power of two. It returns a list of (frac exp) containing a float and an integer satisfying  $x == \text{frac} \times 2^{\text{exp}}$  where the absolute value of  $\text{frac}$  is in the interval  $[0.5, 1)$ .

See also: fl.exp. →index

**3.12.23 fl.gamma : procedure/1**

Usage: (fl.gamma x) => fl

Compute the Gamma function of  $x$ .

See also: fl.lgamma. →index

**3.12.24 fl.hypot : procedure/2**

Usage: (fl.hypot x y) => fl

Compute the square root of  $x^2$  and  $y^2$ .

See also: fl.sqrt. →index

**3.12.25 fl.ilogb : procedure/1**

Usage: (fl.ilogb x) => fl

Return the binary exponent of  $x$  as a floating point number.

See also: fl.exp2. →index

**3.12.26 fl.inf : procedure/1**

Usage: (fl.inf x) => fl

Return positive 64 bit floating point infinity +INF if  $x \geq 0$  and negative 64 bit floating point infinity -INF if  $x < 0$ .

See also: fl.is-nan?. →index



**3.12.27 fl.is-nan? : procedure/1**

Usage: (fl.is-nan? x) => bool

Return true if *x* is not a number according to IEEE 754 floating point arithmetics, nil otherwise.

See also: fl.inf. →index

**3.12.28 fl.j0 : procedure/1**

Usage: (fl.j0 x) => fl

Apply the order-zero Bessel function of the first kind to *x*.

See also: fl.j1, fl.jn, fl.y0, fl.y1, fl.yn. →index

**3.12.29 fl.j1 : procedure/1**

Usage: (fl.j1 x) => fl

Apply the the order-one Bessel function of the first kind *x*.

See also: fl.j0, fl.jn, fl.y0, fl.y1, fl.yn. →index

**3.12.30 fl.jn : procedure/1**

Usage: (fl.jn n x) => fl

Apply the Bessel function of order *n* to *x*. The number *n* must be an integer.

See also: fl.j1, fl.j0, fl.y0, fl.y1, fl.yn. →index

**3.12.31 fl.ldexp : procedure/2**

Usage: (fl.ldexp x n) => fl

Return the inverse of fl.frexp,  $x * 2^n$ .

See also: fl.frexp. →index

**3.12.32 fl.lgamma : procedure/1**

Usage: (fl.lgamma x) => li

Return a list containing the natural logarithm and sign (-1 or +1) of the Gamma function applied to  $x$ .

See also: fl.gamma. →index

**3.12.33 fl.log : procedure/1**

Usage: (fl.log x) => fl

Return the natural logarithm of  $x$ .

See also: fl.log10, fl.log2, fl.logb, fl.log1p. →index

**3.12.34 fl.log10 : procedure/1**

Usage: (fl.log10 x) => fl

Return the decimal logarithm of  $x$ .

See also: fl.log, fl.log2, fl.logb, fl.log1p. →index

**3.12.35 fl.log1p : procedure/1**

Usage: (fl.log1p x) => fl

Return the natural logarithm of  $x + 1$ . This function is more accurate than (fl.log (add1 x)) if  $x$  is close to 0.

See also: fl.log, fl.log2, fl.logb, fl.log10. →index

**3.12.36 fl.log2 : procedure/1**

Usage: (fl.log2 x) => fl

Return the binary logarithm of  $x$ . This is important for calculating entropy, for example.

See also: fl.log, fl.log10, fl.log1p, fl.logb. →index

**3.12.37 fl.logb : procedure/1**

Usage: (fl.logb *x*)=> fl

Return the binary exponent of *x*.

See also: fl.log, fl.log10, fl.log1p, fl.logb, fl.log2. →index

**3.12.38 fl.max : procedure/2**

Usage: (fl.max *x y*)=> fl

Return the larger value of two floating point arguments *x* and *y*.

See also: fl.min, max, min. →index

**3.12.39 fl.min : procedure/2**

Usage: (fl.min *x y*)=> fl

Return the smaller value of two floating point arguments *x* and *y*.

See also: fl.min, max, min. →index

**3.12.40 fl.mod : procedure/2**

Usage: (fl.mod *x y*)=> fl

Return the floating point remainder of *x* / *y*.

See also: fl.reminder. →index

**3.12.41 fl.modf : procedure/1**

Usage: (fl.modf *x*)=> li

Return integer and fractional floating-point numbers that sum to *x*. Both values have the same sign as *x*.

See also: fl.mod. →index

**3.12.42 fl.nan : procedure/1**

Usage: (fl.nan)=> fl

Return the IEEE 754 not-a-number value.

See also: fl.is-nan?, fl.inf. →index

**3.12.43 fl.next-after : procedure/1**

Usage: (fl.next-after x)=> fl

Return the next representable floating point number after *x*.

See also: fl.is-nan?, fl.nan, fl.inf. →index

**3.12.44 fl.pow : procedure/2**

Usage: (fl.pow x y)=> fl

Return *x* to the power of *y* according to 64 bit floating point arithmetics.

See also: fl.pow10. →index

**3.12.45 fl.pow10 : procedure/1**

Usage: (fl.pow10 n)=> fl

Return 10 to the power of integer *n* as a 64 bit floating point number.

See also: fl.pow. →index

**3.12.46 fl.reminder : procedure/2**

Usage: (fl.reminder x y)=> fl

Return the IEEE 754 floating-point remainder of *x* / *y*.

See also: fl.mod. →index

**3.12.47 fl.round : procedure/1**

Usage: (fl.round *x*)=> fl

Round *x* to the nearest integer floating point number according to floating point arithmetics.

See also: fl.round-to-even, fl.truncate, **int**, **float**. →index

**3.12.48 fl.round-to-even : procedure/1**

Usage: (fl.round-to-even *x*)=> fl

Round *x* to the nearest even integer floating point number according to floating point arithmetics.

See also: fl.round, fl.truncate, **int**, **float**. →index

**3.12.49 fl.signbit : procedure/1**

Usage: (fl.signbit *x*)=> bool

Return true if *x* is negative, nil otherwise.

See also: fl.abs. →index

**3.12.50 fl.sin : procedure/1**

Usage: (fl.sin *x*)=> fl

Return the sine of *x*.

See also: fl.cos. →index

**3.12.51 fl.sinh : procedure/1**

Usage: (fl.sinh *x*)=> fl

Return the hyperbolic sine of *x*.

See also: fl.sin. →index

**3.12.52 fl.sqrt : procedure/1**

Usage: (fl.sqrt x) => fl

Return the square root of *x*.

See also: fl.pow. →index

**3.12.53 fl.tan : procedure/1**

Usage: (fl.tan x) => fl

Return the tangent of *x* in radian.

See also: fl.tanh, fl.sin, fl.cos. →index

**3.12.54 fl.tanh : procedure/1**

Usage: (fl.tanh x) => fl

Return the hyperbolic tangent of *x*.

See also: fl.tan, flsinh, fl.cosh. →index

**3.12.55 fl.trunc : procedure/1**

Usage: (fl.trunc x) => fl

Return the integer value of *x* as floating point number.

See also: truncate, int, fl.floor. →index

**3.12.56 fl.y0 : procedure/1**

Usage: (fl.y0 x) => fl

Return the order-zero Bessel function of the second kind applied to *x*.

See also: fl.y1, fl.yn, fl.j0, fl.j1, fl.jn. →index

### 3.12.57 `fl.y1`: procedure/1

Usage: `(fl.y1 x) => fl`

Return the order-one Bessel function of the second kind applied to `x`.

See also: `fl.y0`, `fl.yn`, `fl.j0`, `fl.j1`, `fl.jn`. →index

### 3.12.58 `fl.yn`: procedure/1

Usage: `(fl.yn n x) => fl`

Return the Bessel function of the second kind of order `n` applied to `x`. Argument `n` must be an integer value.

See also: `fl.y0`, `fl.y1`, `fl.j0`, `fl.j1`, `fl.jn`. →index

## 3.13 Graphical User Interface

This section describes the GUI-related functions. These functions are only available when Z3S5 Lisp has been compiled with the embedded GUI package. See the `z3g` executable defined in `cmd/z3g/z3g.go` for an example of how to include the GUI and start it up. The key is that the interpreter must run in its own goroutine because the GUI is blocking once it has been called. GUI functions are threadsafe, or at least as threadsafe as the underlying GUI framework. Functions defined in Lisp are derived from corresponding functions of the `Fyne` framework and listed under the 'gui' label in this help system. The naming conventions for translation between Go and Lisp functions are as follows:

1. Camelcase is translated to lowercase with hyphens.
2. A function `object.VerbQualifier` becomes `verb-object-qualifier`.
3. Getters are written in the form `get-object-qualifier` and setters `set-object-qualifier`.
4. As an exception of the previous rules, when the result of a function is a bool, the form is `object-predicate?`.

Fyne objects are represented by integer numbers. The system internally translates between these numbers and objects. Occasionally, Fyne objects are created on the fly for performance reasons. For example, sometimes color lists of the form `(r g b a)` with integers `r`, `g`, `b`, `a` are used instead of creating and storing color objects using `(nrgba r g b a)`. There are also sometimes shortcut accessors using selector symbols and other convenience wrappers for Fyne functions. When in doubt, refer to the Lisp help for details.

When importing the GUI with `DefGUI`, a `Config` structure is provided that allows for restricted security. This makes it possible to use the GUI functions in a restricted environment that e.g. does not allow the creation of new windows.

### 3.13.1 `activate-menu-last-submenu` : procedure/1

Usage: (`activate-menu-last-submenu` menu)=> `bool`

Find the last active menu item traversing through open submenus, and activate its submenu if one is found. Return true if a submenu was activated, nil otherwise.

See also: `deactivate-menu-last-submenu`, `new-menu`, `activate-menu-next`, `activate-menu-previous`. →index

### 3.13.2 `activate-menu-next` : procedure/1

Usage: (`activate-menu-next` menu)

Activate the menu item following the currently active menu item, if there is any.

See also: `activate-menu-previous`, `new-menu`, `activate-menu-last-submenu`. →index

### 3.13.3 `activate-menu-previous` : procedure/1

Usage: (`activate-menu-previous` menu)

Activate the menu item before the currently active menu item, if there is any.

See also: `activate-menu-next`, `new-menu`, `activate-menu-last-submenu`. →index

### 3.13.4 `append-form` : procedure/0

Usage: (`append-form` form str canvas-object)

Append a new row to the bottom `form` consisting of a label `str` aligned with a `canvas-object`, which may be an entry, button, etc.

See also: `new-form`. →index



### 3.13.5 `center-window-on-screen` : procedure/1

Usage: (`center-window-on-screen` `window`)

As the name implies, this function centers the window on the screen.

See also: `set-window-full-screen`, `window-full-screen?`. →index

### 3.13.6 `close-gui` : procedure/0

Usage: (`close-gui`)

Close the GUI, freeing all resources associated with it. After this function has been called, no further GUI functions can be used.

See also: `forget-gui-object`, `close-window`. →index

### 3.13.7 `close-window` : procedure/1

Usage: (`close-window` `window`)

Closes `window` and delete it from internal storage. This frees window resources. It cannot be re-used after this operation. Use `window-hide` if you want to close/hide a window only temporarily. Notice that unlike in Fyne, there is no mechanism to close an application automatically after its main window has been closed.

See also: `hide-window`. →index

### 3.13.8 `deactivate-menu-child` : procedure/1

Usage: (`deactivate-menu-child` `menu`)

Deactivate the currently active menu item and close its submenu if there is one.

See also: `activate-menu-last-submenu`, `activate-menu-next`, `activate-menu-previous`, `new-menu`. →index

### 3.13.9 `deactivate-menu-last-submenu` : procedure/1

Usage: (`deactivate-menu-last-submenu` `menu`)

Traverse the menu and deactivate the last open submenu found.

See also: `activate-menu-last-submenu`, `activate-menu-next`, `activate-menu-previous`, `new-menu`. →index

### 3.13.10 `disable-object`: procedure/1

Usage: (`disable-object` `obj`)

Disable the canvas object `obj`.

See also: `enable-object`, `hide-object`, `show-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index

### 3.13.11 `enable-object`: procedure/1

Usage: (`enable-object` `obj`)

Enable the canvas object `obj`.

See also: `disable-object`, `hide-object`, `show-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index

### 3.13.12 `entry-accepts-tab?`: procedure/1

Usage: (`entry-accepts-tab?` `entry`)=> `bool`

Return `nil` when the entry does not accept tabs, `t` otherwise.

See also: `new-entry`. →index

### 3.13.13 `forget-gui-object`: procedure/1

Usage: (`forget-gui-object` `int`)

Forget the GUI object `int`. This removes any association with the object but does not free internal resources if the object still exists. Internal use only.

See also: `close-window`, `close-gui`. →index

### 3.13.14 `get-clipboard-content` : procedure/0

Usage: `(get-clipboard-content)`=> `str`

Return the current content of the operating system clipboard as string. This function might raise an error if clipboard access is prohibited by host security settings.

See also: `set-clipboard-content`. →index

### 3.13.15 `get-device-info` : procedure/0

Usage: `(get-device-info)`=> `li`

Return a list with information about the current host device. This returns an association list where 'orientation might be one of '(vertical vertical-upside-down left right unknown), self-explanatory boolean keys 'is-mobile?, 'is-browser, 'has-keyboard?, and 'system-scale with the current scaling factor for graphics as float. The system scale is used to dynamically scale user interface elements to remain legible on hi res displays.

See also: `close-gui`. →index

### 3.13.16 `get-entry-cursor` : procedure/1

Usage: `(get-entry-cursor entry)`=> `sym`

Return a symbol that represents the current cursor of `entry`. Possible values are in '(default text crosshair pointer hresize vresize). Curiously, there is no way to set the cursor yet.

See also: `new-entry`. →index

### 3.13.17 `get-entry-cursor-pos` : procedure/1

Usage: `(get-entry-cursor-pos entry)`=> `li`

Return a list consisting of row number and column number of the current cursor position of the cursor in `entry`.

See also: `set-entry-cursor-row`, `set-entry-cursor-column`. →index

### 3.13.18 `get-menu-item-label` : procedure/1

Usage: `(get-menu-item-label item)`=> `str`

Return the current label of the given menu *item*.

See also: `set-menu-item-label`, `set-menu-item-disabled`, `menu-item-disabled`, `set-menu-item-checked`, `menu-item-checked?`, `new-menu*`, `new-menu`, `new-menu-item`, `new-menu-item-separator`. →index

### 3.13.19 `get-object-min-size`: procedure/1

Usage: `(get-object-min-size obj)=> li`

Return the minimum size of canvas object *obj* as a list containing the width and height as floats. The minimum size is computed based on various internal criteria and can only be changed for some special widgets.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index

### 3.13.20 `get-object-position`: procedure/1

Usage: `(get-object-position obj)=> li`

Return the position of canvas object *obj* as a list containing the x and y coordinates as floats.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index

### 3.13.21 `get-object-size`: procedure/1

Usage: `(get-object-size obj)=> li`

Return the size of canvas object *obj* as a list containing the width and height as floats.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index

### 3.13.22 `get-progress-bar-value`: procedure/1

Usage: `(get-progress-bar-value bar)=> num`

Return the current value of progress-bar `bar`.

See also: `set-progress-bar`, `new-progress-bar`, `new-infinite-progress-bar`. →index

### 3.13.23 `get-text-grid-cell: procedure/3`

Usage: `(get-text-grid-cell grid row column)=> li`

Return the cell of `grid` at `row` and `column`. The result is a list consisting of a string containing one unicode rune and a grid style list.

See also: `set-text-grid-cell`, `get-text-grid-row`, `set-text-grid-rune`, `set-text-grid-style-range`. →index

### 3.13.24 `get-text-grid-row: procedure/2`

Usage: `(get-text-grid-row grid row)=> li`

Obtain a `row` of a text `grid`, where `row` is a 0-based index. This function returns a list of the form `'(row style)`, where `style` is a grid style list and `row` is an array of lists consisting each of a unicode string containing one rune and a grid style list. Each entry of the `row` array represents an individual unicode glyph with a style, whereas the `style` list in the return argument represents an optional style of the whole row.

See also: `set-text-grid-row`, `get-text-grid-row-text`, `get-text-grid-cell`, `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `set-text-grid-cell`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index

### 3.13.25 `get-text-grid-row-text: procedure/2`

Usage: `(get-text-grid-row-text grid row)=> str`

Return the text of `row` in `grid` as a string without any style information.

See also: `set-text-grid-rune`, `get-text-grid-row`, `get-text-grid-cell`, `set-text-grid-row`, `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`,

`get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `set-text-grid-cell`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index

### 3.13.26 `get-text-grid-tab-width`: procedure/1

Usage: (`get-text-grid-tab-width` *grid*)=> **int**

Return the current tabulator width of *grid* in space characters.

See also: `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index

### 3.13.27 `get-text-grid-text`: procedure/1

Usage: (`get-text-grid-text` *grid*)=> *str*

Return the text of *grid* as a string without style information.

See also: `set-text-grid-text`, `new-text-grid`, `get-text-grid-row`, `get-text-grid-rune`, `get-text-grid-cell`. →index

### 3.13.28 `get-window-canvas`: procedure/1

Usage: (`get-window-canvas` *window*)=> **int**

Get the canvas object of *window*, which is the area on which window elements are drawn. This is not the same as the `window-content`, which is a widget or other user interface element. The canvas is used for raw-drawing commands, for example for drawing circles and boxes. With a suitable layout that doesn't re-arrange objects, it can e.g. be used to draw overlays.

See also: `get-window-content`, `set-window-content`. →index

### 3.13.29 `get-window-content`: procedure/2

Usage: (`get-window-content` *window*)=> **int**

Get the canvas object ID that represents the main content of the window. This is usually a widget or a container with layout.

See also: `set-window-content`, `get-window-canvas`. →index

### 3.13.30 `get-window-icon`: procedure/1

Usage: (`get-window-icon` `window`)=> `int`

Obtain the icon ID of the icon of `window`. The resource obtained is not guaranteed to be a visible icon or might be a dummy, as not all windows have icons on all platforms.

See also: `set-window-icon`, `new-icon`, `theme-icon`. →index

### 3.13.31 `get-window-main-menu`: procedure/2

Usage: (`get-window-main-menu` `window`)=> `int`

Get the main menu ID of `window`.

See also: `set-window-main-menu`, `new-main-menu`. →index

### 3.13.32 `get-window-title`: procedure/1

Usage: (`get-window-title` `window`)=> `str`

Return the title of `window` as string.

See also: `set-window-title`. →index

### 3.13.33 `hide-object`: procedure/1

Usage: (`hide-object` `obj`)

Hide the canvas object `obj`.

See also: `disable-object`, `enable-object`, `show-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index

### 3.13.34 `hide-window` : procedure/1

Usage: (`hide-window` `window`)

Hides `window`. It can be shown again using `show-window`.

See also: `show-window`, `close-window`. →index

### 3.13.35 `menu-item-checked?` : procedure/1

Usage: (`menu-item-checked?` `item`)=> `bool`

Return true if `item` is currently checked, nil otherwise.

See also: `set-menu-item-checked`, `set-menu-item-disabled`, `menu-item-disabled?`, `get-menu-item-label`, `set-menu-item-label`, `new-menu*`, `new-menu`, `new-menu-item`, `new-menu-item-separator`. →index

### 3.13.36 `menu-item-disabled?` : procedure/1

Usage: (`menu-item-disabled?` `item`)=> `bool`

Return true if `item` is currently disabled, nil otherwise.

See also: `set-menu-item-disabled`, `set-menu-item-checked`, `menu-item-checked?`, `get-menu-item-label`, `set-menu-item-label`, `new-menu*`, `new-menu`, `new-menu-item`, `new-menu-item-separator`. →index

### 3.13.37 `move-object` : procedure/2

Usage: (`move-object` `obj` `position`)

Move the canvas object `obj` to the given `position` list, containing its `x` and `y` coordinates as floats.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `object-disabled?`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index

### 3.13.38 `new-app-tabs` : procedure/0 or more

Usage: (`new-app-tabs` `tab-item` ...)=> `int`

Create a new application tabs, which allow users to choose different items within an application.



See also: [new-doc-tabs](#), [new-tabitem](#), [new-tabitem-with-icon](#). [→index](#)

### 3.13.39 new-border : procedure/4 or more

Usage: ([new-border](#) *top bottom left right* [*obj ...*])=> [int](#)

Create a new border layout, which is one of the most useful layouts. Any of [top](#), [bottom](#), [left](#), and [right](#) is put in the respective place (with minimum size) and might also be [nil](#) for no widget. The remaining canvas objects [obj](#) are arranged in the center and take maximum size. This allows you e.g. to put a list on the left side of a window, a panel of buttons on the top, and the main content in another container in the center.

See also: [new-container](#), [new-container-without-layout](#). [→index](#)

### 3.13.40 new-button : procedure/2

Usage: ([new-button](#) *label* *proc*)=> [int](#)

Return a new button with the given [label](#) and without an icon. The callback [proc](#) is called without arguments when the button is pressed or tapped.

See also: [new-button-with-icon](#), [new-hyperlink](#), [new-label](#). [→index](#)

### 3.13.41 new-button-with-icon : procedure/3

Usage: ([new-button-with-icon](#) *label* *icon* *proc*)=> [int](#)

Return a new button the given [label](#) and [icon](#). The callback [proc](#) is called without arguments when the button is pressed.

See also: [new-button](#), [new-icon](#), [theme-icon](#). [→index](#)

### 3.13.42 new-center-layout : procedure/0

Usage: ([new-center-layout](#))=> [int](#)

Create a new center layout, which centers container elements (possibly overlapping). This may be used for drawing centered on the window, for example.

See also: [new-form](#), [append-form](#), [new-spacer](#), [new-hbox-layout](#), [new-vbox-layout](#), [new-grid-layout](#), [new-grid-wrap-layout](#), [new-form-layout](#), [new-stack-layout](#), [new-container](#). [→index](#)

### 3.13.43 `new-check` : procedure/2

Usage: (`new-check` `title` `proc`)=> `int`

Create and return a new check box with the given `title` string and a callback procedure `proc`. The callback `proc` is called with the new state of the check box as bool when it has changed.

See also: `new-choice`. →index

### 3.13.44 `new-choice` : procedure/3

Usage: (`new-choice` `selector` `string-list` `proc`)=> `int`

Create and return a new choice representing choices in `string-list`. If `selector` is 'radio-group, a group of radio buttons is created with options in `string-list`. If `selector` is 'select, a more compact selection menu is created with the options in `string-list`. The callback `proc` takes a string that represents the choice that has been selected.

See also: `new-check`. →index

### 3.13.45 `new-circle` : procedure/1 or more

Usage: (`new-circle` `fill-color` [`pos1`] [`pos2`] [`stroke-color`] [`stroke-width`])=> `int`

Draw and return a circle with the given NRGBA `fill-color`. If the optional `pos1` and `pos2` position lists of `x` and `y` coordinates in floats are given, then the circle is drawn inside the rectangle defined by these positions. The optional `stroke-color` and `stroke-width` arguments determine the outline of the circle. Notice that circle's size and position may be set by the layout of the container, so to set these manually using `pos1` and `pos2` you need to make sure the underlying container has no such layout.

See also: `new-rectangle`, `new-line` `new-text`. →index

### 3.13.46 `new-combined-string-validator` : procedure/1 or more

Usage: (`new-combined-string-validator` `validator-1` [...`validator-n`])=> `int`

Combine validators `validator-1` to `validator-n` into a combined string validator and return it.

See also: `set-entry-validator`, `new-validator`, `new-regexp-validator`, `new-time-validator`, `set-object-on-validation-change-callback`, `validate-object`. →index

**3.13.47 new-container : procedure/1 or more**

Usage: (`new-container` `layout` `obj` ...) => `int`

Create a new container with the given `layout` and various canvas objects `obj` arranged by the layout.

See also: `new-container-without-layout`, `new-border`. →index

**3.13.48 new-container-without-layout : procedure/0 or more**

Usage: (`new-container-without-layout` `obj` ...) => `int`

Create a new container without a layout (overlapping objects) with the given canvas objects `obj`.

See also: `new-container`, `new-border`. →index

**3.13.49 new-doc-tabs : procedure/0 or more**

Usage: (`new-doc-tabs` `tab-item` ...) => `int`

Create new document tabs, which allow users to choose different items in a window (not the application as a whole like `app-tabs`).

See also: `new-app-tabs`, `new-tabitem`, `new-tabitem-with-icon`. →index

**3.13.50 new-entry : procedure/1**

Usage: (`new-entry` [`selector`]) => `int`

Create a new text entry field based on the optional `selector` symbol. `selector` can be a symbol in '(single-line multi-line password). The default is 'single-line.

See also: `set-entry-on-change-callback`, `set-entry-validator`, `entry-accepts-tab`?, `get-entry-cursor-pos`, `set-entry-cursor-row`, `set-entry-cursor-column`, `set-entry-on-cursor-change-callback`, `get-entry-cursor`, `get-entry-selected-text`, `set-entry-min-rows-visible`, `set-entry-place-holder`, `set-entry-text`. →index

**3.13.51 new-form : procedure/0**

Usage: (`new-form`)

Return a new form container, which orders widgets in rows, where each row has a label and a widget whose columns are aligned with the other rows. Use `append-form` to add label and widgets.

See also: [append-form](#). [→index](#)

### 3.13.52 `new-form-layout`: procedure/0

Usage: (`new-form-layout`)=> `int`

Create a form layout, which arranges elements in two columns per row, where the columns are aligned.

See also: [new-form](#), [append-form](#), [new-spacer](#), [new-hbox-layout](#), [new-vbox-layout](#), [new-grid-layout](#), [new-grid-wrap-layout](#), [new-center-layout](#), [new-stack-layout](#), [new-container](#). [→index](#)

### 3.13.53 `new-grid-layout`: procedure/1

Usage: (`new-grid-layout` `n`)=> `int`

Create a new grid layout, which arranges elements in `n` columns.

See also: [new-spacer](#), [new-hbox-layout](#), [new-vbox-layout](#), [new-grid-wrap-layout](#), [new-form-layout](#), [new-center-layout](#), [new-stack-layout](#), [new-container](#). [→index](#)

### 3.13.54 `new-grid-wrap-layout`: procedure/2

Usage: (`new-grid-wrap-layout` `width` `height`)=> `int`

Create a new grid wrap layout, which arranges elements such that each element has the given `width` and `height`, and wraps lines based on the size of the parent container.

See also: [new-spacer](#), [new-hbox-layout](#), [new-vbox-layout](#), [new-grid-layout](#), [new-form-layout](#), [new-center-layout](#), [new-stack-layout](#), [new-container](#). [→index](#)

### 3.13.55 `new-hbox-layout`: procedure/0

Usage: (`new-hbox-layout`)=> `int`

Create a new horizontal box layout, which lays out container elements horizontally.

See also: [new-spacer](#), [new-vbox-layout](#), [new-grid-layout](#), [new-grid-wrap-layout](#), [new-form-layout](#), [new-center-layout](#), [new-stack-layout](#), [new-container](#). [→index](#)

### 3.13.56 `new-hyperlink`: procedure/2

Usage: (`new-hyperlink` `label` `url`)=> `int`

Create a new hyperlink with given `label` string and an `url` string as embedded link. A hyperlink looks like a label with link style; when it is clicked, the `url` is called by the default operating system mechanism for opening urls. Using hyperlinks might be disallowed by the host system configuration `HyperlinksAllowed` and may be re-written by the host system using the `CheckHyperlinks` function. If `HyperlinksAllowed` is false in the active GUI config of the host, this function raises an error. It also parses the given `url` and will raise an error if it does not represent a valid URL.

See also: `new-button`, `new-label`. →index

**Warning: Allowing the host to open hyperlinks usually launches a web browser and the details depend on the operating system. There is an added security risk!**

### 3.13.57 `new-icon`: procedure/1

Usage: (`new-icon` `resource`)=> `int`

Create a new icon from `resource`, which must be suitable to create an image.

See also: `theme-icon`. →index

### 3.13.58 `new-image-from-file`: procedure/1

Usage: (`new-image-from-file` `path`)=> `int`

Create and return a new image from the image file at `path`, which must be a PNG file.

See also: `new-image-from-resource`. →index

### 3.13.59 `new-image-from-resource`: procedure/1

Usage: (`new-image-from-resource` `resource`)=> `int`

Create and return a new image from the given `resource`.

See also: `new-image-from-file`, `theme-icon`. →index

### 3.13.60 `new-label` : procedure/1

Usage: (`new-label` `str`)=> `int`

Creates a new text label with string `str`.

See also: `set-label-text`. →index

### 3.13.61 `new-line` : procedure/1 or more

Usage: (`new-line` `fill-color` [`pos1`] [`pos2`] [`stroke-color`] [`stroke-width`])=> `int`

Draw and return a line with the given NRGBA `fill-color` from optional position `pos1` to position `pos2`, where these are lists of `x` and `y` coordinates as floats. The optional `stroke-color` and `stroke-width` determines the outer edges of the line.

See also: `new-circle`, `new-rectangle`, `new-text`. →index

### 3.13.62 `new-list` : procedure/3

Usage: (`new-list` `len-proc` `prep-proc` `update-proc`)=> `int`

Create a new list display. A list consists of rows of simple items like labels. The `len-proc` must be a procedure without arguments returning the length of the list as integer. The `prep-proc` must be a procedure without arguments that returns a canvas object (i.e. a label or other widgets) representing a template for a single list item. The `update-proc` must be a procedure that receives the ID of a canvas object (given by the template) and the 0-based list index as arguments. This procedure then should modify the canvas object with `ID` to display the given list item at the index. See the GUI examples on how to use this function.

See also: `new-table`, `new-tree`. →index

### 3.13.63 `new-main-menu` : procedure/1 or more

Usage: (`new-main-menu` `menu` ...)=> `int`

Return a new main menu with the given menus. A main menu displays a menubar for a window on some desktop platforms but it may also be displayed in other ways.

See also: `new-menu`, `new-menu*`. →index

### 3.13.64 `new-menu` : procedure/1

Usage: (`new-menu` `menu*`)=> `int`

Create a new visible menu widget from the abstract `menu*` created by `new-menu*`.

See also: `new-menu*`, `new-main-menu`. →index

### 3.13.65 `new-menu*` : procedure/1 or more

Usage: (`new-menu*` `label` [`item...`])=> `int`

Make a new abstract menu with given `label` and arbitrary menu items `item...` following. The starred function is used to define a menu but is not bound to any particular way of displaying it (popup-menu, normal menu, main menu). Use `new-menu` and `new-main-menu` to create visible menus and menu bars based on such abstract menus.

See also: `refresh-menu*`, `new-menu`, `new-main-menu`. →index

### 3.13.66 `new-menu-item` : procedure/2 or more

Usage: (`new-menu-item` `str` `proc` [`selector...`])=> `int`

Create a new menu item with given label `str` and callback `proc`, which takes no arguments. The optional `selector` symbol may be one of: 'is-quit - the item is the application Quit menu item (this is dealt with differently by operating system requirements), 'is-separator - the item is a menu item separator and the label string is ignored (redundant, use `new-menu-item-separator` instead), 'disabled - the menu item is disabled, or 'checked - the menu item is checked.

See also: `set-menu-item-checked`, `menu-item-checked?`, `set-menu-item-disabled`, `menu-item-disabled?`, `get-menu-item-label`, `set-menu-item-label`, `new-menu*`, `new-menu`, `new-menu-item-separator`. →index

### 3.13.67 `new-menu-item-separator` : procedure/0

Usage: (`new-menu-item-separator`)=> `int`

Return a new menu item separator, which is a menu item without callback and label that displays a separator between menu items in menus.

See also: `new-menu-item`. →index

### 3.13.68 `new-progress-bar` : procedure/0

Usage: (`new-progress-bar`)=> `int`

Create a new progress bar whose default minimum is 0.0 and maximum is 1.0.

See also: `set-progress-bar`, `new-infinite-progress-bar`, `get-progress-bar-value`.  
→index

### 3.13.69 `new-raster-with-pixels` : procedure/1

Usage: (`new-raster-with-pixels pixel-proc`)=> `int`

Create a new raster image generated dynamically by the given `pixel-proc`. The `pixel-proc` takes `x` and `y` pixel coordinates and the `width` and `height` of the image in pixels, and returns the color of the pixel `x, y` as a color list of the form '(red green blue [alpha]) where `alpha` is optional. Notice that specifying the color of each pixel can be very CPU-intensive for larger images, so optimizations might be necessary.

See also: `new-image-from-file`. →index

### 3.13.70 `new-rectangle` : procedure/1 or more

Usage: (`new-rectangle fill-color` [`width height`] [`position`] [`stroke-color`] [`stroke-width`] [`corner-radius`])=> `int`

Draw and return a rectangle with the given NRGBA `fill-color`. The optional int `width` and `height` arguments set the width and height of the rectangle explicitly (otherwise they are 1). The optional `position` argument must be a list of `x` and `y` coordinates as floats. The optional `stroke-color` and `stroke-width` arguments determine the color and width of the outline of the rectangle, and the optional `corner-radius` defines how rounded the rectangle is. Notice that the rectangle's size and position can be set by the layout of the container, so to set it manually you need to make sure the underlying container has no layout that positions or resizes the rectangle.

See also: `new-circle`, `new-line`, `new-text`. →index

### 3.13.71 `new-regexp-validator` : procedure/2

Usage: (`new-regexp-validator regexp reason`)=> `int`



Create a new string validator from the `regexp` string, which must be a valid regular expression in Go's regexp syntax. The `reason` string is displayed to the user by widgets like `entry` when the validation fails.

See also: `set-entry-validator`, `new-validator`, `new-combined-string-validator`, `new-time-validator`, `set-object-on-validation-change-callback`, `validate-object`.  
→index

### 3.13.72 `new-shortcut` : procedure/1 or more

Usage: (`new-shortcut` `key` [`mod...`])=> `int`

Create a new keyboard shortcut based on symbol or string `key` and a number of modifier keys `mod`. If multiple non-modifier keys are present, only the last one is taken. However, multiple modifier keys are possible. Possible modifiers are symbols or corresponding strings in '(shift control alt suprt). Possible keys are in '(escape return tab backspace insert delete right left down up page-up page-down home end f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 enter 0 1 2 3 4 5 6 7 8 9 key0 key1 key2 key3 key4 key5 key6 key7 key8 key9 a b c d e f g h i j k l m n o p q r s t u v w x y z space tick comma minus period slash backslash left-bracket right-bracket semicolon equal asterisk plus back-tick) and their string variants.

See also: `new-window`. →index

### 3.13.73 `new-slider` : procedure/3

Usage: (`new-slider` `min` `max` `proc`)=> `int`

Create a new slider that allows users to adjust numerical values. The `min` and `max` arguments must be floats. The procedure `proc` takes the current slider float value as argument and is called when the slider changes.

See also: `set-slider-value`. →index

### 3.13.74 `new-spacer` : procedure/0

Usage: (`new-spacer`)=> `int`

Create a new spacer, which adjusts size dynamically by taking up space and displaying nothing. Use this to fill containers e.g. to right align a widget.

See also: `new-hbox-layout`, `new-vbox-layout`, `new-grid-layout`, `new-grid-wrap-layout`, `new-form-layout`, `new-center-layout`, `new-stack-layout`, `new-container`. →index

### 3.13.75 `new-stack-layout` : procedure/0

Usage: (`new-stack-layout`)=> `int`

Create a new stack layout that stacks container elements on top of each other, overlapping. This may be used for drawing, for example.

See also: `new-form`, `append-form`, `new-spacer`, `new-hbox-layout`, `new-vbox-layout`, `new-grid-layout`, `new-grid-wrap-layout`, `new-form-layout`, `new-center-layout`, `new-container`. →index

### 3.13.76 `new-tabitem` : procedure/2

Usage: (`new-tabitem` `title` `obj`)=> `int`

Create a new tab item for use in app-tabs and doc-tabs with a `title` and an embedded canvas object `obj` shown when the tab item is selected in the tabs.

See also: `new-tabitem-with-icon`, `new-app-tabs`, `new-doc-tabs`. →index

### 3.13.77 `new-tabitem-with-icon` : procedure/3

Usage: (`new-tabitem-with-icon` `title` `icon` `obj`)=> `int`

Create a new tab item for use in app-tabs and doc-tabs with given `title` string, `icon` resource, and embedded canvas object `obj` that shown when the tab item is selected in the tabs.

See also: `new-tabitem`, `new-app-tabs`, `new-doc-tabs`. →index

### 3.13.78 `new-table` : procedure/3

Usage: (`new-table` `len-proc` `prep-proc` `update-proc`)=> `int`

Create a new table display. A table consists of a number of rows, each of which has a fixed number of columns such as labels. The `len-proc` must be a procedure without arguments returning the length of the table as integer. The `prep-proc` must be a procedure without arguments that returns a canvas object that represents the table row with updatable columns. The `update-proc` takes the row, column, and ID of a canvas object and updates a table template with the right display for the table cell at `row` and `column`.

See also: `new-list`, `new-tree`. →index

### 3.13.79 `new-text`: procedure/2

Usage: (`new-text` *str* *color*)=> **int**

Draw and return text with the given string *str* and foreground NRGBA *color*.

See also: `set-text-alignment`, `set-text-size`, `set-text-style`, `new-line`, `new-circle`, `new-rectangle`. →index

### 3.13.80 `new-text-grid`: procedure/0 or more

Usage: (`new-text-grid` [*<string>*] [*show-line-numbers* | *show-whitespace* | *tab-width* *<int>*])=> **int**

Create a new text grid widget, which displays multiline text with custom background and foreground colors. The optional *string* argument is the initial text of the grid without formatting. The following symbols might be '`show-line-numbers`' to turn the line number display on and '`show-whitespace`' to display white space characters by special unicode symbols. If the selector '`tab-width`' occurs, then it must be immediately followed by an integer for the tabulator width of the text grid in space characters.

See also: `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index

### 3.13.81 `new-time-validator`: procedure/1

Usage: (`new-time-validator` *format-str*)=> **int**

Create a new string validator for time and date based on the given template *format-str*. This validator uses Go's data parsing function and therefore is quite restrictive. Only datetimes entered in exactly the format given (including timezones) validate successfully. To obtain a more relaxed date and time validator, use `new-validator` to create a custom validator with your own parsing or try a `new-regexp-validator`.

See also: `set-entry-validator`, `new-validator`, `new-combined-string-validator`, `new-time-validator`, `new-regexp-validator`, `set-object-on-validation-change-callback`, `validate-object`. →index

### 3.13.82 `new-tree` : procedure/4

Usage: (`new-tree` `child-uid-proc` `is-branch-proc` `create-node-proc` `update-node-proc`)=> `int`

Create a new tree display. A tree displays nested branches and leaf nodes. The `child-uid-proc` is a procedure that takes an id string as argument. If the string is empty, it should return a list of top-level branch uid strings. If the string is not empty, it represents an uid; the procedure should then return a list of all child id strings of that branch. This defines the tree's structure. All id strings must be unique to the tree. The `is-branch-proc` takes an id string as argument and should return non-nil if the id represents a branch, nil if it has no children. The `create-node-proc` takes a bool `b` as argument and should return a branch template if `b` is non-nil and a leaf template object if `b` is nil. Finally, the `update-node-proc` is a procedure that takes a node id string, a boolean that is true if the node is a branch, and a node template canvas-object as it is returned by `create-node-proc`. The procedure should fill the template with the display values for the respective node id.

See also: `new-list`, `new-table`. →index

### 3.13.83 `new-validator` : procedure/1

Usage: (`new-validator` `proc`)=> `int`

Create a new string validator based on validation procedure `proc`. The procedure `proc` takes a string as argument and returns a string. If the string returned is not the empty string "", then validation fails and the returned string is given as a reason for validation failure. If the empty string is returned, then validation succeeds. If an error occurs in `proc`, then validation fails with the error's error message as reason. Notice that validators are fairly limited and can only be attached to a few validatable objects such as text entry fields. For a more general approach, it might make sense to implement your own validation system based on key press, focus change, and change callbacks of various GUI objects.

See also: `set-entry-validator`, `new-combined-string-validator`, `new-regexp-validator`, `new-time-validator`, `set-object-on-validation-change-callback`, `validate-object`. →index

### 3.13.84 `new-vbox-layout` : procedure/0

Usage: (`new-vbox-layout`)=> `int`

Create a new vertical box layout, which lays out container elements vertically.

See also: `new-spacer`, `new-hbox-layout`, `new-grid-layout`, `new-grid-wrap-layout`, `new-form-layout`, `new-center-layout`, `new-stack-layout`, `new-container`. →index

### 3.13.85 `new-window` : procedure/1

Usage: (`new-window` `title`)=> `int`

Create a new window with `title` string and return the window ID. This function raises an error if the host configuration `WindowsAllowed` is not true. In certain embedded uses, creating new windows is not allowed and you should check the documentation how to find a pre-configured window and add user interface elements to it.

See also: `set-window-content`, `close-window`, `show-window`. →index

### 3.13.86 `nrgba` : procedure/4

Usage: (`nrgba` `red` `green` `blue` `alpha`)=> `int`

Create an RGBA color where `red`, `green`, `blue`, and `alpha` are 8-bit uint integers, i.e., values between 0 and 255 (inclusive). Notice that some GUI functions require NRGBA color returned by this function, whereas others require a color list of int values '(red green blue alpha). This is for performance reasons, since it sometimes faster to convert a list to a color on-the-fly and sometimes more convenient to store pre-defined colors for later re-use.

See also: `new-rectangle`, `new-circle`, `new-line`, `new-text`. →index

### 3.13.87 `object-disabled?` : procedure/1

Usage: (`object-disabled?` `obj`)=> `bool`

Return true if the canvas object `obj` is disabled, nil otherwise.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index

### 3.13.88 `refresh-main-menu` : procedure/1

Usage: (`refresh-main-menu` `main-menu`)

Refresh the given `main-menu` display. This should be called after some submenus or menu items in the main menu have changed.

See also: `new-main-menu`, `refresh-menu*`. →index

**3.13.89 refresh-menu\* : procedure/1**

Usage: ([refresh-menu\\*](#) [menu](#))

Refresh the given [menu](#) after a change was made that has a visual impact. This will refresh the menu widget in which this abstract menu occurs.

See also: [refresh-main-menu](#), [new-menu\\*](#). →index

**3.13.90 refresh-object : procedure/1**

Usage: ([refresh-object](#) [obj](#))

Refresh the canvas object [obj](#), causing the graphical display to be re-drawn as soon as possible. This may be needed if the object's state has changed.

See also: [disable-object](#), [enable-object](#), [show-object](#), [hide-object](#), [object-disabled?](#), [move-object](#), [resize-object](#), [get-object-size](#), [get-object-min-size](#), [object-visible?](#), [get-object-position](#), [new-entry](#), [new-label](#). →index

**3.13.91 resize-object : procedure/2**

Usage: ([resize-object](#) [obj](#) [width](#) [height](#))

Resize canvas object [obj](#) to the given [width](#) and [height](#) as floats.

See also: [disable-object](#), [enable-object](#), [show-object](#), [hide-object](#), [object-disabled?](#), [move-object](#), [get-object-size](#), [get-object-min-size](#), [get-object-position](#), [object-visible?](#), [refresh-object](#), [new-entry](#), [new-label](#). →index

**3.13.92 set-clipboard-content : procedure/1**

Usage: ([set-clipboard-content](#) [str](#))

Set the operating system clipboard content to string [str](#). This function might raise an error if clipboard access is prohibited by host security settings.

See also: [get-clipboard-content](#). →index

**3.13.93 set-entry-cursor-column : procedure/2**

Usage: ([set-entry-cursor-column](#) [entry](#) [column](#))

Set the column position of the cursor in `entry` to integer `column`.

See also: `get-entry-cursor-pos`, `set-entry-cursor-row`. →index

### 3.13.94 `set-entry-cursor-row`: procedure/2

Usage: (`set-entry-cursor-row` `entry` `row`)

Set the row position of the cursor in `entry` to integer `row`.

See also: `get-entry-cursor-pos`, `set-entry-cursor-column`. →index

### 3.13.95 `set-entry-min-rows-visible`: procedure/2

Usage: (`set-entry-min-rows-visible` `entry` `rows`)

Set the minimum number of rows of `entry` that are visible. This ensures that `rows` text rows are visible and is a way of setting the entry's minimum size. Curiously, there is no corresponding `set-entry-min-columns-visible` function yet.

See also: `new-entry`. →index

### 3.13.96 `set-entry-on-change-callback`: procedure/2

Usage: (`set-entry-on-change-callback` `entry` `proc`)

Set the callback of `entry` that is triggered when the entry text changes. `proc` must be a procedure that takes the entry text as string.

See also: `new-entry`, `set-entry-cursor-change-callback`. →index

### 3.13.97 `set-entry-on-cursor-change-callback`: procedure/2

Usage: (`set-entry-cursor-change-callback` `entry` `proc`)

Set the cursor change callback of `entry` to `proc`, which is a procedure that takes the entry ID as argument.

See also: `new-entry`, `set-entry-on-change-callback`. →index

### 3.13.98 `set-entry-place-holder` : procedure/2

Usage: (`set-entry-place-holder` `entry` `str`)

Set the place holder string of `entry` to `str`. This is displayed as a prompt when no text is entered.

See also: `new-entry`, `set-entry-text`. →index

### 3.13.99 `set-entry-text` : procedure/2

Usage: (`set-entry-text` `entry` `str`)

Set the text of `entry` to string `str`.

See also: `new-entry`, `set-entry-place-holder`. →index

### 3.13.100 `set-entry-validator` : procedure/2

Usage: (`set-entry-validator` `entry` `validator`)

Set the `validator` of `entry`. A validator must be created first from a special procedure or a regular expression.

See also: `new-entry`, `new-validator`, `new-combined-string-validator`, `new-time-validator`, `new-regexp-validator`, `validate-object`. →index

### 3.13.101 `set-label-text` : procedure/2

Usage: (`set-label-text` `label` `str`)

Sets the text of `label` to string `str`. This might resize the label depending on the layout in which the label is put.

See also: `new-label`. →index

### 3.13.102 `set-menu-item-checked` : procedure/2

Usage: (`set-menu-item-checked` `item` `checked?`)

Set the menu item check mark display if `checked?` is non-nil, remove it otherwise.

See also: `menu-item-checked?`, `set-menu-item-disabled`, `menu-item-disabled?`, `get-menu-item-label`, `set-menu-item-label`, `new-menu*`, `new-menu`, `new-menu-item`, `new-menu-item-separator`. →index



### 3.13.103 `set-menu-item-disabled`: procedure/2

Usage: (`set-menu-item-disabled` `item` `disabled?`)

Disable the menu item if `disabled?` is non-nil, enable it otherwise.

See also: `menu-item-disabled?`, `set-menu-item-checked`, `menu-item-checked?`, `get-menu-item-label`, `set-menu-item-label`, `new-menu*`, `new-menu`, `new-menu-item`, `new-menu-item-separator`. →index

### 3.13.104 `set-menu-item-label`: procedure/2

Usage: (`set-menu-item-label` `item` `str`)

Set the label of menu `item` to `str`.

See also: `get-menu-item-label`, `set-menu-item-disabled`, `menu-item-disabled`, `set-menu-item-checked`, `menu-item-checked?`, `new-menu*`, `new-menu`, `new-menu-item`, `new-menu-item-separator`. →index

### 3.13.105 `set-object-on-validation-change-callback`: procedure/2

Usage: (`set-object-on-validation-change-callback` `obj` `proc`)

Set a validatable object's `obj` validation change callback, which is called when the object's validation changes. The callback `proc` takes a string or nil as argument. When it is nil, the validation was successful. When it is a string, then the validation failed with the string as reason. This can be used to track validation changes of any validatable object (such as a text entry) to e.g. display custom messages or icons when the validation fails or succeeds.

See also: `validate-object`, `new-validator`, `set-entry-validator`. →index

### 3.13.106 `set-progress-bar`: procedure/1 or more

Usage: (`set-progress-bar` `bar` `value` [`selector` `value`])

Set the value of progress-bar `bar` as follows. If a single number is provided, then the current value of the progress-bar is set to this number. If a selector symbol is provided, then if it is 'value, the progress-bar value is set to the following number, if it is 'max or 'min, then the progress-bar maximum or minimum values are set to the respective following number. If it is 'formatter, then the following value must be a procedure that takes the progress-bar ID as argument and returns a string that represents the display of the progress-bar at the given time.

See also: `get-progress-bar-value`, `new-progress-bar`, `new-infinite-progress-bar`.  
→index

### 3.13.107 `set-slider-value: procedure/2`

Usage: (`set-slider-value` `slider` `fl`)

Set the value of `slider` to float `fl`.

See also: `new-slider`. →index

### 3.13.108 `set-text-alignment: procedure/2`

Usage: (`set-text-alignment` `text` `sym`)

Set the alignment of `text` to `sym`, which must be one of '(leading center trailing).

See also: `new-text`, `set-text-size`, `set-text-style`. →index

### 3.13.109 `set-text-grid-cell: procedure/4`

Usage: (`set-text-grid-cell` `grid` `row` `column` `li`)

Set the text `grid` cell at `row` and `column` (both 0-indexed) to the list `li`, where `li` must consist of a unicode string containing one rune and a valid grid style list.

See also: `get-text-grid-cell`, `set-text-grid-rune`, `get-text-grid-row`, `set-text-grid-row`. →index

### 3.13.110 `set-text-grid-row: procedure/3`

Usage: (`set-text-grid-row` `grid` `row` `row-spec`)

Set the `row` of `grid` to the given `row-spec`, which is a list containing an array of grid cells like in the return value of `get-text-grid-row` and a grid style for the row as a whole.

See also: `get-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-cell`, `set-text-grid-rune`, `set-text-grid-style-range`. →index

**3.13.111 set-text-grid-row-style : procedure/3**

Usage: (set-text-grid-row-style grid row style)

Set the style of text `grid` at `row` to the given grid `style`.

See also: `set-text-grid-row`, `set-text-grid-cell`, `get-text-grid-row`, `set-text-grid-rune`, `set-text-grid-style-range`. →index

**3.13.112 set-text-grid-rune : procedure/4**

Usage: (set-text-grid-rune grid row column str)

Set the rune of `grid` at `row` and `column` to the unicode glyph in string `str`.

See also: `set-text-grid-style`, `set-text-grid-cell`, `get-text-grid-cell`. →index

**3.13.113 set-text-grid-show-line-numbers : procedure/2**

Usage: (set-text-grid-show-line-numbers grid show?)

Set whether `grid` shows line numbers. If `show?` is not nil, then line numbers are shown, otherwise they are not shown.

See also: `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index

**3.13.114 set-text-grid-show-whitespace : procedure/2**

Usage: (set-text-grid-show-whitespace grid show?)

Set whether `grid` shows whitespace characters. If `show?` is not nil, then whitespace characters are shown, otherwise they are not shown.

See also: `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`,

`set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index

### 3.13.115 `set-text-grid-style:procedure/4`

Usage: (`set-text-grid-style` `grid` `row` `column` `style`)

Set the grid style of `grid` at `row` and `column` to the list `style`.

See also: `set-text-grid-cell`, `set-text-grid-rune`, `set-text-grid-style-range`.  
→index

### 3.13.116 `set-text-grid-style-range:procedure/6`

Usage: (`set-text-grid-style-range` `grid` `start-row` `start-column` `end-row` `end-column` `style`)

Set the grid style of `grid` starting at `start-row` and `start-column` and ending at `end-row` and `end-column` (all inclusive) to the grid `style`.

See also: `set-text-grid-style`, `set-text-grid-cell`, `set-text-grid-row-style`.  
→index

### 3.13.117 `set-text-grid-tab-width:procedure/2`

Usage: (`set-text-grid-tab-width` `grid` `width`)

Set the tabulator width of `grid` to integer `width` space characters.

See also: `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index

### 3.13.118 `set-text-grid-text:procedure/2`

Usage: (`set-text-grid-text` `grid` `str`)

Set the text of the text `grid` to the given `str`.

See also: `get-text-grid-text`, `new-text-grid`, `set-text-grid-rune`, `set-text-grid-row`.  
→index

### 3.13.119 `set-text-size`: procedure/2

Usage: (`set-text-size` *text* *size*)

Set the size of *text* to float *size*.

See also: `new-text`, `set-text-alignment`, `set-text-style`. →index

### 3.13.120 `set-text-style`: procedure/2

Usage: (`set-text-style` *text* *li*)

Set the style of *text* to the specification in list *li*, which must contain symbols in '(bold italic monospace symbol tab-width). If a symbol in the list is `tab-width`, it must be followed by an integer. `bold` sets boldface, `italic` makes the style italic, `monospace` selects the monospace/typewriter font, and `symbol` selects the `symbol` font. `tab-width` followed by an integer sets the width of tabulator in terms of the number of space characters.

See also: `new-text`, `set-text-alignment`, `set-text-size`. →index

### 3.13.121 `set-window-content`: procedure/2

Usage: (`set-window-content` *window* *canvas-object*)

Set the main content of the window, which must be an existing canvas object such as a widget or container with layout.

See also: `get-window-content`, `get-window-canvas`, `new-window`. →index

### 3.13.122 `set-window-fixed-size`: procedure/2

Usage: (`set-window-fixed-size` *window* *fixed-size?*)

If *fixed-size?* is not nil, then *window* is set to fixed size, i.e., it has no resize button and cannot be resized by the user; otherwise, the window is set to being resizable.

See also: `window-fixed-size?`. →index

**3.13.123 set-window-full-screen : procedure/2**

Usage: (`set-window-full-screen window full-screen?`)

If `full-screen?` is not nil, then `window` is set to full screen mode, otherwise the window is set to normal mode. In full screen mode the window is centered and fills the whole screen of the main monitor (multiple monitors are currently not supported).

See also: `window-full-screen?`, `center-window-on-screen`. →index

**3.13.124 set-window-icon : procedure/1**

Usage: (`set-window-icon window resource`)

Set the icon of `window` to the given icon `resource`. Setting the icon does not guarantee that it is displayed, since this is platform-dependent.

See also: `get-window-icon`, `new-icon`, `theme-icon`. →index

**3.13.125 set-window-main-menu : procedure/2**

Usage: (`set-window-main-menu window main-menu`)

Set the main menu of `window` to `main-menu`.

See also: `new-main-menu`, `new-menu`, `new-menu*`. →index

**3.13.126 set-window-on-close-callback : procedure/2**

Usage: (`set-window-on-close-callback window proc`)

Sets the close callback of `window`. `proc` must be a function that takes no arguments and is evaluated when the window is closed.

See also: `show-window`, `close-window`, `hide-window`. →index

**3.13.127 set-window-padded : procedure/2**

Usage: (`set-window-padded window padded?`)

If `padded?` is not nil, then `window` is set to being padded. This is the default of new window. If `padded?` is nil, then the window's padding is removed, which means that the whole content area of the window

can be filled with user interface elements and draw commands. This would be used for a game display, for instance.

See also: `window-padded?`, `set-window-full-screen`, `window-full-screen?`, `center-window-on-screen`. [→index](#)

### 3.13.128 `set-window-size`: procedure/2

Usage: (`set-window-size` `window` `width` `height`)

Set the size of `window` to `width` and `height` as floats. Sizes and positions are generally given as floats whose accuracy is only guaranteed to 32 bit.

See also: `new-window`, `show-window`, `hide-window`. [→index](#)

### 3.13.129 `set-window-title`: procedure/2

Usage: (`set-window-title` `window` `title`)

Set the title of `window` to string `title`.

See also: `get-window-title`. [→index](#)

### 3.13.130 `show-object`: procedure/1

Usage: (`show-object` `obj`)

Show the canvas object `obj`.

See also: `disable-object`, `enable-object`, `hide-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. [→index](#)

### 3.13.131 `text-grid-show-line-numbers?`: procedure/1

Usage: (`text-grid-show-line-numbers?` `grid`)=> `bool`

Return true if the text `grid` shows line numbers, nil otherwise.

See also: `new-text-grid`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`,

`get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index

### 3.13.132 `text-grid-show-whitespace?` : procedure/1

Usage: (`text-grid-show-whitespace?` `grid`)=> `bool`

Return true if the text `grid` shows whitespace glyphs, nil otherwise.

See also: `new-text-grid`, `text-grid-show-line-numbers?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index

### 3.13.133 `theme-icon` : procedure/1

Usage: (`theme-icon` `selector`)=> `int`

Obtain a pre-defined icon from the application's theme based on the symbol `selector`, which may be one of '(cancel check-button check-button-checked color-achromatic color-chromatic color-palette computer confirm content-add content-clear content-copy content-cut content-paste content-redo content-remove content-undo delete document-create document-print document-download error file-application file-audio file-image file-text file-video file folder-new folder-open folder grid help history home info list login logout mail-attachment mail-compose mail-forward mail-reply-all mail-reply mail-send media-fast-forward media-fast-rewind media-music media-pause media-photo media-play media-record media-replay media-skip-next media-skip-previous media-stop media-video media-expand menu more-horizontal more-vertical move-down move-up navigate-back navigate-next question radio-button radio-button-checked search-replace search settings storage upload view-full-screen view-refresh view-restore visibility-off visibility volume-down volume-mute volume-up warning).

See also: `new-icon`, `new-image-from`, `new-image-from-resource`. →index

### 3.13.134 `trigger-menu-last` : procedure/1

Usage: (`trigger-menu-last` `menu`)

Find the last active menu or submenu item and trigger it.



See also: [activate-menu-last-submenu](#), [activate-menu-next](#), [activate-menu-previous](#), [new-menu](#). [→index](#)

### 3.13.135 `validate-object`: procedure/2

Usage: ([validate-object](#) *obj*)=> *str*

Validate the validatable object *obj* programmatically and return the validation failure as string, or the empty string if validation succeeded. It sometimes makes sense to call this explicitly in order to force the object to display its validation state.

See also: [set-object-on-validation-change-callback](#), [new-validator](#), [set-entry-validator](#). [→index](#)

### 3.13.136 `window-fixed-size?`: procedure/1

Usage: ([window-fixed-size?](#) *window*)=> *bool*

Return nil if *window* is fixed size, true otherwise.

See also: [set-window-fixed-size](#). [→index](#)

### 3.13.137 `window-full-screen?`: procedure/1

Usage: ([window-full-screen?](#) *window*)=> *bool*

Return nil if *window* is full screen, true otherwise.

See also: [set-window-full-screen](#), [center-window-on-screen](#). [→index](#)

### 3.13.138 `window-padded?`: procedure/1

Usage: ([window-padded?](#) *window*)=> *bool*

Return nil if *window* is padded, true otherwise.

See also: [set-window-padded](#), [set-window-full-screen](#), [center-window-on-screen](#). [→index](#)

## 3.14 Help System

This section lists functions related to the built-in help system.

### 3.14.1 **help** : dict

Usage: `*help*`

Dict containing all help information for symbols.

See also: `help`, `defhelp`, `apropos`. →index

### 3.14.2 **apropos** : procedure/1

Usage: `(apropos sym) => #li`

Get a list of procedures and symbols related to `sym` from the help system.

See also: `defhelp`, `help-entry`, `help`, `*help*`. →index

### 3.14.3 **help** : macro/1

Usage: `(help sym)`

Display help information about `sym` (unquoted).

See also: `defhelp`, `help-topics`, `help-about`, `help-topic-info`, `set-help-topic-info`, `help-entry`, `*help*`, `apropos`. →index

### 3.14.4 **help->manual-entry** : nil

Usage: `(help->manual-entry key [level] [link?]) => str`

Looks up help for `key` and converts it to a manual section as markdown string. If there is no entry for `key`, then nil is returned. The optional `level` integer indicates the heading nesting. If `link?` is true an anchor is created for the key.

See also: `help`. →index

### 3.14.5 **help-about** : procedure/1 or more

Usage: `(help-about topic [sel]) => li`

Obtain a list of symbols for which help about `topic` is available. If optional `sel` argument is left out or `any`, then any symbols with which the topic is associated are listed. If the optional `sel` argument is `first`, then a symbol is only listed if it has `topic` as first topic entry. This restricts the number of entries returned to a more essential selection.

See also: [help-topics](#), [help](#), [apropos](#). →index

### 3.14.6 [help-entry](#) : procedure/1

Usage: ([help-entry](#) [sym](#))=> [list](#)

Get usage and help information for [sym](#).

See also: [defhelp](#), [help](#), [apropos](#), [\\*help\\*](#), [help-topics](#), [help-about](#), [set-help-topic-info](#), [help-topic-info](#). →index

### 3.14.7 [help-strings](#) : procedure/2

Usage: ([help-strings](#) [sym](#) [del](#))=> [li](#)

Obtain a string of help strings for a given symbol [sym](#). The fields in the string are separated by string [del](#).

See also: [help](#), [help-entry](#), [\\*help\\*](#). →index

### 3.14.8 [help-topic-info](#) : procedure/1

Usage: ([help-topic-info](#) [topic](#))=> [li](#)

Return a list containing a heading and an info string for help [topic](#), or nil if no info is available.

See also: [set-help-topic-info](#), [defhelp](#), [help](#). →index

### 3.14.9 [help-topics](#) : procedure/0

Usage: ([help-topics](#))=> [li](#)

Obtain a list of help topics for commands.

See also: [help](#), [help-topic](#), [apropos](#). →index

### 3.14.10 [prune-unneeded-help-entries](#) : procedure/0

Usage: ([prune-unneeded-help-entries](#))

Remove help entries for which no toplevel symbol is defined. This function may need to be called when a module is not being used (e.g. because of a missing build tag) and it is desirable that only help for existing symbols is available.

See also: `find-unneeded-help-entries`, `find-missing-help-entries`, `help`, `*help*`.  
→index

### 3.14.11 `set-help-topic-info`: procedure/3

Usage: (`set-help-topic-info` `topic` `header` `info`)

Set a human-readable information entry for help `topic` with human-readable `header` and `info` strings.

See also: `defhelp`, `help-topic-info`. →index

## 3.15 Library System

This miscellaneous mini-library system allows importing programs with a prefix by source-transforming them.

### 3.15.1 `global-sym?`: procedure/1

Usage: (`global-sym?` `sym`)=> `bool`

Returns true if `sym` is a global symbol, nil otherwise. By convention, a symbol counts as global if it starts with a "\*" character. This is used by library functions to determine whether a top-level symbol ought to be treated as local or global to the library.

See also: `load`, `include`, `sym?`. →index

### 3.15.2 `load`: procedure/1 or more

Usage: (`load` `prefix` [`fi`])

Loads the Lisp file at `fi` as a library or program with the given `prefix`. If only a prefix is specified, `load` attempts to find a corresponding file at path (str+ (sysdir 'z3s5-data) "/prg/prefix/prefix.lisp"). Loading binds all non-global toplevel symbols of the definitions in file `fi` to the form `prefix.symbol` and replaces calls to them in the definitions appropriately. Symbols starting with "\*" *such as* `cancel*` are not modified. To give an example, if `fi` contains a definition (`defun bar ...`) and the prefix is 'foo, then the result of the import is equivalent to (`defun foo.bar ...`), and so on for any other definitions. The importer preorder-traverses the source and looks for `setq` and `lambdas` after macro expansion has taken place. By convention, the entry point of executable programs is a function (`run`) so the loaded program can be executed with the command (`prefix.run`).

See also: `include`, `global-sym?`. →index

### 3.16 Soundex, Metaphone, etc.

The package `ling` provides various phonemic transcription functions like Soundex and Metaphone that are commonly used for fuzzy search and similarity comparisons between strings.

#### 3.16.1 `ling.damerau-levenshtein: procedure/2`

Usage: `(ling.damerau-levenshtein s1 s2)=> num`

Compute the Damerau-Levenshtein distance between `s1` and `s2`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. →index

#### 3.16.2 `ling.hamming: procedure/2`

Usage: `(ling.hamming s1 s2)=> num`

Compute the Hamming distance between `s1` and `s2`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. →index

#### 3.16.3 `ling.jaro: procedure/2`

Usage: `(ling.jaro s1 s2)=> num`

Compute the Jaro distance between `s1` and `s2`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. →index

#### 3.16.4 `ling.jaro-winkler: procedure/2`

Usage: `(ling.jaro-winkler s1 s2)=> num`

Compute the Jaro-Winkler distance between `s1` and `s2`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. →index

### 3.16.5 `ling.levenshtein`: procedure/2

Usage: `(ling.levenshtein s1 s2)=> num`

Compute the Levenshtein distance between `s1` and `s2`.

See also: `ling.match-rating-compare`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. →index

### 3.16.6 `ling.match-rating-codex`: procedure/1

Usage: `(ling.match-rating-codex s)=> str`

Compute the Match-Rating-Codex of string `s`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. →index

### 3.16.7 `ling.match-rating-compare`: procedure/2

Usage: `(ling.match-rating-compare s1 s2)=> bool`

Returns true if `s1` and `s2` are equal according to the Match-rating Comparison algorithm, nil otherwise.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. →index

### 3.16.8 `ling.metaphone`: procedure/1

Usage: `(ling.metaphone s)=> str`

Compute the Metaphone representation of string `s`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.soundex`. [→index](#)

### 3.16.9 `ling.nysiis`: procedure/1

Usage: `(ling.nysiis s)=> str`

Compute the Nysiis representation of string `s`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.metaphone`, `ling.soundex`. [→index](#)

### 3.16.10 `ling.porter`: procedure/1

Usage: `(ling.porter s)=> str`

Compute the stem of word string `s` using the Porter stemming algorithm.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#)

### 3.16.11 `ling.soundex`: procedure/1

Usage: `(ling.soundex s)=> str`

Compute the Soundex representation of string `s`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#)

## 3.17 Lisp - Traditional Lisp Functions

This section comprises a large number of list processing functions as well the standard control flow macros and functions you'd expect in a Lisp system.

### 3.17.1 `alist?`: procedure/1

Usage: (`alist?` `li`)=> `bool`

Return true if `li` is an association list, nil otherwise. This also works for a-lists where each element is a pair rather than a full list.

See also: `assoc`. →index

### 3.17.2 `and`: macro/0 or more

Usage: (`and` `expr1` `expr2` ...)=> `any`

Evaluate `expr1` and if it is not nil, then evaluate `expr2` and if it is not nil, evaluate the next expression, until all expressions have been evaluated. This is a shortcut logical and.

See also: `or`. →index

### 3.17.3 `append`: procedure/1 or more

Usage: (`append` `li1` `li2` ...)=> `li`

Concatenate the lists given as arguments.

See also: `cons`. →index

### 3.17.4 `apply`: procedure/2

Usage: (`apply` `proc` `arg`)=> `any`

Apply function `proc` to argument list `arg`.

See also: `functional?`. →index

### 3.17.5 `assoc`: procedure/2

Usage: (`assoc` `key` `alist`)=> `li`

Return the sublist of `alist` that starts with `key` if there is any, nil otherwise. Testing is done with `equal?`. An association list may be of the form ((`key1` `value1`)(`key2` `value2`)...) or ((`key1` . `value1`) (`key2` . `value2`) ...)

See also: `assoc`, `assoc1`, `alist?`, `eq?`, `equal?`. →index



### 3.17.6 `assoc1` : procedure/2

Usage: (`assoc1` `sym` `li`)=> `any`

Get the second element in the first sublist in `li` that starts with `sym`. This is equivalent to (`cadr` (`assoc` `sym` `li`)).

See also: `assoc`, `alist?`. →index

### 3.17.7 `assq` : procedure/2

Usage: (`assq` `key` `alist`)=> `li`

Return the sublist of `alist` that starts with `key` if there is any, nil otherwise. Testing is done with `eq?`. An association list may be of the form ((`key1` `value1`)(`key2` `value2`)...) or ((`key1` . `value1`) (`key2` . `value2`) ...)

See also: `assoc`, `assoc1`, `eq?`, `alist?`, `equal?`. →index

### 3.17.8 `atom?` : procedure/1

Usage: (`atom?` `x`)=> `bool`

Return true if `x` is an atomic value, nil otherwise. Atomic values are numbers and symbols.

See also: `sym?`. →index

### 3.17.9 `bool?` : procedure/1

Usage: (`bool?` `datum`)=> `bool`

Return true if `datum` is either true or nil. Note: This predicate only exists for type-completeness and you should never use it as part of testing whether something is true or false - per convention, a value is true if it is non-nil and not when it is true, which is the special boolean value this predicate tests in addition to nil.

See also: `null?`, `not`. →index

### 3.17.10 `build-list` : procedure/2

Usage: (`build-list` `n` `proc`)=> `list`

Build a list with `n` elements by applying `proc` to the counter `n` each time.

See also: [list](#), [list?](#), [map](#), [foreach](#). →[index](#)

### 3.17.11 **caaar** : procedure/1

Usage: ([caaar](#) *x*)=> *any*

Equivalent to (car (car (car *x*))).

See also: [car](#), [cdr](#), [caar](#), [cadr](#), [cdar](#), [cddr](#), [caadr](#), [cadar](#), [caddr](#), [cdaar](#), [cdadr](#), [cddar](#), [cdddr](#), [nth](#), [1st](#), [2nd](#), [3rd](#). →[index](#)

### 3.17.12 **caadr** : procedure/1

Usage: ([caadr](#) *x*)=> *any*

Equivalent to (car (car (cdr *x*))).

See also: [car](#), [cdr](#), [caar](#), [cadr](#), [cdar](#), [cddr](#), [caaar](#), [cadar](#), [caddr](#), [cdaar](#), [cdadr](#), [cddar](#), [cdddr](#), [nth](#), [1st](#), [2nd](#), [3rd](#). →[index](#)

### 3.17.13 **caar** : procedure/1

Usage: ([caar](#) *x*)=> *any*

Equivalent to (car (car *x*)).

See also: [car](#), [cdr](#), [cadr](#), [cdar](#), [cddr](#), [caaar](#), [caadr](#), [cadar](#), [caddr](#), [cdaar](#), [cdadr](#), [cddar](#), [cdddr](#), [nth](#), [1st](#), [2nd](#), [3rd](#). →[index](#)

### 3.17.14 **cadar** : procedure/1

Usage: ([cadar](#) *x*)=> *any*

Equivalent to (car (cdr (car *x*))).

See also: [car](#), [cdr](#), [caar](#), [cadr](#), [cdar](#), [cddr](#), [caaar](#), [caadr](#), [caddr](#), [cdaar](#), [cdadr](#), [cddar](#), [cdddr](#), [nth](#), [1st](#), [2nd](#), [3rd](#). →[index](#)

### 3.17.15 **caddr** : procedure/1

Usage: ([caddr](#) *x*)=> *any*

Equivalent to `(car (cdr (cdr x)))`.

See also: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index

### 3.17.16 `cadr` : procedure/1

Usage: `(cadr x)` => `any`

Equivalent to `(car (cdr x))`.

See also: `car`, `cdr`, `caar`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index

### 3.17.17 `car` : procedure/1

Usage: `(car li)` => `any`

Get the first element of a list or pair `li`, an error if there is not first element.

See also: `list`, `list?`, `pair?`. →index

### 3.17.18 `case` : macro/2 or more

Usage: `(case expr (clause1 ... clausen))` => `any`

Standard case macro, where you should use `t` for the remaining alternative. Example: `(case (get dict 'key) ((a b) (out "a or b"))(t (out "something else!")))`.

See also: `cond`. →index

### 3.17.19 `cdaar` : procedure/1

Usage: `(cdaar x)` => `any`

Equivalent to `(cdr (car (car x)))`.

See also: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index

**3.17.20 cdadr : procedure/1**

Usage: (`cdadr` `x`)=> `any`

Equivalent to (`cdr` (`car` (`cdr` `x`))).

See also: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index

**3.17.21 cdar : procedure/1**

Usage: (`cdar` `x`)=> `any`

Equivalent to (`cdr` (`car` `x`)).

See also: `car`, `cdr`, `caar`, `cadr`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index

**3.17.22 cddar : procedure/1**

Usage: (`cddar` `x`)=> `any`

Equivalent to (`cdr` (`cdr` (`car` `x`))).

See also: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index

**3.17.23 cdddr : procedure/1**

Usage: (`cdddr` `x`)=> `any`

Equivalent to (`cdr` (`cdr` (`cdr` `x`))).

See also: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index

**3.17.24 cddr : procedure/1**

Usage: (`cddr` `x`)=> `any`

Equivalent to (`cdr` (`cdr` `x`)).

See also: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index

### 3.17.25 `cdr` : procedure/1

Usage: (`cdr` `li`)=> `any`

Get the rest of a list `li`. If the list is proper, the `cdr` is a list. If it is a pair, then it may be an element. If the list is empty, `nil` is returned.

See also: `car`, `list`, `list?`, `pair?`. →index

### 3.17.26 `cond` : special form

Usage: (`cond` ((`test1` `expr1` ...) (`test2` `expr2` ...) ...)=> `any`

Evaluate the tests sequentially and execute the expressions after the test when a test is true. To express the else case, use (`t` `exprn` ...) at the end of the `cond`-clauses to execute `exprn`...

See also: `if`, `when`, `unless`. →index

### 3.17.27 `cons` : procedure/2

Usage: (`cons` `a` `b`)=> `pair`

Cons two values into a pair. If `b` is a list, the result is a list. Otherwise the result is a pair.

See also: `cdr`, `car`, `list?`, `pair?`. →index

### 3.17.28 `cons?` : procedure/1

Usage: (`cons?` `x`)=> `bool`

return true if `x` is not an atom, nil otherwise.

See also: `atom?`. →index

### 3.17.29 `count-partitions` : procedure/2

Usage: (`count-partitions` `m` `k`)=> `int`

Return the number of partitions for dividing `m` items into parts of size `k` or less, where the size of the last partition may be less than `k` but the remaining ones have size `k`.

See also: `nth-partition`, `get-partitions`. →index

### 3.17.30 `defmacro` : macro/2 or more

Usage: `(defmacro name args body ...)`

Define a macro `name` with argument list `args` and `body`. Macros are expanded at compile-time.

See also: `macro`. [→index](#)

### 3.17.31 `defun` : macro/1 or more

Usage: `(defun ident (params ...) body ...)`

Define a function with name `ident`, a possibly empty list of `params`, and the remaining `body` expressions. This is a macro for `(setq ident (lambda (params ...) body ...))` and binds the lambda-form to the given symbol. Like lambdas, the `params` of `defun` allow for a `&rest` keyword before the last parameter name. This binds all remaining arguments of a variadic function call to this parameter as a list.

See also: `setq`, `defmacro`. [→index](#)

### 3.17.32 `dolist` : macro/1 or more

Usage: `(dolist (name list [result]) body ...)=> li`

Traverse the list `list` in order, binding `name` to each element subsequently and evaluate the `body` expressions with this binding. The optional `result` is the result of the traversal, nil if it is not provided.

See also: `letrec`, `foreach`, `map`. [→index](#)

### 3.17.33 `dotimes` : macro/1 or more

Usage: `(dotimes (name count [result]) body ...)=> any`

Iterate `count` times, binding `name` to the counter starting from 0 until the counter has reached count-1, and evaluate the `body` expressions each time with this binding. The optional `result` is the result of the iteration, nil if it is not provided.

See also: `letrec`, `dolist`, `while`. [→index](#)

### 3.17.34 `eq?` : procedure/2

Usage: `(eq? x y)=> bool`

Return true if `x` and `y` are equal, nil otherwise. In contrast to other LISPs, `eq?` checks for deep equality of arrays and dicts. However, lists are compared by checking whether they are the same cell in memory. Use `equal?` to check for deep equality of lists and other objects.

See also: `equal?`. →index

### 3.17.35 `eq?` : procedure/2

Usage: `(eq? x y) => bool`

Returns true if `x` is equal to `y`, nil otherwise. This is currently the same as `equal?` but the behavior might change.

See also: `equal?`. →index

**Warning: Deprecated.**

### 3.17.36 `equal?` : procedure/2

Usage: `(equal? x y) => bool`

Return true if `x` and `y` are equal, nil otherwise. The equality is tested recursively for containers like lists and arrays.

See also: `eq?`, `eq?`. →index

### 3.17.37 `filter` : procedure/2

Usage: `(filter li pred) => li`

Return the list based on `li` with each element removed for which `pred` returns nil.

See also: `list`. →index

### 3.17.38 `flatten` : procedure/1

Usage: `(flatten lst) => list`

Flatten `lst`, making all elements of sublists elements of the flattened list.

See also: `car`, `cdr`, `remove-duplicates`. →index

### 3.17.39 `get-partitions`: procedure/2

Usage: (`get-partitions` `x` `n`)=> `proc`/1\*

Return an iterator procedure that returns lists of the form (start-offset end-offset bytes) with 0-index offsets for a given index `k`, or nil if there is no corresponding part, such that the sizes of the partitions returned in `bytes` summed up are `x` and each partition is `n` or lower in size. The last partition will be the smallest partition with a `bytes` value smaller than `n` if `x` is not dividable without rest by `n`. If no argument is provided for the returned iterator, then it returns the number of partitions.

See also: `nth-partition`, `count-partitions`, `get-file-partitions`, `iterate`. →index

### 3.17.40 `identity`: procedure/1

Usage: (`identity` `x`)

Return `x`.

See also: `apply`, `equal?`. →index

### 3.17.41 `if`: macro/3

Usage: (`if` `cond` `expr1` `expr2`)=> `any`

Evaluate `expr1` if `cond` is true, otherwise evaluate `expr2`.

See also: `cond`, `when`, `unless`. →index

### 3.17.42 `iterate`: procedure/2

Usage: (`iterate` `it` `proc`)

Apply `proc` to each argument returned by iterator `it` in sequence, similar to the way `foreach` works. An iterator is a procedure that takes one integer as argument or no argument at all. If no argument is provided, the iterator returns the number of iterations. If an integer is provided, the iterator returns a non-nil value for the given index.

See also: `foreach`, `get-partitions`. →index

### 3.17.43 `lambda`: special form

Usage: (`lambda` `args` `body` ...)=> `closure`



Form a function closure (lambda term) with argument list in `args` and body expressions `body`.

See also: `defun`, `functional?`, `macro?`, `closure?`. →index

#### 3.17.44 `lcons` : procedure/2

Usage: `(lcons datum li)=> list`

Insert `datum` at the end of the list `li`. There may be a more efficient implementation of this in the future. Or, maybe not. Who knows?

See also: `cons`, `list`, `append`, `nreverse`. →index

#### 3.17.45 `let` : macro/1 or more

Usage: `(let args body ...)=> any`

Bind each pair of symbol and expression in `args` and evaluate the expressions in `body` with these local bindings. Return the value of the last expression in `body`.

See also: `letrec`. →index

#### 3.17.46 `letrec` : macro/1 or more

Usage: `(letrec args body ...)=> any`

Recursive let binds the symbol, expression pairs in `args` in a way that makes prior bindings available to later bindings and allows for recursive definitions in `args`, then evaluates the `body` expressions with these bindings.

See also: `let`. →index

#### 3.17.47 `list` : procedure/0 or more

Usage: `(list [args] ...)=> li`

Create a list from all `args`. The arguments must be quoted.

See also: `cons`. →index

**3.17.48 list-exists? : procedure/2**

Usage: `(list-exists? li pred)` => `bool`

Return true if `pred` returns true for at least one element in list `li`, nil otherwise.

See also: `exists?`, `forall?`, `array-exists?`, `str-exists?`, `seq?`. →index

**3.17.49 list-forall? : procedure/2**

Usage: `(list-all? li pred)` => `bool`

Return true if predicate `pred` returns true for all elements of list `li`, nil otherwise.

See also: `foreach`, `map`, `forall?`, `array-forall?`, `str-forall?`, `exists?`. →index

**3.17.50 list-foreach : procedure/2**

Usage: `(list-foreach li proc)`

Apply `proc` to each element of list `li` in order, for the side effects.

See also: `mapcar`, `map`, `foreach`. →index

**3.17.51 list-last : procedure/1**

Usage: `(list-last li)` => `any`

Return the last element of `li`.

See also: `reverse`, `nreverse`, `car`, `1st`, `last`. →index

**3.17.52 list-ref : procedure/2**

Usage: `(list-ref li n)` => `any`

Return the element with index `n` of list `li`. Lists are 0-indexed.

See also: `array-ref`, `nth`. →index

**3.17.53 list-reverse : procedure/1**

Usage: (`list-reverse` `li`)=> `li`

Create a reversed copy of `li`.

See also: `reverse`, `array-reverse`, `str-reverse`. →index

**3.17.54 list-slice : procedure/3**

Usage: (`list-slice` `li` `low` `high`)=> `li`

Return the slice of the list `li` starting at index `low` (inclusive) and ending at index `high` (exclusive).

See also: `slice`, `array-slice`. →index

**3.17.55 list? : procedure/1**

Usage: (`list?` `obj`)=> `bool`

Return true if `obj` is a list, nil otherwise.

See also: `cons?`, `atom?`, `null?`. →index

**3.17.56 macro : special form**

Usage: (`macro` `args` `body` ...)=> `macro`

Like a lambda term but the `body` expressions are macro-expanded at compile time instead of run-time.

See also: `defun`, `lambda`, `functional?`, `macro?`, `closure?`. →index

**3.17.57 mapcar : procedure/2**

Usage: (`mapcar` `li` `proc`)=> `li`

Return the list obtained from applying `proc` to each elements in `li`.

See also: `map`, `foreach`. →index

**3.17.58 member : procedure/2**

Usage: (`member` `key` `li`)=> `li`

Return the cdr of `li` starting with `key` if `li` contains an element equal? to `key`, nil otherwise.

See also: `assoc`, `equal?`. →index

**3.17.59 memq : procedure/2**

Usage: (`memq` `key` `li`)

Return the cdr of `li` starting with `key` if `li` contains an element eq? to `key`, nil otherwise.

See also: `member`, `eq?`. →index

**3.17.60 nconc : procedure/0 or more**

Usage: (`nconc` `li1` `li2` ...)=> `li`

Concatenate `li1`, `li2`, and so forth, like with `append`, but destructively modifies `li1`.

See also: `append`. →index

**3.17.61 not : procedure/1**

Usage: (`not` `x`)=> `bool`

Return true if `x` is nil, nil otherwise.

See also: `and`, `or`. →index

**3.17.62 nreverse : procedure/1**

Usage: (`nreverse` `li`)=> `li`

Destructively reverse `li`.

See also: `reverse`. →index

**3.17.63 nth-partition : procedure/3**

Usage: `(nth-partition m k idx)=> li`

Return a list of the form (start-offset end-offset bytes) for the partition with index `idx` of `m` into parts of size `k`. The index `idx` as well as the start- and end-offsets are 0-based.

See also: `count-partitions`, `get-partitions`. →index

**3.17.64 null? : procedure/1**

Usage: `(null? li)=> bool`

Return true if `li` is nil, nil otherwise.

See also: `not`, `list?`, `cons?`. →index

**3.17.65 num? : procedure/1**

Usage: `(num? n)=> bool`

Return true if `n` is a number (exact or inexact), nil otherwise.

See also: `str?`, `atom?`, `sym?`, `closure?`, `intrinsic?`, `macro?`. →index

**3.17.66 or : macro/0 or more**

Usage: `(or expr1 expr2 ...)=> any`

Evaluate the expressions until one of them is not nil. This is a logical shortcut or.

See also: `and`. →index

**3.17.67 progn : special form**

Usage: `(progn expr1 expr2 ...)=> any`

Sequentially execute the expressions `expr1`, `expr2`, and so forth, and return the value of the last expression.

See also: `defun`, `lambda`, `cond`. →index

### 3.17.68 **quasiquote** : special form

Usage: (`quasiquote` `li`)

Quote `li`, except that values in `li` may be unquoted (~evaluated) when prefixed with “,” and embedded lists can be unquote-splice by prefixing them with unquote-splice “,@”. An unquoted expression’s value is inserted directly, whereas unquote-splice inserts the values of a list in-sequence into the embedding list. Quasiquote is used in combination with gensym to define non-hygienic macros. In Z3S5 Lisp, “,” and “,@” are syntactic markers and there are no corresponding unquote and unquote-splice functions. The shortcut for quasiquote is “`~`”.

See also: `quote`, `gensym`, `macro`, `defmacro`. →index

### 3.17.69 **quote** : special form

Usage: (`quote` `x`)

Quote symbol `x`, so it evaluates to `x` instead of the value bound to it. Syntactic shortcut is ‘.

See also: `quasiquote`. →index

### 3.17.70 **rplacd** : procedure/2

Usage: (`rplacd` `li1` `li2`)=> `li`

Destructively replace the cdr of `li1` with `li2` and return the result afterwards.

See also: `rplaca`. →index

### 3.17.71 **rplaca** : procedure/2

Usage: (`rplaca` `li` `a`)=> `li`

Destructively mutate `li` such that its car is `a`, return the list afterwards.

See also: `rplacd`. →index

### 3.17.72 **setcar** : procedure/1

Usage: (`setcar` `li` `elem`)=> `li`

Mutate `li` such that its car is `elem`. Same as `rplaca`.

See also: `rplaca`, `rplacd`, `setcdr`. →index

**3.17.73 setcdr : procedure/1**

Usage: `(setcdr li li2)=> li`

Mutate `li1` such that its cdr is `li2`. Same as `rplacd`.

See also: `rplacd`, `rplaca`, `setcar`. →index

**3.17.74 setq : special form**

Usage: `(setq sym1 value1 ...)`

Set `sym1` (without need for quoting it) to `value`, and so forth for any further symbol, value pairs.

See also: `bind`, `unbind`. →index

**3.17.75 sort : procedure/2**

Usage: `(sort li proc)=> li`

Sort the list `li` by the given less-than procedure `proc`, which takes two arguments and returns true if the first one is less than the second, nil otherwise.

See also: `array-sort`. →index

**3.17.76 sort-symbols : nil**

Usage: `(sort-symbols li)=> list`

Sort the list of symbols `li` alphabetically.

See also: `out`, `dp`, `du`, `dump`. →index

**3.17.77 sym? : procedure/1**

Usage: `(sym? sym)=> bool`

Return true if `sym` is a symbol, nil otherwise.

See also: `str?`, `atom?`. →index

### 3.17.78 `type-of`: macro/1

Usage: (`type-of` `datum`)=> `sym`

Returns the type of `datum` as symbol like `type-of*` but without having to quote the argument. If `datum` is an unbound symbol, then this macro returns 'unbound. Otherwise the type of a given symbol's value or the type of a given literal is returned.

See also: `type-of*`. →index

### 3.17.79 `type-of*`: procedure/1

Usage: (`type-of*` `datum`)=> `sym`

Return the type of `datum` as a symbol. This uses existing predicates and therefore is not faster than testing with predicates directly.

See also: `num?`, `str?`, `sym?`, `list?`, `array?`, `bool?`, `eof?`, `boxed?`, `intrinsic?`, `closure?`, `macro?`, `blob?`. →index

### 3.17.80 `unless`: macro/1 or more

Usage: (`unless` `cond` `expr` ...)=> `any`

Evaluate expressions `expr` if `cond` is not true, returns void otherwise.

See also: `if`, `when`, `cond`. →index

### 3.17.81 `void`: procedure/0 or more

Usage: (`void` [`any`] ...)

Always returns void, no matter what values are given to it. Void is a special value that is not printed in the console.

See also: `void?`. →index

### 3.17.82 `void?`: procedure/1

Usage: (`void?` `datum`)

Return true if `datum` is the special symbol void, nil otherwise.

See also: `void`. →index



### 3.17.83 **when** : macro/1 or more

Usage: (**when** *cond* *expr* ...) => *any*

Evaluate the expressions *expr* if *cond* is true, returns void otherwise.

See also: **if**, **cond**, **unless**. →index

### 3.17.84 **while** : macro/1 or more

Usage: (**while** *test* *body* ...) => *any*

Evaluate the expressions in *body* while *test* is not nil.

See also: **letrec**, **dotimes**, **dolist**. →index

## 3.18 Numeric Functions

This section describes functions that provide standard arithmetics for non-floating point numbers such as integers. Notice that Z3S5 Lisp uses automatic bignum support but only for select standard operations like multiplication, addition, and subtraction.

### 3.18.1 **%** : procedure/2

Usage: (**%** *x* *y*) => *num*

Compute the remainder of dividing number *x* by *y*.

See also: **mod**, **/**. →index

### 3.18.2 **\*** : procedure/0 or more

Usage: (**\*** [*args*] ...) => *num*

Multiply all *args*. Special cases: *()* is 1 and *( x )* is *x*.

See also: **+**, **-**, **/**. →index

### 3.18.3 + : procedure/0 or more

Usage: (+ [args] ...) => num

Sum up all `args`. Special cases: (+) is 0 and (+ x) is x.

See also: -, \*, /. →index

### 3.18.4 - : procedure/1 or more

Usage: (- x [y1] [y2] ...) => num

Subtract `y1`, `y2`, ..., from `x`. Special case: (- x) is -x.

See also: +, \*, /. →index

### 3.18.5 / : procedure/1 or more

Usage: (/ x y1 [y2] ...) => float

Divide `x` by `y1`, then by `y2`, and so forth. The result is a float.

See also: +, \*, -. →index

### 3.18.6 /= : procedure/2

Usage: (/= x y) => bool

Return true if number `x` is not equal to `y`, nil otherwise.

See also: >, >=, <, <=. →index

### 3.18.7 < : procedure/2

Usage: (< x y) => bool

Return true if `x` is smaller than `y`.

See also: <=, >=, >. →index

**3.18.8 <= : procedure/2**

Usage: (`<= x y`)=> `bool`

Return true if `x` is smaller than or equal to `y`, nil otherwise.

See also: `>`, `<`, `>=`, `/=`. →index

**3.18.9 = : procedure/2**

Usage: (`= x y`)=> `bool`

Return true if number `x` equals number `y`, nil otherwise.

See also: `eq?`, `equal?`. →index

**3.18.10 > : procedure/2**

Usage: (`> x y`)=> `bool`

Return true if `x` is larger than `y`, nil otherwise.

See also: `<`, `>=`, `<=`, `/=`. →index

**3.18.11 >= : procedure/2**

Usage: (`>= x y`)=> `bool`

Return true if `x` is larger than or equal to `y`, nil otherwise.

See also: `>`, `<`, `<=`, `/=`. →index

**3.18.12 abs : procedure/1**

Usage: (`abs x`)=> `num`

Returns the absolute value of number `x`.

See also: `*`, `-`, `+`, `/`. →index

**3.18.13 add1 : procedure/1**

Usage: (`add1` `n`)=> `num`

Add 1 to number `n`.

See also: `sub1`, `+`, `-`. →index

**3.18.14 div : procedure/2**

Usage: (`div` `n` `k`)=> `int`

Integer division of `n` by `k`.

See also: `truncate`, `/`, `int`. →index

**3.18.15 even? : procedure/1**

Usage: (`even?` `n`)=> `bool`

Returns true if the integer `n` is even, nil if it is not even.

See also: `odd?`. →index

**3.18.16 float : procedure/1**

Usage: (`float` `n`)=> `float`

Convert `n` to a floating point value.

See also: `int`. →index

**3.18.17 int : procedure/1**

Usage: (`int` `n`)=> `int`

Return `n` as an integer, rounding down to the nearest integer if necessary.

See also: `float`. →index

**Warning:** If the number is very large this may result in returning the maximum supported integer number rather than the number as integer.

**3.18.18 max : procedure/1 or more**

Usage: (`max` `x1` `x2` ...) => `num`

Return the maximum of the given numbers.

See also: `min`, `minmax`. →index

**3.18.19 min : procedure/1 or more**

Usage: (`min` `x1` `x2` ...) => `num`

Return the minimum of the given numbers.

See also: `max`, `minmax`. →index

**3.18.20 minmax : procedure/3**

Usage: (`minmax` `pred` `li` `acc`) => `any`

Go through `li` and test whether for each `elem` the comparison (`pred` `elem` `acc`) is true. If so, `elem` becomes `acc`. Once all elements of the list have been compared, `acc` is returned. This procedure can be used to implement generalized minimum or maximum procedures.

See also: `min`, `max`. →index

**3.18.21 mod : procedure/2**

Usage: (`mod` `x` `y`) => `num`

Compute `x` modulo `y`.

See also: `%`, `/`. →index

**3.18.22 odd? : procedure/1**

Usage: (`odd?` `n`) => `bool`

Returns true if the integer `n` is odd, nil otherwise.

See also: `even?`. →index

### 3.18.23 rand : procedure/2

Usage: (rand prng lower upper) => **int**

Return a random integer in the interval [lower `` upper], both inclusive, from pseudo-random number generator prng. The prng argument must be an integer from 0 to 9 (inclusive).

See also: rnd, rndseed. →index

### 3.18.24 rnd : procedure/0

Usage: (rnd prng) => num

Return a random value in the interval [0, 1] from pseudo-random number generator prng. The prng argument must be an integer from 0 to 9 (inclusive).

See also: rand, rndseed. →index

### 3.18.25 rndseed : procedure/1

Usage: (rndseed prng n)

Seed the pseudo-random number generator prng (0 to 9) with 64 bit integer value n. Larger values will be truncated. Seeding affects both the rnd and the rand function for the given prng.

See also: rnd, rand. →index

### 3.18.26 sub1 : procedure/1

Usage: (sub1 n) => num

Subtract 1 from n.

See also: add1, +, -. →index

### 3.18.27 truncate : procedure/1 or more

Usage: (truncate x [y]) => **int**

Round down to nearest integer of x. If y is present, divide x by y and round down to the nearest integer.

See also: div, /, **int**. →index

### 3.19 Object-oriented Programming

The OOP system uses arrays to store objects and also offers a more lightweight array-based structure system. It is not built for performance but may be useful to prevent writing object-oriented wrapper data structures again and again. This is also the reason why it was decided to embed the OOP system with a fixed API rather than providing it as an include file, allowing for interoperable object-oriented programming without having to worry about whether the extension is loaded. It's very simple and lightweight.

#### 3.19.1 `call-method` : procedure/3

Usage: (`call-method` *obj* *mname* *args*)=> *any*

Execute method *mname* of object *obj* with additional arguments in list *args*. The first argument in the method call is always *obj* itself.

See also: `defmethod`, `defclass`, `new`, `isa?`, `class-of`. →index

#### 3.19.2 `call-super` : procedure/3

Usage: (`call-super` *obj* *mname* *args*)=> *any*

Execute method *mname* of the first superclass of *obj* that has a method with that name.

See also: `call-method`, `supers`. →index

#### 3.19.3 `class-name` : procedure/1

Usage: (`class-name` *c*)=> *sym*

Return the name of a class *c*. An error occurs if *c* is not a valid class.

See also: `class?`, `isa?`. →index

#### 3.19.4 `class-of` : procedure/1

Usage: (`class-of` *obj*)=> *class* or *nil*

Return the class of object *obj*, nil if *obj* is not a valid object array.

See also: `new`, `isa?`. →index

### 3.19.5 `class?` : procedure/1

Usage: (`class?` `c`)=> `bool`

Return true if `c` is a class array (not a name for a class!), nil otherwise.

See also: `object?`, `isa?`. →index

### 3.19.6 `copy-record` : procedure/1

Usage: (`copy-record` `r`)=> `record`

Creates a non-recursive, shallow copy of record `r`.

See also: `record?`. →index

### 3.19.7 `defclass` : macro/2 or more

Usage: (`defclass` `name` `supers` [`props`] ...)

Defines symbol `name` as class with superclasses `supers` and property clauses `props` listed as remaining arguments. A `props` clause is either a symbol for a property or a list of the form (sym default) for the property `sym` with `default` value. The class is bound to `name` and a class predicate `name?` is created. Argument `supers` may be a class name or a list of class names.

See also: `defmethod`, `new`. →index

### 3.19.8 `defmethod` : macro/2 or more

Usage: (`defmethod` `class-name` `args` [`body`] ...)

Define a method `class-name` for class `class` and method name `name` with a syntax parallel to `defun`, where `args` are the arguments of the methods and `body` is the rest of the method. The given `class-name` must decompose into a valid class name `class` of a previously created class and method name `name` and is bound to the symbol `class-name`. The remaining arguments are like for `defun`. So for example (`defmethod` `employee-name` (`this`) (`prop` `this` 'last-name)) defines a method `name` for an existing class `employee` which retrieves the property `last-name`. Note that `defmethod` is dynamic: If you define a class B with class A as superclass, then B only inherits methods from A that have already been defined for A at the time of defining B!

See also: `defclass`, `new`, `call-method`. →index



### 3.19.9 **defstruct** : macro/1 or more

Usage: `(defstruct name props ...)=> struct`

Binds symbol `name` to a struct with name `name` and with properties `props`. Each clause of `props` must be either a symbol for the property name or a list of the form `(prop default-value)` where `prop` is the symbol for the property name and `default-value` is the value it has by default. For each property `p`, accessors `name-p` and setters `name-p!` are created, as well as a function `name-p*` that takes a record `r`, a value `v`, and a procedure `proc` that takes no arguments. When `name-p*` is called on record `r`, it temporarily sets property `p` of `r` to the provided value `v` and calls the procedure `proc`. Afterwards, the original value of `p` is restored. Since this function mutates the record during the execution of `proc` and does not protect this operation against race conditions, it is not thread-safe. (But you can include a mutex as property and make it thread-safe by wrapping it into `with-mutex-lock`.) The `defstruct` macro returns the struct that is bound to `name`.

See also: `new-struct`, `make`, `with-mutex-lock`. →index

### 3.19.10 **has-method?** : procedure/2

Usage: `(has-method? obj name)=> bool`

Return true if `obj` has a method with name `name`, nil otherwise.

See also: `defmethod`, `has-prop?`, `new`, `props`, `methods`, `prop`, `setprop`. →index

### 3.19.11 **has-prop?** : procedure/2

Usage: `(has-prop? obj slot)=> bool`

Return true if `obj` has a property named `slot`, nil otherwise.

See also: `has-method?`, `new`, `props`, `methods`, `prop`, `setprop`. →index

### 3.19.12 **isa?** : procedure/2

Usage: `(isa? obj class)=> bool`

Return true if `obj` is an instance of `class`, nil otherwise.

See also: `supers`. →index

### 3.19.13 **make** : macro/2

Usage: (`make` `name` `props`)

Create a new record (struct instance) of struct `name` (unquoted) with properties `props`. Each clause in `props` must be a list of property name and initial value.

See also: `make*`, `defstruct`. →index

### 3.19.14 **make\*** : macro/1 or more

Usage: (`make*` `name` `prop1` ...)

Create a new record (struct instance) of struct `name` (unquoted) with property clauses `prop-1` ... `prop-n`, where each clause is a list of property name and initial value like in `make`.

See also: `make`, `defstruct`. →index

### 3.19.15 **methods** : procedure/1

Usage: (`methods` `obj`)=> `li`

Return the list of methods of `obj`, which must be a class, object, or class name.

See also: `has-method?`, `new`, `props`, `prop`, `setprop`, `has-prop?`. →index

### 3.19.16 **new** : macro/1 or more

Usage: (`new class` [`props`] ...)

Create a new object of class `class` with initial property bindings `props` clauses as remaining arguments. Each `props` clause must be a list of the form (`sym` `value`), where `sym` is a symbol and `value` is evaluated first before it is assigned to `sym`.

See also: `defclass`. →index

### 3.19.17 **new-struct** : procedure/2

Usage: (`new-struct` `name` `li`)

Defines a new structure `name` with the properties in the a-list `li`. Structs are more lightweight than classes and do not allow for inheritance. Instances of structs (“records”) are arrays.

See also: `defstruct`. →index

### 3.19.18 `object?` : procedure/1

Usage: `(object? obj)` => `bool`

Return true if `obj` is an object array, nil otherwise.

See also: `class?`, `isa?`. →index

### 3.19.19 `prop` : procedure/2

Usage: `(prop obj slot)` => `any`

Return the value in `obj` for property `slot`, or an error if the object does not have a property with that name.

See also: `new`, `isa?`, `setslot`, `object?`, `class-name`, `supers`, `props`, `methods`, `has-slot?`. →index

### 3.19.20 `props` : procedure/1

Usage: `(props obj)` => `li`

Return the list of properties of `obj`. An error occurs if `obj` is not a valid object.

See also: `methods`, `has-prop?`, `new`, `prop`, `setprop`. →index

### 3.19.21 `record?` : procedure/1

Usage: `(record? s)` => `bool`

Returns true if `s` is a struct record, i.e., an instance of a struct; nil otherwise. Notice that records are not really types distinct from arrays, they simply contain a marker '%record as first element. With normal use no confusion should arise. Since the internal representation might change, you ought not use ordinary array procedures for records.

See also: `struct?`, `defstruct`. →index

### 3.19.22 `setprop` : procedure/3

Usage: `(setprop obj slot value)`

Set property `slot` in `obj` to `value`. An error occurs if the object does not have a property with that name.

See also: [new](#), [isa?](#), [prop](#), [object?](#), [class-name](#), [supers](#), [props](#), [methods](#), [has-prop?](#). [→index](#)

### 3.19.23 **struct-index** : procedure/1

Usage: ([struct-index](#) *s*)=> *dict*

Returns the index of struct *s* as a dict. This dict is an internal representation of the struct's instance data.

See also: [defstruct](#). [→index](#)

### 3.19.24 **struct-instantiate** : procedure/2

Usage: ([struct-instantiate](#) *s* *li*)=> *record*

Instantiates the struct *s* with property a-list *li* as values for its properties and return the record. If a property is not in *li*, its value is set to nil.

See also: [make](#), [defstruct](#), [struct?](#), [record?](#). [→index](#)

### 3.19.25 **struct-name** : procedure/1

Usage: ([struct-name](#) *s*)=> *sym*

Returns the name of a struct *s*. This is rarely needed since the struct is bound to a symbol with the same name.

See also: [defstruct](#). [→index](#)

### 3.19.26 **struct-props** : procedure/1

Usage: ([struct-props](#) *s*)=> *dict*

Returns the properties of structure *s* as dict.

See also: [defstruct](#). [→index](#)

### 3.19.27 **struct-size** : procedure/1

Usage: ([struct-size](#) *s*)=> *int*

Returns the number of properties of struct *s*.

See also: [defstruct](#). [→index](#)

### 3.19.28 `struct? : procedure/1`

Usage: `(struct? datum) => boo`

Returns true if `datum` is a struct, nil otherwise.

See also: `defstruct`. [→index](#)

### 3.19.29 `supers : procedure/1`

Usage: `(supers c) => li`

Return the list of superclasses of class `c`. An error occurs if `c` is not a valid class.

See also: `class?`, `isa?`, `class-name`. [→index](#)

## 3.20 Semver Semantic Versioning

The `semver` package provides functions to deal with the validation and parsing of semantic versioning strings.

### 3.20.1 `semver.build : procedure/1`

Usage: `(semver.build s) => str`

Return the build part of a semantic versioning string.

See also: `semver.canonical`, `semver.major`, `semver.major-minor`. [→index](#)

### 3.20.2 `semver.canonical : procedure/1`

Usage: `(semver.canonical s) => str`

Return a canonical semver string based on a valid, yet possibly not canonical version string `s`.

See also: `semver.major`. [→index](#)

### 3.20.3 `semver.compare : procedure/2`

Usage: `(semver.compare s1 s2) => int`

Compare two semantic version strings `s1` and `s2`. The result is 0 if `s1` and `s2` are the same version, -1 if `s1 < s2` and 1 if `s1 > s2`.

See also: `semver.major`, `semver.major-minor`. →index

### 3.20.4 `semver.is-valid? : procedure/1`

Usage: `(semver.is-valid? s) => bool`

Return true if `s` is a valid semantic versioning string, nil otherwise.

See also: `semver.major`, `semver.major-minor`, `semver.compare`. →index

### 3.20.5 `semver.major : procedure/1`

Usage: `(semver.major s) => str`

Return the major part of the semantic versioning string.

See also: `semver.major-minor`, `semver.build`. →index

### 3.20.6 `semver.major-minor : procedure/1`

Usage: `(semver.major-minor s) => str`

Return the major.minor prefix of a semantic versioning string. For example, `(semver.major-minor "v2.1.4")` returns "v2.1".

See also: `semver.major`, `semver.build`. →index

### 3.20.7 `semver.max : procedure/2`

Usage: `(semver.max s1 s2) => str`

Canonicalize `s1` and `s2` and return the larger version of them.

See also: `semver.compare`. →index

### 3.20.8 `semver.prerelease : procedure/1`

Usage: `(semver.prerelease s) => str`

Return the prerelease part of a version string, or the empty string if there is none. For example, `(semver.prerelease "v2.1.0-pre+build")` returns "-pre".

See also: `semver.build`, `semver.major`, `semver.major-minor`. →index

## 3.21 Sequence Functions

Sequences are either strings, lists, or arrays. Sequence functions are generally abstractions for more specific functions of these data types, and therefore may be a bit slower than their native counterparts. It is still recommended to use them liberally, since they make programs more readable.

### 3.21.1 10th : procedure/1 or more

Usage: (10th seq [default])=> any

Get the tenth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [4th](#), [5th](#), [6th](#), [7th](#), [8th](#), [9th](#). →index

### 3.21.2 1st : procedure/1 or more

Usage: (1st seq [default])=> any

Get the first element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [2nd](#), [3rd](#), [4th](#), [5th](#), [6th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index

### 3.21.3 2nd : procedure/1 or more

Usage: (2nd seq [default])=> any

Get the second element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [3rd](#), [4th](#), [5th](#), [6th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index

### 3.21.4 3rd : procedure/1 or more

Usage: (3rd seq [default])=> any

Get the third element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [4th](#), [5th](#), [6th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index

### 3.21.5 4th : procedure/1 or more

Usage: (4th seq [default])=> any

Get the fourth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [5th](#), [6th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index

### 3.21.6 5th : procedure/1 or more

Usage: (5th seq [default])=> any

Get the fifth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [4th](#), [6th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index

### 3.21.7 6th : procedure/1 or more

Usage: (6th seq [default])=> any

Get the sixth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [4th](#), [5th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index

### 3.21.8 7th : procedure/1 or more

Usage: (7th seq [default])=> any

Get the seventh element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [4th](#), [5th](#), [6th](#), [8th](#), [9th](#), [10th](#). →index



### 3.21.9 8th : procedure/1 or more

Usage: (8th seq [default])=> any

Get the eighth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: nth, nthdef, car, list-ref, array-ref, string-ref, 1st, 2nd, 3rd, 4th, 5th, 6th, 7th, 9th, 10th. →index

### 3.21.10 9th : procedure/1 or more

Usage: (9th seq [default])=> any

Get the ninth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: nth, nthdef, car, list-ref, array-ref, string-ref, 1st, 2nd, 3rd, 4th, 5th, 6th, 7th, 8th, 10th. →index

### 3.21.11 exists? : procedure/2

Usage: (exists? seq pred)=> bool

Return true if **pred** returns true for at least one element in sequence **seq**, nil otherwise.

See also: forall?, list-exists?, array-exists?, str-exists?, seq?. →index

### 3.21.12 forall? : procedure/2

Usage: (forall? seq pred)=> bool

Return true if predicate **pred** returns true for all elements of sequence **seq**, nil otherwise.

See also: foreach, map, list-forall?, array-forall?, str-forall?, exists?, str-exists?, array-exists?, list-exists?. →index

### 3.21.13 foreach : procedure/2

Usage: (foreach seq proc)

Apply **proc** to each element of sequence **seq** in order, for the side effects.

See also: seq?, map. →index

### 3.21.14 `index` : procedure/2 or more

Usage: (`index` `seq` `elem` [`pred`])=> `int`

Return the first index of `elem` in `seq` going from left to right, using equality predicate `pred` for comparisons (default is `eq?`). If `elem` is not in `seq`, -1 is returned.

See also: `nth`, `seq?`. →index

### 3.21.15 `last` : procedure/1 or more

Usage: (`last` `seq` [`default`])=> `any`

Get the last element of sequence `seq` or return `default` if the sequence is empty. If `default` is not given and the sequence is empty, an error is raised.

See also: `nth`, `nthdef`, `car`, `list-ref`, `array-ref`, `string-ref`, `1st`, `2nd`, `3rd`, `4th`, `5th`, `6th`, `7th`, `8th`, `9th`, `10th`. →index

### 3.21.16 `len` : procedure/1

Usage: (`len` `seq`)=> `int`

Return the length of `seq`. Works for lists, strings, arrays, and dicts.

See also: `seq?`. →index

### 3.21.17 `map` : procedure/2

Usage: (`map` `seq` `proc`)=> `seq`

Return the copy of `seq` that is the result of applying `proc` to each element of `seq`.

See also: `seq?`, `mapcar`, `strmap`. →index

### 3.21.18 `map-pairwise` : procedure/2

Usage: (`map-pairwise` `seq` `proc`)=> `seq`

Applies `proc` in order to subsequent pairs in `seq`, assembling the sequence that results from the results of `proc`. Function `proc` takes two arguments and must return a proper list containing two elements. If the number of elements in `seq` is odd, an error is raised.

See also: `map`. →index

**3.21.19 nth : procedure/2**

Usage: (`nth seq n`)=> `any`

Get the `n`-th element of sequence `seq`. Sequences are 0-indexed.

See also: `nthdef`, `list`, `array`, `string`, `1st`, `2nd`, `3rd`, `4th`, `5th`, `6th`, `7th`, `8th`, `9th`, `10th`.  
→index

**3.21.20 nthdef : procedure/3**

Usage: (`nthdef seq n default`)=> `any`

Return the `n`-th element of sequence `seq` (0-indexed) if `seq` is a sequence and has at least `n+1` elements, default otherwise.

See also: `nth`, `seq?`, `1st`, `2nd`, `3rd`, `4th`, `5th`, `6th`, `7th`, `8th`, `9th`, `10th`. →index

**3.21.21 remove-duplicates : procedure/1**

Usage: (`remove-duplicates seq`)=> `seq`

Remove all duplicates in sequence `seq`, return a new sequence with the duplicates removed.

See also: `seq?`, `map`, `foreach`, `nth`. →index

**3.21.22 reverse : procedure/1**

Usage: (`reverse seq`)=> `sequence`

Reverse a sequence non-destructively, i.e., return a copy of the reversed sequence.

See also: `nth`, `seq?`, `1st`, `2nd`, `3rd`, `4th`, `6th`, `7th`, `8th`, `9th`, `10th`, `last`. →index

**3.21.23 seq? : procedure/1**

Usage: (`seq? seq`)=> `bool`

Return true if `seq` is a sequence, nil otherwise.

See also: `list`, `array`, `string`, `slice`, `nth`. →index

### 3.21.24 `slice` : procedure/3

Usage: (`slice seq low high`)=> `seq`

Return the subsequence of `seq` starting from `low` inclusive and ending at `high` exclusive. Sequences are 0-indexed.

See also: `list`, `array`, `string`, `nth`, `seq?`. →index

### 3.21.25 `take` : procedure/3

Usage: (`take seq n`)=> `seq`

Return the sequence consisting of the `n` first elements of `seq`.

See also: `list`, `array`, `string`, `nth`, `seq?`. →index

## 3.22 Sound Support

Only a few functions are provided for sound support.

### 3.22.1 `beep` : procedure/1

Usage: (`beep sel`)

Play a built-in system sound. The argument `sel` may be one of '(error start ready click okay confirm info).

See also: `set-volume`. →index

### 3.22.2 `set-volume` : procedure/1

Usage: (`set-volume fl`)

Set the master volume for all sound to `fl`, a value between 0.0 and 1.0.

See also: `beep`. →index

## 3.23 String Manipulation

These functions all manipulate strings in one way or another.

### 3.23.1 **fmt** : procedure/1 or more

Usage: (**fmt** *s* [*args*] ...) => *str*

Format string *s* that contains format directives with arbitrary many *args* as arguments. The number of format directives must match the number of arguments. The format directives are the same as those for the esoteric and arcane programming language “Go”, which was used on Earth for some time.

See also: [out](#). →index

### 3.23.2 **instr** : procedure/2

Usage: (**instr** *s1* *s2*) => **int**

Return the index of the first occurrence of *s2* in *s1* (from left), or -1 if *s1* does not contain *s2*.

See also: [str?](#), [index](#). →index

### 3.23.3 **shorten** : procedure/2

Usage: (**shorten** *s* *n*) => *str*

Shorten string *s* to length *n* in a smart way if possible, leave it untouched if the length of *s* is smaller than *n*.

See also: [substr](#). →index

### 3.23.4 **spaces** : procedure/1

Usage: (**spaces** *n*) => *str*

Create a string consisting of *n* spaces.

See also: [strbuild](#), [strleft](#), [strright](#). →index

### 3.23.5 **str+** : procedure/0 or more

Usage: (**str+** [*s*] ...) => *str*

Append all strings given to the function.

See also: [str?](#). →index

### 3.23.6 `str-count-substr` : procedure/2

Usage: `(str-count-substr s1 s2)`=> `int`

Count the number of non-overlapping occurrences of substring `s2` in string `s1`.

See also: `str-replace`, `str-replace*`, `instr`. →index

### 3.23.7 `str-empty?` : procedure/1

Usage: `(str-empty? s)`=> `bool`

Return true if the string `s` is empty, nil otherwise.

See also: `strlen`. →index

### 3.23.8 `str-exists?` : procedure/2

Usage: `(str-exists? s pred)`=> `bool`

Return true if `pred` returns true for at least one character in string `s`, nil otherwise.

See also: `exists?`, `forall?`, `list-exists?`, `array-exists?`, `seq?`. →index

### 3.23.9 `str-forall?` : procedure/2

Usage: `(str-forall? s pred)`=> `bool`

Return true if predicate `pred` returns true for all characters in string `s`, nil otherwise.

See also: `foreach`, `map`, `forall?`, `array-forall?`, `list-forall`, `exists?`. →index

### 3.23.10 `str-foreach` : procedure/2

Usage: `(str-foreach s proc)`

Apply `proc` to each element of string `s` in order, for the side effects.

See also: `foreach`, `list-foreach`, `array-foreach`, `map`. →index

### 3.23.11 `str-index`: procedure/2 or more

Usage: (`str-index` *s* *chars* [*pos*])=> **int**

Find the first char in *s* that is in the charset *chars*, starting from the optional *pos* in *s*, and return its index in the string. If no matching char is found, nil is returned.

See also: `strsplit`, `chars`, `inchars`. →index

### 3.23.12 `str-join`: procedure/2

Usage: (`str-join` *li* *del*)=> *str*

Join a list of strings *li* where each of the strings is separated by string *del*, and return the result string.

See also: `strlen`, `strsplit`, `str-slice`. →index

### 3.23.13 `str-ref`: procedure/2

Usage: (`str-ref` *s* *n*)=> *n*

Return the unicode char as integer at position *n* in *s*. Strings are 0-indexed.

See also: `nth`. →index

### 3.23.14 `str-remove-number`: procedure/1

Usage: (`str-remove-number` *s* [*del*])=> *str*

Remove the suffix number in *s*, provided there is one and it is separated from the rest of the string by *del*, where the default is a space character. For instance, “Test 29” will be converted to “Test”, “User-Name1-23-99” with delimiter “-” will be converted to “User-Name1-23”. This function will remove intermediate delimiters in the middle of the string, since it disassembles and reassembles the string, so be aware that this is not preserving inputs in that respect.

See also: `strsplit`. →index

### 3.23.15 `str-remove-prefix`: procedure/1

Usage: (`str-remove-prefix` *s* *prefix*)=> *str*

Remove the prefix `prefix` from string `s`, return the string without the prefix. If the prefix does not match, `s` is returned. If `prefix` is longer than `s` and matches, the empty string is returned.

See also: `str-remove-suffix`. [→index](#)

### 3.23.16 `str-remove-suffix`: procedure/1

Usage: `(str-remove-suffix s suffix)=> str`

remove the suffix `suffix` from string `s`, return the string without the suffix. If the suffix does not match, `s` is returned. If `suffix` is longer than `s` and matches, the empty string is returned.

See also: `str-remove-prefix`. [→index](#)

### 3.23.17 `str-replace`: procedure/4

Usage: `(str-replace s t1 t2 n)=> str`

Replace the first `n` instances of substring `t1` in `s` by `t2`.

See also: `str-replace*`, `str-count-substr`. [→index](#)

### 3.23.18 `str-replace*`: procedure/3

Usage: `(str-replace* s t1 t2)=> str`

Replace all non-overlapping substrings `t1` in `s` by `t2`.

See also: `str-replace`, `str-count-substr`. [→index](#)

### 3.23.19 `str-reverse`: procedure/1

Usage: `(str-reverse s)=> str`

Reverse string `s`.

See also: `reverse`, `array-reverse`, `list-reverse`. [→index](#)

### 3.23.20 `str-segment`: procedure/3

Usage: `(str-segment str start end)=> list`



Parse a string `str` into words that start with one of the characters in string `start` and end in one of the characters in string `end` and return a list consisting of lists of the form `(bool s)` where `bool` is true if the string starts with a character in `start`, nil otherwise, and `s` is the extracted string including start and end characters.

See also: `str+`, `strsplit`, `fmt`, `strbuild`. →index

### 3.23.21 `str-slice`: procedure/3

Usage: `(str-slice s low high)=> s`

Return a slice of string `s` starting at character with index `low` (inclusive) and ending at character with index `high` (exclusive).

See also: `slice`. →index

### 3.23.22 `strbuild`: procedure/2

Usage: `(strbuild s n)=> str`

Build a string by repeating string `s` `n` times.

See also: `str+`. →index

### 3.23.23 `strcase`: procedure/2

Usage: `(strcase s sel)=> str`

Change the case of the string `s` according to selector `sel` and return a copy. Valid values for `sel` are 'lower for conversion to lower-case, 'upper for uppercase, 'title for title case and 'utf-8 for utf-8 normalization (which replaces unprintable characters with "?").

See also: `strmap`. →index

### 3.23.24 `strcenter`: procedure/2

Usage: `(strcenter s n)=> str`

Center string `s` by wrapping space characters around it, such that the total length the result string is `n`.

See also: `strleft`, `strright`, `strlimit`. →index

**3.23.25 strcnt: procedure/2**

Usage: (`strcnt s del`)=> `int`

Return the number of non-overlapping substrings `del` in `s`.

See also: `strsplit`, `str-index`. →index

**3.23.26 strleft: procedure/2**

Usage: (`strleft s n`)=> `str`

Align string `s` left by adding space characters to the right of it, such that the total length the result string is `n`.

See also: `strcenter`, `strright`, `strlimit`. →index

**3.23.27 strlen: procedure/1**

Usage: (`strlen s`)=> `int`

Return the length of `s`.

See also: `len`, `seq?`, `str?`. →index

**3.23.28 strless: procedure/2**

Usage: (`strless s1 s2`)=> `bool`

Return true if string `s1` < `s2` in lexicographic comparison, nil otherwise.

See also: `sort`, `array-sort`, `strcase`. →index

**3.23.29 strlimit: procedure/2**

Usage: (`strlimit s n`)=> `str`

Return a string based on `s` cropped to a maximal length of `n` (or less if `s` is shorter).

See also: `strcenter`, `strleft`, `strright`. →index

**3.23.30 strmap : procedure/2**

Usage: (`strmap s proc`)=> `str`

Map function `proc`, which takes a number and returns a number, over all unicode characters in `s` and return the result as new string.

See also: `map`. →index

**3.23.31 stropen : procedure/1**

Usage: (`stropen s`)=> `streamport`

Open the string `s` as input stream.

See also: `open`, `close`. →index

**3.23.32 strright : procedure/2**

Usage: (`strright s n`)=> `str`

Align string `s` right by adding space characters in front of it, such that the total length the result string is `n`.

See also: `strcenter`, `strleft`, `strlimit`. →index

**3.23.33 strsplit : procedure/2**

Usage: (`strsplit s del`)=> `array`

Return an array of strings obtained from `s` by splitting `s` at each occurrence of string `del`.

See also: `str?`. →index

**3.24 System Functions**

These functions concern the inner workings of the Lisp interpreter. Your warranty might be void if you abuse them!

### 3.24.1 *error-handler* : dict

Usage: (*\*error-handler\** *err*)

The global error handler dict that contains procedures which take an error and handle it. If an entry is nil, the default handler is used, which outputs the error using *error-printer*. The dict contains handlers based on concurrent thread IDs and ought not be manipulated directly.

See also: *\*error-printer\**. [→index](#)

### 3.24.2 *\*error-printer\** : procedure/1

Usage: (*\*error-printer\** *err*)

The global printer procedure which takes an error and prints it.

See also: *error*. [→index](#)

### 3.24.3 *last-error* : sym

Usage: *\*last-error\** => *str*

Contains the last error that has occurred.

See also: *\*error-printer\**, *\*error-handler\**. [→index](#)

**Warning: This may only be used for debugging! Do not use this for error handling, it will surely fail!**

### 3.24.4 *reflect* : symbol

Usage: *\*reflect\** => *li*

The list of feature identifiers as symbols that this Lisp implementation supports.

See also: *feature?*, *on-feature*. [→index](#)

### 3.24.5 *add-hook* : procedure/2

Usage: (*add-hook* *hook* *proc*)=> *id*

Add hook procedure *proc* which takes a list of arguments as argument under symbolic or numeric *hook* and return an integer hook *id* for this hook. If *hook* is not known, nil is returned.

See also: *remove-hook*, *remove-hooks*, *replace-hook*. [→index](#)

### 3.24.6 `add-hook-internal` : procedure/2

Usage: `(add-hook-internal hook proc)`=> `int`

Add a procedure `proc` to hook with numeric ID `hook` and return this procedures hook ID. The function does not check whether the hook exists.

See also: `add-hook`. →index

**Warning: Internal use only.**

### 3.24.7 `add-hook-once` : procedure/2

Usage: `(add-hook-once hook proc)`=> `id`

Add a hook procedure `proc` which takes a list of arguments under symbolic or numeric `hook` and return an integer hook `id`. If `hook` is not known, nil is returned.

See also: `add-hook`, `remove-hook`, `replace-hook`. →index

### 3.24.8 `bind` : procedure/2

Usage: `(bind sym value)`

Bind `value` to the global symbol `sym`. In contrast to `setq` both values need quoting.

See also: `setq`. →index

### 3.24.9 `bound?` : macro/1

Usage: `(bound? sym)`=> `bool`

Return true if a value is bound to the symbol `sym`, nil otherwise.

See also: `bind`, `setq`. →index

### 3.24.10 `boxed?` : procedure/1

Usage: `(boxed? x)`=> `bool`

Return true if `x` is a boxed value, nil otherwise. Boxed values are special objects that are special in the system and sometimes cannot be garbage collected.

See also: `type-of`, `num?`, `str?`, `sym?`, `list?`, `array?`, `macro?`, `closure?`, `intrinsic?`, `eof?`. →index

### 3.24.11 `can-externalize?` : procedure/1

Usage: (`can-externalize?` datum)=> bool

Recursively determines if `datum` can be externalized and returns true in this case, nil otherwise.

See also: `externalize`, `externalize0`. →index

### 3.24.12 `closure?` : procedure/1

Usage: (`closure?` x)=> bool

Return true if `x` is a closure, nil otherwise. Use `function?` for testing whether `x` can be executed.

See also: `functional?`, `macro?`, `intrinsic?`, `functional-arity`, `functional-has-rest?`. →index

### 3.24.13 `collect-garbage` : procedure/0 or more

Usage: (`collect-garbage` [sort])

Force a garbage-collection of the system's memory. If `sort` is 'normal, then only a normal incremental garbage collection is performed. If `sort` is 'total, then the garbage collection is more thorough and the system attempts to return unused memory to the host OS. Default is 'normal.

See also: `memstats`. →index

**Warning: There should rarely be a use for this. Try to use less memory-consuming data structures instead.**

### 3.24.14 `current-error-handler` : procedure/0

Usage: (`current-error-handler`)=> proc

Return the current error handler, a default if there is none.

See also: `default-error-handler`, `push-error-handler`, `pop-error-handler`, `*current-error-handler*`, `*current-error-continuation*`. →index

### 3.24.15 `def-custom-hook` : procedure/2

Usage: (`def-custom-hook` sym proc)

Define a custom hook point, to be called manually from Lisp. These have IDs starting from 65636.

See also: [add-hook](#). [→index](#)

### 3.24.16 `default-error-handler` : procedure/0

Usage: (`default-error-handler`)=> `proc`

Return the default error handler, irrespectively of the current-error-handler.

See also: [current-error-handler](#), [push-error-handler](#), [pop-error-handler](#), [\\*current-error-handler\\*](#), [\\*current-error-continuation\\*](#). [→index](#)

### 3.24.17 `dict-protect` : procedure/1

Usage: (`dict-protect` `d`)

Protect dict `d` against changes. Attempting to set values in a protected dict will cause an error, but all values can be read and the dict can be copied. This function requires permission 'allow-protect.

See also: [dict-unprotect](#), [dict-protected?](#), [protect](#), [unprotect](#), [protected?](#), [permissions](#), [permission?](#). [→index](#)

**Warning: Protected dicts are full readable and can be copied, so you may need to use `protect` to also prevent changes to the toplevel symbol storing the dict!**

### 3.24.18 `dict-protected?` : procedure/1

Usage: (`dict-protected?` `d`)

Return true if the dict `d` is protected against mutation, nil otherwise.

See also: [dict-protect](#), [dict-unprotect](#), [protect](#), [unprotect](#), [protected?](#), [permissions](#), [permission?](#). [→index](#)

### 3.24.19 `dict-unprotect` : procedure/1

Usage: (`dict-unprotect` `d`)

Unprotect the dict `d` so it can be mutated again. This function requires permission 'allow-unprotect.

See also: [dict-protect](#), [dict-protected?](#), [protect](#), [unprotect](#), [protected?](#), [permissions](#), [permission?](#). [→index](#)

**3.24.20 dump : procedure/0 or more**

Usage: (`dump` [`sym`] [`all?`])=> `li`

Return a list of symbols starting with the characters of `sym` or starting with any characters if `sym` is omitted, sorted alphabetically. When `all?` is true, then all symbols are listed, otherwise only symbols that do not contain "\_" are listed. By convention, the underscore is used for auxiliary functions.

See also: `dump-bindings`, `save-zimage`, `load-zimage`. →index

**3.24.21 dump-bindings : procedure/0**

Usage: (`dump-bindings`)=> `li`

Return a list of all top-level symbols with bound values, including those intended for internal use.

See also: `dump`. →index

**3.24.22 error : procedure/0 or more**

Usage: (`error` [`msgstr`] [`expr`] ...)

Raise an error, where `msgstr` and the optional expressions `expr...` work as in a call to `fmt`.

See also: `fmt`, `with-final`. →index

**3.24.23 error->str : procedure/1**

Usage: (`error->str` `datum`)=> `str`

Convert a special error value to a string.

See also: `*last-error*`, `error`, `error?`. →index

**3.24.24 error? : procedure/1**

Usage: (`error?` `datum`)=> `bool`

Return true if `datum` is a special error value, nil otherwise.

See also: `*last-error*`, `error->str`, `error`, `eof?`, `valid?`. →index



### 3.24.25 `eval` : procedure/1

Usage: (`eval` `expr`)=> `any`

Evaluate the expression `expr` in the Z3S5 Machine Lisp interpreter and return the result. The evaluation environment is the system's environment at the time of the call.

See also: `break`, `apply`. →index

### 3.24.26 `exit` : procedure/0 or more

Usage: (`exit` [`n`])

Immediately shut down the system and return OS host error code `n`. The shutdown is performed gracefully and exit hooks are executed.

See also: `.` →index

### 3.24.27 `expand-macros` : procedure/1

Usage: (`expand-macros` `expr`)=> `expr`

Expands the macros in `expr`. This is an ordinary function and will not work on already compiled expressions such as a function bound to a symbol. However, it can be used to expand macros in expressions obtained by `read`.

See also: `internalize`, `externalize`, `load-library`. →index

### 3.24.28 `expect` : macro/2

Usage: (`expect` `value` `given`)

Registers a test under the current test name that checks that `value` is returned by `given`. The test is only executed when (`run-selftest`) is executed.

See also: `expect-err`, `expect-ok`, `run-selftest`, `testing`. →index

### 3.24.29 `expect-err` : macro/1 or more

Usage: (`expect-err` `expr` ...)

Registers a test under the current test name that checks that `expr` produces an error.

See also: `expect`, `expect-ok`, `run-selftest`, `testing`. →index

**3.24.30 expect-false : macro/1 or more**

Usage: ([expect-false](#) [expr](#) ...)

Registers a test under the current test name that checks that [expr](#) is nil.

See also: [expect](#), [expect-ok](#), [run-selftest](#), [testing](#). →index

**3.24.31 expect-ok : macro/1 or more**

Usage: ([expect-err](#) [expr](#) ...)

Registers a test under the current test name that checks that [expr](#) does not produce an error.

See also: [expect](#), [expect-ok](#), [run-selftest](#), [testing](#). →index

**3.24.32 expect-true : macro/1 or more**

Usage: ([expect-true](#) [expr](#) ...)

Registers a test under the current test name that checks that [expr](#) is true (not nil).

See also: [expect](#), [expect-ok](#), [run-selftest](#), [testing](#). →index

**3.24.33 externalize : procedure/1**

Usage: ([externalize](#) [sym](#) [[nonce](#)])=> [sexpr](#)

Obtain an external representation of top-level symbol [sym](#). The optional [nonce](#) must be a value unique in each system zimage, in order to distinguish data from procedures.

See also: [can-externalize?](#), [externalize0](#), [current-zimage](#), [save-zimage](#), [load-zimage](#). →index

**3.24.34 externalize0 : procedure/1**

Usage: ([externalize0](#) [arg](#))=> [any](#)

Attempts to externalize [arg](#) but falls back to the internal expression if [arg](#) cannot be externalized. This procedure never fails but [can-externalize?](#) may be false for the result. This function is only used in miscellaneous printing. Use [externalize](#) to externalize expressions for writing to disk.

See also: [externalize](#), [can-externalize?](#). →index

**3.24.35 feature? : procedure/1**

Usage: (`feature? sym`)=> `bool`

Return true if the Lisp feature identified by symbol `sym` is available, nil otherwise.

See also: `*reflect*`, `on-feature`. →index

**3.24.36 find-missing-help-entries : procedure/0**

Usage: (`find-missing-help-entries`)=> `li`

Return a list of global symbols for which help entries are missing.

See also: `dump`, `dump-bindings`, `find-unneeded-help-entries`. →index

**3.24.37 find-unneeded-help-entries : procedure/0**

Usage: (`find-unneeded-help-entries`)=> `li`

Return a list of help entries for which no symbols are defined.

See also: `dump`, `dump-bindings`, `find-missing-help-entries`. →index

**Warning: This function returns false positives! Special forms like `setq` and `macro` are listed even though they clearly are useful and should have a help entry.**

**3.24.38 functional-arity : procedure/1**

Usage: (`functional-arity proc`)=> `int`

Return the arity of a functional `proc`.

See also: `functional?`, `functional-has-rest?`. →index

**3.24.39 functional-has-rest? : procedure/1**

Usage: (`functional-has-rest? proc`)=> `bool`

Return true if the functional `proc` has a `&rest` argument, nil otherwise.

See also: `functional?`, `functional-arity`. →index

**3.24.40 functional? : macro/1**

Usage: (`functional?` `arg`)=> `bool`

Return true if `arg` is either a builtin function, a closure, or a macro, nil otherwise. This is the right predicate for testing whether the argument is applicable and has an arity.

See also: `closure?`, `proc?`, `functional-arity`, `functional-has-rest?`. →index

**3.24.41 gensym : procedure/0**

Usage: (`gensym`)=> `sym`

Return a new symbol guaranteed to be unique during runtime.

See also: `nonce`. →index

**3.24.42 hook : procedure/1**

Usage: (`hook` `symbol`)

Lookup the internal hook number from a symbolic name.

See also: `*hooks*`, `add-hook`, `remove-hook`, `remove-hooks`. →index

**3.24.43 include : procedure/1**

Usage: (`include` `fi`)=> `any`

Evaluate the lisp file `fi` one expression after the other in the current environment.

See also: `read`, `write`, `open`, `close`. →index

**3.24.44 intern : procedure/1**

Usage: (`intern` `s`)=> `sym`

Create a new interned symbol based on string `s`.

See also: `gensym`, `str->sym`, `make-symbol`. →index

### 3.24.45 `internalize` : procedure/2

Usage: (`internalize` *arg* *nonce*)

Internalize an external representation of *arg*, using *nonce* for distinguishing between data and code that needs to be evaluated.

See also: `externalize`. →index

### 3.24.46 `intrinsic` : procedure/1

Usage: (`intrinsic` *sym*)=> *any*

Attempt to obtain the value that is intrinsically bound to *sym*. Use this function to express the intention to use the pre-defined builtin value of a symbol in the base language.

See also: `bind`, `unbind`. →index

**Warning:** This function currently only returns the binding but this behavior might change in future.

### 3.24.47 `intrinsic?` : procedure/1

Usage: (`intrinsic?` *x*)=> *bool*

Return true if *x* is an intrinsic built-in function, nil otherwise. Notice that this function tests the value and not that a symbol has been bound to the intrinsic.

See also: `functional?`, `macro?`, `closure?`. →index

**Warning:** What counts as an intrinsic or not may change from version to version. This is for internal use only.

### 3.24.48 `macro?` : procedure/1

Usage: (`macro?` *x*)=> *bool*

Return true if *x* is a macro, nil otherwise.

See also: `functional?`, `intrinsic?`, `closure?`, `functional-arity`, `functional-has-rest?`. →index

**3.24.49 make-symbol : procedure/1**

Usage: (`make-symbol` `s`)=> `sym`

Create a new symbol based on string `s`.

See also: `str->sym`. [→index](#)

**3.24.50 memstats : procedure/0**

Usage: (`memstats`)=> `dict`

Return a dict with detailed memory statistics for the system.

See also: `collect-garbage`. [→index](#)

**3.24.51 nonce : procedure/0**

Usage: (`nonce`)=> `str`

Return a unique random string. This is not cryptographically secure but the string satisfies reasonable GUID requirements.

See also: `externalize`, `internalize`. [→index](#)

**3.24.52 on-feature : macro/1 or more**

Usage: (`on-feature` `sym` `body` ...)=> `any`

Evaluate the expressions of `body` if the Lisp feature `sym` is supported by this implementation, do nothing otherwise.

See also: `feature?`, `*reflect*`. [→index](#)

**3.24.53 permission? : procedure/1**

Usage: (`permission?` `sym` [`default`])=> `bool`

Return true if the permission for `sym` is set, nil otherwise. If the permission flag is unknown, then `default` is returned. The default for `default` is nil.

See also: `permissions`, `set-permissions`, `when-permission`, `sys`. [→index](#)

### 3.24.54 `permissions` : procedure/0

Usage: (`permissions`)

Return a list of all active permissions of the current interpreter. Permissions are: `load-prelude` - load the init file on start; `load-user-init` - load the local user init on startup, file if present; `allow-unprotect` - allow the user to unprotect protected symbols (for redefining them); `allow-protect` - allow the user to protect symbols from redefinition or unbinding; `interactive` - make the session interactive, this is particularly used during startup to determine whether hooks are installed and feedback is given. Permissions have to generally be set or removed in careful combination with `revoke-permissions`, which redefines symbols and functions.

See also: `set-permissions`, `permission?`, `when-permission`, `sys.` →index

### 3.24.55 `pop-error-handler` : procedure/0

Usage: (`pop-error-handler`)=> `proc`

Remove the topmost error handler from the error handler stack and return it. For internal use only.

See also: `with-error-handler`. →index

### 3.24.56 `pop-finalizer` : procedure/0

Usage: (`pop-finalizer`)=> `proc`

Remove a finalizer from the finalizer stack and return it. For internal use only.

See also: `push-finalizer`, `with-final`. →index

### 3.24.57 `proc?` : macro/1

Usage: (`proc?` `arg`)=> `bool`

Return true if `arg` is a procedure, nil otherwise.

See also: `functional?`, `closure?`, `functional-arity`, `functional-has-rest?`. →index

### 3.24.58 `protect` : procedure/0 or more

Usage: (`protect` [`sym`] ...)

Protect symbols `sym`... against changes or rebinding. The symbols need to be quoted. This operation requires the permission `'allow-protect` to be set.

See also: `protected?`, `unprotect`, `dict-protect`, `dict-unprotect`, `dict-protected?`, `permissions`, `permission?`, `setq`, `bind`, `interpret`. →index

### 3.24.59 `protect-toplevel-symbols` : procedure/0

Usage: (`protect-toplevel-symbols`)

Protect all `toplevel` symbols that are not yet protected and aren't in the *mutable-toplevel-symbols* dict.

See also: `protected?`, `protect`, `unprotect`, `declare-unprotected`, `declare-volatile`, `when-permission?`, `dict-protect`, `dict-protected?`, `dict-unprotect`. →index

### 3.24.60 `protected?` : procedure/1

Usage: (`protected?` `sym`)

Return true if `sym` is protected, nil otherwise.

See also: `protect`, `unprotect`, `dict-unprotect`, `dict-protected?`, `permission`, `permission?`, `setq`, `bind`, `interpret`. →index

### 3.24.61 `push-error-handler` : procedure/1

Usage: (`push-error-handler` `proc`)

Push an error handler `proc` on the error handler stack. For internal use only.

See also: `with-error-handler`. →index

### 3.24.62 `push-finalizer` : procedure/1

Usage: (`push-finalizer` `proc`)

Push a finalizer procedure `proc` on the finalizer stack. For internal use only.

See also: `with-final`, `pop-finalizer`. →index



**3.24.63 read-eval-reply : procedure/0**

Usage: (`read-eval-reply`)

Start a new read-eval-reply loop.

See also: `end-input`, `sys.` →index

**Warning: Internal use only. This function might not do what you expect it to do.**

**3.24.64 remove-hook : procedure/2**

Usage: (`remove-hook hook id`)=> `bool`

Remove the symbolic or numeric `hook` with `id` and return true if the hook was removed, nil otherwise.

See also: `add-hook`, `remove-hooks`, `replace-hook`. →index

**3.24.65 remove-hook-internal : procedure/2**

Usage: (`remove-hook-internal hook id`)

Remove the hook with ID `id` from numeric `hook`.

See also: `remove-hook`. →index

**Warning: Internal use only.**

**3.24.66 remove-hooks : procedure/1**

Usage: (`remove-hooks hook`)=> `bool`

Remove all hooks for symbolic or numeric `hook`, return true if the hook exists and the associated procedures were removed, nil otherwise.

See also: `add-hook`, `remove-hook`, `replace-hook`. →index

**3.24.67 replace-hook : procedure/2**

Usage: (`replace-hook hook proc`)

Remove all hooks for symbolic or numeric `hook` and install the given `proc` as the only hook procedure.

See also: `add-hook`, `remove-hook`, `remove-hooks`. →index

**3.24.68 run-hook : procedure/1**

Usage: ([run-hook](#) [hook](#))

Manually run the hook, executing all procedures for the hook.

See also: [add-hook](#), [remove-hook](#). →index

**3.24.69 run-hook-internal : procedure/1 or more**

Usage: ([run-hook-internal](#) [hook](#) [[args](#)] ...)

Run all hooks for numeric hook ID [hook](#) with [args](#)... as arguments.

See also: [run-hook](#). →index

**Warning: Internal use only.**

**3.24.70 run-selftest : procedure/0**

Usage: ([run-selftest](#))

Run a self test of the Z3S5 Lisp system and report errors to standard output.

See also: [help](#), [testing](#). →index

**3.24.71 set-permissions : nil**

Usage: ([set-permissions](#) [li](#))

Set the permissions for the current interpreter. This will trigger an error when the permission cannot be set due to a security violation. Generally, permissions can only be downgraded (made more stringent) and never relaxed. See the information for [permissions](#) for an overview of symbolic flags.

See also: [permissions](#), [permission?](#), [when-permission](#), [sys](#). →index

**3.24.72 sleep : procedure/1**

Usage: ([sleep](#) [ms](#))

Halt the current task execution for [ms](#) milliseconds.

See also: [sleep-ns](#), [time](#), [now](#), [now-ns](#). →index

**3.24.73 sleep-ns : procedure/1**

Usage: (`sleep-ns` `n`)

Halt the current task execution for `n` nanoseconds.

See also: `sleep`, `time`, `now`, `now-ns`. →index

**3.24.74 sys-key? : procedure/1**

Usage: (`sys-key?` `key`)=> `bool`

Return true if the given sys key `key` exists, nil otherwise.

See also: `sys`, `setsys`. →index

**3.24.75 sysmsg : procedure/1**

Usage: (`sysmsg` `msg`)

Asynchronously display a system message string `msg` if in console or page mode, otherwise the message is logged.

See also: `sysmsg*`, `synout`, `synouty`, `out`, `outy`. →index

**3.24.76 sysmsg\* : procedure/1**

Usage: (`sysmsg*` `msg`)

Display a system message string `msg` if in console or page mode, otherwise the message is logged.

See also: `sysmsg`, `synout`, `synouty`, `out`, `outy`. →index

**3.24.77 testing:macro/1**

Usage: (`testing` `name`)

Registers the string `name` as the name of the tests that are next registered with `expect`.

See also: `expect`, `expect-err`, `expect-ok`, `run-selftest`. →index

**3.24.78 try : macro/2 or more**

Usage: (**try** (*finals* ...) *body* ...)

Evaluate the forms of the *body* and afterwards the forms in *finals*. If during the execution of *body* an error occurs, first all *finals* are executed and then the error is printed by the default error printer.

See also: *with-final*, *with-error-handler*. →index

**3.24.79 unprotect : procedure/0 or more**

Usage: (*unprotect* [*sym*] ...)

Unprotect symbols *sym*..., allowing mutation or rebinding them. The symbols need to be quoted. This operation requires the permission 'allow-unprotect to be set, or else an error is caused.

See also: *protect*, **protected?**, *dict-unprotect*, *dict-protected?*, *permissions*, *permission?*, *setq*, *bind*, *interpret*. →index

**3.24.80 unprotect-toplevel-symbols : procedure/0**

Usage: (*unprotect-toplevel-symbols*)

Attempts to unprotect all toplevel symbols.

See also: *protect-toplevel-symbols*, *protect*, *unprotect*, *declare-unprotected*. →index

**3.24.81 warn : procedure/1 or more**

Usage: (*warn* *msg* [*args*...])

Output the warning message *msg* in error colors. The optional *args* are applied to the message as in *fmt*. The message should not end with a newline.

See also: *error*. →index

**3.24.82 when-permission : macro/1 or more**

Usage: (*when-permission* *perm* *body* ...) => *any*

Execute the expressions in *body* if and only if the symbolic permission *perm* is available.

See also: *permission?*. →index

### 3.24.83 `with-colors` : procedure/3

Usage: (`with-colors` `textcolor` `backcolor` `proc`)

Execute `proc` for display side effects, where the default colors are set to `textcolor` and `backcolor`. These are color specifications like in `the-color`. After `proc` has finished or if an error occurs, the default colors are restored to their original state.

See also: `the-color`, `color`, `set-color`, `with-final`. →index

### 3.24.84 `with-error-handler` : macro/2 or more

Usage: (`with-error-handler` `handler` `body` ...)

Evaluate the forms of the `body` with error handler `handler` in place. The handler is a procedure that takes the error as argument and handles it. If an error occurs in `handler`, a default error handler is used. Handlers are only active within the same thread.

See also: `with-final`. →index

### 3.24.85 `with-final` : macro/2 or more

Usage: (`with-final` `finalizer` `body` ...)

Evaluate the forms of the `body` with the given finalizer as error handler. If an error occurs, then `finalizer` is called with that error and nil. If no error occurs, `finalizer` is called with nil as first argument and the result of evaluating all forms of `body` as second argument.

See also: `with-error-handler`. →index

## 3.25 Time & Date

This section lists functions that are time and date-related. Most of them use (`now`) and turn it into more human-readable form.

### 3.25.1 `date->epoch-ns` : procedure/7

Usage: (`date->epoch-ns` `Y` `M` `D` `h` `m` `s` `ns`)=> `int`

Return the Unix epoch nanoseconds based on the given year `Y`, month `M`, day `D`, hour `h`, minute `m`, seconds `s`, and nanosecond fraction of a second `ns`, as it is e.g. returned in a (`now`) datelist.

See also: [epoch-ns->datelist](#), [datestr->datelist](#), [datestr](#), [datestr\\*](#), [day-of-week](#), [week-of-date](#), [now](#). →index

### 3.25.2 [datelist->epoch-ns](#) : procedure/1

Usage: ([datelist->epoch-ns](#) [dateli](#))=> [int](#)

Convert a datelist to Unix epoch nanoseconds. This function uses the Unix nanoseconds from the 5th value of the second list in the datelist, as it is provided by functions like ([now](#)). However, if the Unix nanoseconds value is not specified in the list, it uses [date->epoch-ns](#) to convert to Unix epoch nanoseconds. Datelists can be incomplete. If the month is not specified, January is assumed. If the day is not specified, the 1st is assumed. If the hour is not specified, 12 is assumed, and corresponding defaults for minutes, seconds, and nanoseconds are 0.

See also: [date->epoch-ns](#), [datestr](#), [datestr\\*](#), [datestr->datelist](#), [epoch-ns->datelist](#), [now](#). →index

### 3.25.3 [datestr](#) : procedure/1

Usage: ([datestr](#) [datelist](#))=> [str](#)

Return datelist, as it is e.g. returned by ([now](#)), as a string in format YYYY-MM-DD HH:mm.

See also: [now](#), [datestr\\*](#), [datestr->datelist](#). →index

### 3.25.4 [datestr\\*](#) : procedure/1

Usage: ([datestr\\*](#) [datelist](#))=> [str](#)

Return the datelist, as it is e.g. returned by ([now](#)), as a string in format YYYY-MM-DD HH:mm:ss.nanoseconds.

See also: [now](#), [datestr](#), [datestr->datelist](#). →index

### 3.25.5 [datestr->datelist](#) : procedure/1

Usage: ([datestr->datelist](#) [s](#))=> [li](#)

Convert a date string in the format of [datestr](#) and [datestr\\*](#) into a date list as it is e.g. returned by ([now](#)).

See also: [datestr\\*](#), [datestr](#), [now](#). →index

### 3.25.6 day+ : procedure/2

Usage: (day+ date<sub>li</sub> n)=> date<sub>li</sub>

Adds *n* days to the given date *date<sub>li</sub>* in datelist format and returns the new datelist.

See also: sec+, minute+, hour+, week+, month+, year+, now. →index

### 3.25.7 day-of-week : procedure/3

Usage: (day-of-week Y M D)=> int

Return the day of week based on the date with year *Y*, month *M*, and day *D*. The first day number 0 is Sunday, the last day is Saturday with number 6.

See also: week-of-date, datestr->datelist, date->epoch-ns, epoch-ns->datelist, datestr, datestr\*, now. →index

### 3.25.8 epoch-ns->datelist : procedure/1

Usage: (epoch-ns->datelist ns)=> li

Return the date list in UTC time corresponding to the Unix epoch nanoseconds *ns*.

See also: date->epoch-ns, datestr->datelist, datestr, datestr\*, day-of-week, week-of-date, now. →index

### 3.25.9 hour+ : procedure/2

Usage: (hour+ date<sub>li</sub> n)=> date<sub>li</sub>

Adds *n* hours to the given date *date<sub>li</sub>* in datelist format and returns the new datelist.

See also: sec+, minute+, day+, week+, month+, year+, now. →index

### 3.25.10 minute+ : procedure/2

Usage: (minute+ date<sub>li</sub> n)=> date<sub>li</sub>

Adds *n* minutes to the given date *date<sub>li</sub>* in datelist format and returns the new datelist.

See also: sec+, hour+, day+, week+, month+, year+, now. →index

**3.25.11 month+ : procedure/2**

Usage: (month+ date*li* n)=> date*li*

Adds *n* months to the given date *date**li* in datelist format and returns the new datelist.

See also: *sec+*, *minute+*, *hour+*, *day+*, *week+*, *year+*, *now*. →index

**3.25.12 now : procedure/0**

Usage: (now)=> *li*

Return the current datetime in UTC format as a list of values in the form '((year month day weekday iso-week) (hour minute second nanosecond unix-nano-second)).

See also: *now-ns*, *datestr*, *time*, *date*->*epoch-ns*, *epoch-ns*->*datelist*. →index

**3.25.13 now-ms : procedure/0**

Usage: (now-*ms*)=> *num*

Return the relative system time as a call to (now-ns) but in milliseconds.

See also: *now-ns*, *now*. →index

**3.25.14 now-ns : procedure/0**

Usage: (now-*ns*)=> *int*

Return the current time in Unix nanoseconds.

See also: *now*, *time*. →index

**3.25.15 sec+ : procedure/2**

Usage: (sec+ date*li* n)=> date*li*

Adds *n* seconds to the given date *date**li* in datelist format and returns the new datelist.

See also: *minute+*, *hour+*, *day+*, *week+*, *month+*, *year+*, *now*. →index



### 3.25.16 `time : procedure/1`

Usage: `(time proc)`=> `int`

Return the time in nanoseconds that it takes to execute the procedure with no arguments `proc`.

See also: `now-ns`, `now`. →index

### 3.25.17 `week+ : procedure/2`

Usage: `(week+ dateli n)`=> `dateli`

Adds `n` weeks to the given date `dateli` in datelist format and returns the new datelist.

See also: `sec+`, `minute+`, `hour+`, `day+`, `month+`, `year+`, `now`. →index

### 3.25.18 `week-of-date : procedure/3`

Usage: `(week-of-date Y M D)`=> `int`

Return the week of the date in the year given by year `Y`, month `M`, and day `D`.

See also: `day-of-week`, `datestr->datelist`, `date->epoch-ns`, `epoch-ns->datelist`, `datestr`, `datestr*`, `now`. →index

### 3.25.19 `year+ : procedure/2`

Usage: `(month+ dateli n)`=> `dateli`

Adds `n` years to the given date `dateli` in datelist format and returns the new datelist.

See also: `sec+`, `minute+`, `hour+`, `day+`, `week+`, `month+`, `now`. →index

## 3.26 User Interface

This section lists miscellaneous user interface commands such as color for terminals.

### 3.26.1 `colors : dict`

Usage: `*colors*`

A global dict that maps default color names to color lists (r g b), (r g b a) or selectors for (color selector). This can be used with procedure `the-color` to translate symbolic names to colors.

See also: [the-color](#). [→index](#)

### 3.26.2 `color` : procedure/1

Usage: (`color sel`)=> (`r g b a`)

Return the color based on `sel`, which may be `'text` for the text color, `'back` for the background color, `'textarea` for the color of the text area, `'gfx` for the current graphics foreground color, and `'frame` for the frame color. In standard Z3S5 Lisp only `'text` and `'back` are available as selectors and implementations are free to ignore these.

See also: [set-color](#), [reset-color](#), [the-color](#), [with-colors](#). [→index](#)

### 3.26.3 `darken` : procedure/1

Usage: (`darken color [amount]`)=> (`r g b a`)

Return a darker version of `color`. The optional positive `amount` specifies the amount of darkening (0-255).

See also: [the-color](#), [\\*colors\\*](#), [lighten](#). [→index](#)

### 3.26.4 `lighten` : procedure/1

Usage: (`lighten color [amount]`)=> (`r g b a`)

Return a lighter version of `color`. The optional positive `amount` specifies the amount of lightening (0-255).

See also: [the-color](#), [\\*colors\\*](#), [darken](#). [→index](#)

### 3.26.5 `out` : procedure/1

Usage: (`out expr`)

Output `expr` on the console with current default background and foreground color.

See also: [outy](#), [synout](#), [synouty](#), [output-at](#). [→index](#)

### 3.26.6 outy : procedure/1

Usage: ([outy spec](#))

Output styled text specified in [spec](#). A specification is a list of lists starting with 'fg for foreground, 'bg for background, or 'text for unstyled text. If the list starts with 'fg or 'bg then the next element must be a color suitable for (the-color spec). Following may be a string to print or another color specification. If a list starts with 'text then one or more strings may follow.

See also: [\\*colors\\*](#), [the-color](#), [set-color](#), [color](#), [gfx.color](#), [output-at](#), [out](#). →index

### 3.26.7 random-color : procedure/0 or more

Usage: ([random-color](#) [[alpha](#)])

Return a random color with optional [alpha](#) value. If [alpha](#) is not specified, it is 255.

See also: [the-color](#), [\\*colors\\*](#), [darken](#), [lighten](#). →index

### 3.26.8 reset-color : procedure/0

Usage: ([reset-color](#))

Reset the 'text and 'back colors of the display to default values. These values are not specified in the color database and depend on the runtime implementation. Other colors like 'gfx or 'frame are not affected.

See also: [set-color](#), [color](#), [the-color](#), [with-colors](#). →index

### 3.26.9 set-color : procedure/1

Usage: ([set-color sel colorlist](#))

Set the color according to [sel](#) to the color [colorlist](#) of the form '(r g b a). See [color](#) for information about [sel](#).

See also: [color](#), [reset-color](#), [the-color](#), [with-colors](#). →index

### 3.26.10 synout : procedure/1

Usage: ([synout arg](#))

Like `out`, but enforcing a new input line afterwards. This needs to be used when outputting concurrently in a future or task.

See also: `out`, `outy`, `synouty`. →index

**Warning: Concurrent display output can lead to unexpected visual results and ought to be avoided.**

### 3.26.11 `the-color : procedure/1`

Usage: `(the-color colors-spec)=> (r g b a)`

Return the color list (r g b a) based on a color specification, which may be a color list (r g b), a color selector for (color selector) or a color name such as 'dark-blue.

See also: `*colors*`, `color`, `set-color`, `outy`. →index

### 3.26.12 `the-color-names : procedure/0`

Usage: `(the-color-names)=> li`

Return the list of color names in `colors`.

See also: `*colors*`, `the-color`. →index

## 3.27 Runtime System Images

The following functions provide functionality for saving, loading, and running of runtime system images to and from disk.

### 3.27.1 `current-zimage : procedure/0`

Usage: `(current-zimage [nonce])=> dict`

Obtain a dict of all toplevel bindings. If the `nonce` is provided, procedures are externalized as (nonce proc) to distinguish them from data. This function may use a lot of memory. Consider saving or loading zimages directly from disk instead. Notice that the dict is not the same format as the one used by `load-zimage` and `save-zimage`.

See also: `load-zimage`, `save-zimage`, `externalize`. →index

### 3.27.2 `declare-volatile` : procedure/1

Usage: (`declare-volatile` `sym`)

Declares `sym`, which has to be quoted, as a volatile toplevel symbol. Volatile toplevel symbols are neither saved to nor loaded from zimages.

See also: `save-zimage`, `load-zimage`, `declare-unprotected`. →index

### 3.27.3 `load-zimage` : procedure/1 or more

Usage: (`load-zimage` `fi`)=> `li`

Load the zimage file `fi`, if possible, and return a list containing information about the zimage after it has been loaded. If the zimage fails the semantic version check, then an error is raised.

See also: `save-zimage`, `run-zimage`, `zimage-loadable?`. →index

### 3.27.4 `read-zimage` : procedure/2

Usage: (`read-zimage` `in` `fi`)

Reads and evaluates the zimage in stream `in` from file `fi`. The file `fi` argument is used in error messages. This procedure raises errors when the zimage is malformed or the version check fails.

See also: `load-zimage`, `run-zimage`, `zimage-header`. →index

### 3.27.5 `run-zimage` : procedure/1 or more

Usage: (`run-zimage` `fi`)

Load the zimage file `fi` and start it at the designated entry point. Raises an error if the zimage version is not compatible or the zimage cannot be run.

See also: `load-zimage`, `save-zimage`, `zimage-runnable?`, `zimage-loadable?`. →index

### 3.27.6 `save-zimage` : procedure/1 or more

Usage: (`save-zimage` `min-version` `info` `entry-point` `fi`)=> `int`

Write the current state of the system as a zimage to file `fi`. If the file already exists, it is overwritten. The `min-version` argument designates the minimum system version required to load the zimage. The `info` argument should be a list whose first argument is a human-readable string explaining the

purpose of the zimage and remainder is user data. The `entry-point` is either nil or an expression that can be evaluated to start the zimage after it has been loaded with `run-zimage`.

See also: `load-zimage`, `current-zimage`, `dump`, `run-zimage`, `zimage-loadable?`, `zimage-runable?`, `externalize`. →index

### 3.27.7 `write-zimage` : procedure/4

Usage: `(write-zimage out min-version info entry-point)=> list`

Write the current state of the system as an zimage to stream `out`. The `min-version` argument designates the minimum system version required to load the zimage. The `info` argument should be a list whose first argument is a human-readable string explaining the purpose of the zimage and remainder is user data. The `entry-point` is either nil or an expression that can be evaluated to start the zimage after it has been loaded with `run-zimage`. The procedure returns a header with information of the zimage.

See also: `save-zimage`, `read-zimage`, `load-zimage`, `current-zimage`, `externalize`. →index

### 3.27.8 `zimage-header` : procedure/1

Usage: `(zimage-header fi)=> li`

Return the zimage header from file `fi`.

See also: `load-zimage`, `run-zimage`. →index

### 3.27.9 `zimage-loadable?` : procedure/1 or more

Usage: `(zimage-loadable? fi)`

Checks whether the file `fi` is loadable. This does not check whether the file actually is an zimage file, so you can only use this on readable lisp files.

See also: `zimage-runable?`, `load-zimage`, `save-zimage`, `current-zimage`. →index

### 3.27.10 `zimage-runable?` : procedure/1 or more

Usage: `(zimage-runable? [sel] fi)`

Returns the non-nil entry-point of the zimage if the the zimage in file `fi` can be run, nil otherwise.

See also: `load-zimage`, `zimage-loadable?`, `save-zimage`, `current-zimage`. →index

## 4 Complete Reference

### 4.1 % : procedure/2

Usage: (`%` `x` `y`)=> `num`

Compute the remainder of dividing number `x` by `y`.

See also: `mod`, `/`. →index →topic

### 4.2 \* : procedure/0 or more

Usage: (`*` [`args`] ...)=> `num`

Multiply all `args`. Special cases: `()` is 1 and `(x)` is `x`.

See also: `+`, `-`, `/`. →index →topic

### 4.3 colors : dict

Usage: `*colors*`

A global dict that maps default color names to color lists (r g b), (r g b a) or selectors for (color selector). This can be used with procedure `the-color` to translate symbolic names to colors.

See also: `the-color`. →index →topic

### 4.4 error-handler : dict

Usage: (`*error-handler*` `err`)

The global error handler dict that contains procedures which take an error and handle it. If an entry is nil, the default handler is used, which outputs the error using `error-printer`. The dict contains handlers based on concurrent thread IDs and ought not be manipulated directly.

See also: `*error-printer*`. →index →topic

### 4.5 \*error-printer\* : procedure/1

Usage: (`*error-printer*` `err`)

The global printer procedure which takes an error and prints it.

See also: `error`. →index →topic

#### 4.6 *help* : dict

Usage: `*help*`

Dict containing all help information for symbols.

See also: `help`, `defhelp`, `apropos`. →index →topic

#### 4.7 *hooks* : dict

Usage: `*hooks*`

A dict containing translations from symbolic names to the internal numeric representations of hooks.

See also: `hook`, `add-hook`, `remove-hook`, `remove-hooks`. →index

#### 4.8 *last-error* : sym

Usage: `*last-error*` => `str`

Contains the last error that has occurred.

See also: `*error-printer*`, `*error-handler*`. →index

**Warning: This may only be used for debugging! Do not use this for error handling, it will surely fail!** →topic

#### 4.9 *reflect* : symbol

Usage: `*reflect*` => `li`

The list of feature identifiers as symbols that this Lisp implementation supports.

See also: `feature?`, `on-feature`. →index →topic

#### 4.10 *+* : procedure/0 or more

Usage: `(+ [args] ...)` => `num`

Sum up all `args`. Special cases: `(+)` is 0 and `(+ x)` is `x`.

See also: `-`, `*`, `/`. →index →topic



#### 4.11 `-` : procedure/1 or more

Usage: `(- x [y1] [y2] ...)` => `num`

Subtract `y1, y2, ...`, from `x`. Special case: `(- x)` is `-x`.

See also: `+`, `*`, `/`. →index →topic

#### 4.12 `/` : procedure/1 or more

Usage: `(/ x y1 [y2] ...)` => `float`

Divide `x` by `y1`, then by `y2`, and so forth. The result is a float.

See also: `+`, `*`, `-`. →index →topic

#### 4.13 `/=` : procedure/2

Usage: `(/= x y)` => `bool`

Return true if number `x` is not equal to `y`, nil otherwise.

See also: `>`, `>=`, `<`, `<=`. →index →topic

#### 4.14 `10th` : procedure/1 or more

Usage: `(10th seq [default])` => `any`

Get the tenth element of a sequence or the optional `default`. If there is no such element and no default is provided, then an error is raised.

See also: `nth`, `nthdef`, `car`, `list-ref`, `array-ref`, `string-ref`, `1st`, `2nd`, `3rd`, `4th`, `5th`, `6th`, `7th`, `8th`, `9th`. →index →topic

#### 4.15 `1st` : procedure/1 or more

Usage: `(1st seq [default])` => `any`

Get the first element of a sequence or the optional `default`. If there is no such element and no default is provided, then an error is raised.

See also: `nth`, `nthdef`, `car`, `list-ref`, `array-ref`, `string-ref`, `2nd`, `3rd`, `4th`, `5th`, `6th`, `7th`, `8th`, `9th`, `10th`. →index →topic

#### 4.16 2nd : procedure/1 or more

Usage: (2nd seq [**default**])=> any

Get the second element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [3rd](#), [4th](#), [5th](#), [6th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index →topic

#### 4.17 3rd : procedure/1 or more

Usage: (3rd seq [**default**])=> any

Get the third element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [4th](#), [5th](#), [6th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index →topic

#### 4.18 4th : procedure/1 or more

Usage: (4th seq [**default**])=> any

Get the fourth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [5th](#), [6th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index →topic

#### 4.19 5th : procedure/1 or more

Usage: (5th seq [**default**])=> any

Get the fifth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [4th](#), [6th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index →topic

#### 4.20 6th : procedure/1 or more

Usage: (6th seq [default])=> any

Get the sixth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [4th](#), [5th](#), [7th](#), [8th](#), [9th](#), [10th](#). →index →topic

#### 4.21 7th : procedure/1 or more

Usage: (7th seq [default])=> any

Get the seventh element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [4th](#), [5th](#), [6th](#), [8th](#), [9th](#), [10th](#). →index →topic

#### 4.22 8th : procedure/1 or more

Usage: (8th seq [default])=> any

Get the eighth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [4th](#), [5th](#), [6th](#), [7th](#), [9th](#), [10th](#). →index →topic

#### 4.23 9th : procedure/1 or more

Usage: (9th seq [default])=> any

Get the ninth element of a sequence or the optional **default**. If there is no such element and no default is provided, then an error is raised.

See also: [nth](#), [nthdef](#), [car](#), [list-ref](#), [array-ref](#), [string-ref](#), [1st](#), [2nd](#), [3rd](#), [4th](#), [5th](#), [6th](#), [7th](#), [8th](#), [10th](#). →index →topic

#### 4.24 < : procedure/2

Usage: (`< x y`)=> `bool`

Return true if `x` is smaller than `y`.

See also: `<=`, `>=`, `>`. →index →topic

#### 4.25 <= : procedure/2

Usage: (`<= x y`)=> `bool`

Return true if `x` is smaller than or equal to `y`, nil otherwise.

See also: `>`, `<`, `>=`, `/=`. →index →topic

#### 4.26 = : procedure/2

Usage: (`= x y`)=> `bool`

Return true if number `x` equals number `y`, nil otherwise.

See also: `eq?`, `equal?`. →index →topic

#### 4.27 > : procedure/2

Usage: (`> x y`)=> `bool`

Return true if `x` is larger than `y`, nil otherwise.

See also: `<`, `>=`, `<=`, `/=`. →index →topic

#### 4.28 >= : procedure/2

Usage: (`>= x y`)=> `bool`

Return true if `x` is larger than or equal to `y`, nil otherwise.

See also: `>`, `<`, `<=`, `/=`. →index →topic

## 4.29 `abs` : procedure/1

Usage: `(abs x) => num`

Returns the absolute value of number `x`.

See also: `*`, `-`, `+`, `/`. [→index](#) [→topic](#)

## 4.30 `action` : class

Usage: `(new action <info-clause> <name-clause> <proc-clause> ...) => action`

The action class describes instances of actions that serve as plugins for the system hosting Z3S5 Lisp. Each action has a `name`, `prefix` and `info` string property and a unique `id`. Property `args` is an array that specifies the type of arguments of the action. This may be used by an implementation of `action.get-args`. The `proc` property must be a function taking the action and a task-id as argument and processing the action sequentially until it is completed or `task-recv` returns the 'stop signal. An action may store the result of computation in the `result` property, an error in the `error` property, and an arbitrary state in the `state` property. After processing or if an error occurs, `action.result` should be called so the host can process the result or error. The action system requires the implementation of procedures `action.start`, `action.progress`, `action.get-args`, and `action.result`. These are usually defined in the host system, for example in the Go implementation of an application using Z3S5 Lisp actions, and serve as callback functions from Lisp to Go.

See also: `action`, `action-stop`, `action.start`, `action.progress`, `action.get-args`, `action.result`. [→index](#) [→topic](#)

## 4.31 `action-start` : method

Usage: `(action-start action)`

Start `action`, which runs the action's `proc` in a task with the action and a task-id as argument. The `proc` of the `action` should periodically check for the 'stop signal using `task-recv` if the action should be cancellable, should call `action.progress` to report progress, `action.error` in case of an error, and `action.result` to report the result.

See also: `action`, `action-stop`, `action-start`, `action.progress`, `action.get-args`, `action.result`. [→index](#) [→topic](#)

## 4.32 `action-stop` : method

Usage: `(action-stop action)`

The stop method sends a 'stop signal to the action's running `proc`. It is up to the `proc` to check for the signal using `task-recv` and terminate the action gracefully.

See also: `action`, `action-stop`, `action-start`, `action.progress`, `action.get-args`, `action.result`. [→index](#) [→topic](#)

### 4.33 `activate-menu-last-submenu` : procedure/1

Usage: (`activate-menu-last-submenu` `menu`)=> `bool`

Find the last active menu item traversing through open submenus, and activate its submenu if one is found. Return true if a submenu was activated, nil otherwise.

See also: `deactivate-menu-last-submenu`, `new-menu`, `activate-menu-next`, `activate-menu-previous`. [→index](#) [→topic](#)

### 4.34 `activate-menu-next` : procedure/1

Usage: (`activate-menu-next` `menu`)

Activate the menu item following the currently active menu item, if there is any.

See also: `activate-menu-previous`, `new-menu`, `activate-menu-last-submenu`. [→index](#) [→topic](#)

### 4.35 `activate-menu-previous` : procedure/1

Usage: (`activate-menu-previous` `menu`)

Activate the menu item before the currently active menu item, if there is any.

See also: `activate-menu-next`, `new-menu`, `activate-menu-last-submenu`. [→index](#) [→topic](#)

### 4.36 `add-hook` : procedure/2

Usage: (`add-hook` `hook` `proc`)=> `id`

Add hook procedure `proc` which takes a list of arguments as argument under symbolic or numeric `hook` and return an integer hook `id` for this hook. If `hook` is not known, nil is returned.

See also: `remove-hook`, `remove-hooks`, `replace-hook`. [→index](#) [→topic](#)

### 4.37 `add-hook-internal` : procedure/2

Usage: `(add-hook-internal hook proc)`=> `int`

Add a procedure `proc` to hook with numeric ID `hook` and return this procedures hook ID. The function does not check whether the hook exists.

See also: `add-hook`. →index

**Warning: Internal use only.** →topic

### 4.38 `add-hook-once` : procedure/2

Usage: `(add-hook-once hook proc)`=> `id`

Add a hook procedure `proc` which takes a list of arguments under symbolic or numeric `hook` and return an integer hook `id`. If `hook` is not known, nil is returned.

See also: `add-hook`, `remove-hook`, `replace-hook`. →index →topic

### 4.39 `add1` : procedure/1

Usage: `(add1 n)`=> `num`

Add 1 to number `n`.

See also: `sub1`, `+`, `-`. →index →topic

### 4.40 `alist->dict` : procedure/1

Usage: `(alist->dict li)`=> `dict`

Convert an association list `li` into a dictionary. Note that the value will be the cdr of each list element, not the second element, so you need to use an alist with proper pairs '(a . b) if you want b to be a single value.

See also: `dict->alist`, `dict`, `dict->list`, `list->dict`. →index →topic

### 4.41 `alist?` : procedure/1

Usage: `(alist? li)`=> `bool`

Return true if `li` is an association list, nil otherwise. This also works for a-lists where each element is a pair rather than a full list.

See also: [assoc](#). →index →topic

#### 4.42 **and** : macro/0 or more

Usage: ([and](#) [expr1](#) [expr2](#) ...) => [any](#)

Evaluate [expr1](#) and if it is not nil, then evaluate [expr2](#) and if it is not nil, evaluate the next expression, until all expressions have been evaluated. This is a shortcut logical and.

See also: [or](#). →index →topic

#### 4.43 **append** : procedure/1 or more

Usage: ([append](#) [li1](#) [li2](#) ...) => [li](#)

Concatenate the lists given as arguments.

See also: [cons](#). →index →topic

#### 4.44 **append-form** : procedure/0

Usage: ([append-form](#) [form](#) [str](#) [canvas-object](#))

Append a new row to the bottom [form](#) consisting of a label [str](#) aligned with a [canvas-object](#), which may be an entry, button, etc.

See also: [new-form](#). →index →topic

#### 4.45 **apply** : procedure/2

Usage: ([apply](#) [proc](#) [arg](#)) => [any](#)

Apply function [proc](#) to argument list [arg](#).

See also: [functional?](#). →index →topic

#### 4.46 **apropos** : procedure/1

Usage: ([apropos](#) [sym](#)) => [#li](#)

Get a list of procedures and symbols related to [sym](#) from the help system.

See also: [defhelp](#), [help-entry](#), [help](#), [\\*help\\*](#). →index →topic



#### 4.47 `array` : procedure/0 or more

Usage: `(array [arg1] ...)=> array`

Create an array containing the arguments given to it.

See also: `array?`, `build-array`. [→index](#) [→topic](#)

#### 4.48 `array->list` : procedure/1

Usage: `(array->list arr)=> li`

Convert array `arr` into a list.

See also: `list->array`, `array`. [→index](#) [→topic](#)

#### 4.49 `array->str` : procedure/1

Usage: `(array-str arr)=> s`

Convert an array of unicode glyphs as integer values into a string. If the given sequence is not a valid UTF-8 sequence, an error is thrown.

See also: `str->array`. [→index](#) [→topic](#)

#### 4.50 `array-append` : procedure/2

Usage: `(array-append arr elem)=> array`

Append `elem` to the array `arr`. This function is destructive and mutates the array. Use `array-copy` if you need a copy.

See also: `array-ref`, `array-len`, `build-array`, `array-slice`, `array`, `array-copy`. [→index](#) [→topic](#)

#### 4.51 `array-copy` : procedure/1

Usage: `(array-copy arr)=> array`

Return a copy of `arr`.

See also: `array`, `array?`, `array-map!`, `array-pmap!`. [→index](#) [→topic](#)

#### 4.52 `array-exists?` : procedure/2

Usage: `(array-exists? arr pred)`=> `bool`

Return true if `pred` returns true for at least one element in array `arr`, nil otherwise.

See also: `exists?`, `forall?`, `list-exists?`, `str-exists?`, `seq?`. →index →topic

#### 4.53 `array-forall?` : procedure/2

Usage: `(array-forall? arr pred)`=> `bool`

Return true if predicate `pred` returns true for all elements of array `arr`, nil otherwise.

See also: `foreach`, `map`, `forall?`, `str-forall?`, `list-forall?`, `exists?`. →index →topic

#### 4.54 `array-foreach` : procedure/2

Usage: `(array-foreach arr proc)`

Apply `proc` to each element of array `arr` in order, for the side effects.

See also: `foreach`, `list-foreach`, `map`. →index →topic

#### 4.55 `array-len` : procedure/1

Usage: `(array-len arr)`=> `int`

Return the length of array `arr`.

See also: `len`. →index →topic

#### 4.56 `array-map!` : procedure/2

Usage: `(array-map! arr proc)`

Traverse array `arr` in unspecified order and apply `proc` to each element. This mutates the array.

See also: `array-walk`, `array-pmap!`, `array?`, `map`, `seq?`. →index →topic

#### 4.57 `array-pmap!` : procedure/2

Usage: `(array-pmap! arr proc)`

Apply `proc` in unspecified order in parallel to array `arr`, mutating the array to contain the value returned by `proc` each time. Because of the calling overhead for parallel execution, for many workloads `array-map!` might be faster if `proc` is very fast. If `proc` is slow, then `array-pmap!` may be much faster for large arrays on machines with many cores.

See also: `array-map!`, `array-walk`, `array?`, `map`, `seq?`. →index →topic

#### 4.58 `array-ref` : procedure/1

Usage: `(array-ref arr n)=> any`

Return the element of `arr` at index `n`. Arrays are 0-indexed.

See also: `array?`, `array`, `nth`, `seq?`. →index →topic

#### 4.59 `array-reverse` : procedure/1

Usage: `(array-reverse arr)=> array`

Create a copy of `arr` that reverses the order of all of its elements.

See also: `reverse`, `list-reverse`, `str-reverse`. →index →topic

#### 4.60 `array-set` : procedure/3

Usage: `(array-set arr idx value)`

Set the value at index `idx` in `arr` to `value`. Arrays are 0-indexed. This mutates the array.

See also: `array?`, `array`. →index →topic

#### 4.61 `array-slice` : procedure/3

Usage: `(array-slice arr low high)=> array`

Slice the array `arr` starting from `low` (inclusive) and ending at `high` (exclusive) and return the slice. This function is destructive and mutates the slice. Use `array-copy` if you need a copy.

See also: `array-ref`, `array-len`, `array-append`, `build-array`, `array`, `array-copy`. →index →topic

#### 4.62 `array-sort` : procedure/2

Usage: `(array-sort arr proc)` => `arr`

Destructively sorts array `arr` by using comparison proc `proc`, which takes two arguments and returns true if the first argument is smaller than the second argument, nil otherwise. The array is returned but it is not copied and modified in place by this procedure. The sorting algorithm is not guaranteed to be stable.

See also: `sort`. →index →topic

#### 4.63 `array-walk` : procedure/2

Usage: `(array-walk arr proc)`

Traverse the array `arr` from first to last element and apply `proc` to each element for side-effects. Function `proc` takes the index and the array element at that index as argument. If `proc` returns nil, then the traversal stops and the index is returned. If `proc` returns non-nil, traversal continues. If `proc` never returns nil, then the index returned is -1. This function does not mutate the array.

See also: `array-map!`, `array-pmap!`, `array?`, `map`, `seq?`. →index →topic

#### 4.64 `array?` : procedure/1

Usage: `(array? obj)` => `bool`

Return true if `obj` is an array, nil otherwise.

See also: `seq?`, `array`. →index →topic

#### 4.65 `ascii85->blob` : procedure/1

Usage: `(ascii85->blob str)` => `blob`

Convert the ascii85 encoded string `str` to a binary blob. This will raise an error if `str` is not a valid ascii85 encoded string.

See also: `blob->ascii85`, `base64->blob`, `str->blob`, `hex->blob`. →index →topic

#### 4.66 **assoc** : procedure/2

Usage: (`assoc` *key* *alist*)=> *li*

Return the sublist of *alist* that starts with *key* if there is any, nil otherwise. Testing is done with `equal?`. An association list may be of the form ((*key1* *value1*)(*key2* *value2*)...) or ((*key1* . *value1*) (*key2* . *value2*) ...)

See also: `assoc`, `assoc1`, `alist?`, `eq?`, `equal?`. →index →topic

#### 4.67 **assoc1** : procedure/2

Usage: (`assoc1` *sym* *li*)=> *any*

Get the second element in the first sublist in *li* that starts with *sym*. This is equivalent to (`cadr` (`assoc` *sym* *li*)).

See also: `assoc`, `alist?`. →index →topic

#### 4.68 **assq** : procedure/2

Usage: (`assq` *key* *alist*)=> *li*

Return the sublist of *alist* that starts with *key* if there is any, nil otherwise. Testing is done with `eq?`. An association list may be of the form ((*key1* *value1*)(*key2* *value2*)...) or ((*key1* . *value1*) (*key2* . *value2*) ...)

See also: `assoc`, `assoc1`, `eq?`, `alist?`, `equal?`. →index →topic

#### 4.69 **atom?** : procedure/1

Usage: (`atom?` *x*)=> *bool*

Return true if *x* is an atomic value, nil otherwise. Atomic values are numbers and symbols.

See also: `sym?`. →index →topic

#### 4.70 **base64->blob** : procedure/1

Usage: (`base64->blob` *str*)=> *blob*

Convert the base64 encoded string *str* to a binary blob. This will raise an error if *str* is not a valid base64 encoded string.

See also: `blob->base64`, `hex->blob`, `ascii85->blob`, `str->blob`. [→index](#) [→topic](#)

#### 4.71 `beep`: procedure/1

Usage: (`beep sel`)

Play a built-in system sound. The argument `sel` may be one of '(error start ready click okay confirm info).

See also: `set-volume`. [→index](#) [→topic](#)

#### 4.72 `bind`: procedure/2

Usage: (`bind sym value`)

Bind `value` to the global symbol `sym`. In contrast to `setq` both values need quoting.

See also: `setq`. [→index](#) [→topic](#)

#### 4.73 `bitand`: procedure/2

Usage: (`bitand n m`)=> `int`

Return the bitwise and of integers `n` and `m`.

See also: `bitxor`, `bitor`, `bitclear`, `bitshl`, `bitshr`. [→index](#) [→topic](#)

#### 4.74 `bitclear`: procedure/2

Usage: (`bitclear n m`)=> `int`

Return the bitwise and-not of integers `n` and `m`.

See also: `bitxor`, `bitand`, `bitor`, `bitshl`, `bitshr`. [→index](#) [→topic](#)

#### 4.75 `bitor`: procedure/2

Usage: (`bitor n m`)=> `int`

Return the bitwise or of integers `n` and `m`.

See also: `bitxor`, `bitand`, `bitclear`, `bitshl`, `bitshr`. [→index](#) [→topic](#)

#### 4.76 `bitshl`: procedure/2

Usage: (`bitshl` *n* *m*)=> **int**

Return the bitwise left shift of *n* by *m*.

See also: `bitxor`, `bitor`, `bitand`, `bitclear`, `bitshr`. →index →topic

#### 4.77 `bitshr`: procedure/2

Usage: (`bitshr` *n* *m*)=> **int**

Return the bitwise right shift of *n* by *m*.

See also: `bitxor`, `bitor`, `bitand`, `bitclear`, `bitshl`. →index →topic

#### 4.78 `bitxor`: procedure/2

Usage: (`bitxor` *n* *m*)=> **int**

Return the bitwise exclusive or value of integers *n* and *m*.

See also: `bitand`, `bitor`, `bitclear`, `bitshl`, `bitshr`. →index →topic

#### 4.79 `blob->ascii85`: procedure/1 or more

Usage: (`blob->ascii85` *b* [*start*] [*end*])=> **str**

Convert the blob *b* to an ascii85 encoded string. If the optional *start* and *end* are provided, then only bytes from *start* (inclusive) to *end* (exclusive) are converted.

See also: `blob->hex`, `blob->str`, `blob->base64`, `valid?`, `blob?`. →index →topic

#### 4.80 `blob->base64`: procedure/1 or more

Usage: (`blob->base64` *b* [*start*] [*end*])=> **str**

Convert the blob *b* to a base64 encoded string. If the optional *start* and *end* are provided, then only bytes from *start* (inclusive) to *end* (exclusive) are converted.

See also: `base64->blob`, `valid?`, `blob?`, `blob->str`, `blob->hex`, `blob->ascii85`. →index →topic

#### 4.81 `blob->hex` : procedure/1 or more

Usage: `(blob->hex b [start] [end])=> str`

Convert the blob `b` to a hexadecimal string of byte values. If the optional `start` and `end` are provided, then only bytes from `start` (inclusive) to `end` (exclusive) are converted.

See also: `hex->blob`, `str->blob`, `valid?`, `blob?`, `blob->base64`, `blob->ascii85`. [→index](#) [→topic](#)

#### 4.82 `blob->str` : procedure/1 or more

Usage: `(blob->str b [start] [end])=> str`

Convert blob `b` into a string. Notice that the string may contain binary data that is not suitable for displaying and does not represent valid UTF-8 glyphs. If the optional `start` and `end` are provided, then only bytes from `start` (inclusive) to `end` (exclusive) are converted.

See also: `str->blob`, `valid?`, `blob?`. [→index](#) [→topic](#)

#### 4.83 `blob-chksum` : procedure/1 or more

Usage: `(blob-chksum b [start] [end])=> blob`

Return the checksum of the contents of blob `b` as new blob. The checksum is cryptographically secure. If the optional `start` and `end` are provided, then only the bytes from `start` (inclusive) to `end` (exclusive) are checksummed.

See also: `fchksum`, `blob-free`. [→index](#) [→topic](#)

#### 4.84 `blob-equal?` : procedure/2

Usage: `(blob-equal? b1 b2)=> bool`

Return true if `b1` and `b2` are equal, nil otherwise. Two blobs are equal if they are either both invalid, both contain no valid data, or their contents contain exactly the same binary data.

See also: `str->blob`, `blob->str`, `blob-free`. [→index](#) [→topic](#)



#### 4.85 `blob-free` : procedure/1

Usage: `(blob-free b)`

Frees the binary data stored in blob `b` and makes the blob invalid.

See also: `make-blob`, `valid?`, `str->blob`, `blob->str`, `blob-equal?`. [→index](#) [→topic](#)

#### 4.86 `blob?` : procedure/1

Usage: `(blob? obj)=> bool`

Return true if `obj` is a binary blob, nil otherwise.

See also: `blob->ascii85`, `blob->base64`, `blob->hex`, `blob->str`, `blob-free`, `blob-chksum`, `blob-equal?`, `valid?`. [→index](#) [→topic](#)

#### 4.87 `bool?` : procedure/1

Usage: `(bool? datum)=> bool`

Return true if `datum` is either true or nil. Note: This predicate only exists for type-completeness and you should never use it as part of testing whether something is true or false - per convention, a value is true if it is non-nil and not when it is true, which is the special boolean value this predicate tests in addition to nil.

See also: `null?`, `not`. [→index](#) [→topic](#)

#### 4.88 `bound?` : macro/1

Usage: `(bound? sym)=> bool`

Return true if a value is bound to the symbol `sym`, nil otherwise.

See also: `bind`, `setq`. [→index](#) [→topic](#)

#### 4.89 `boxed?` : procedure/1

Usage: `(boxed? x)=> bool`

Return true if `x` is a boxed value, nil otherwise. Boxed values are special objects that are special in the system and sometimes cannot be garbage collected.

See also: [type-of](#), [num?](#), [str?](#), [sym?](#), [list?](#), [array?](#), [macro?](#), [closure?](#), [intrinsic?](#), [eof?](#).  
→index →topic

#### 4.90 **build-array**: procedure/2

Usage: ([build-array](#) *n* *init*)=> *array*

Create an array containing *n* elements with initial value *init*.

See also: [array](#), [array?](#), [array-slice](#), [array-append](#), [array-copy](#). →index →topic

#### 4.91 **build-list**: procedure/2

Usage: ([build-list](#) *n* *proc*)=> *list*

Build a list with *n* elements by applying *proc* to the counter *n* each time.

See also: [list](#), [list?](#), [map](#), [foreach](#). →index →topic

#### 4.92 **caaar**: procedure/1

Usage: ([caaar](#) *x*)=> *any*

Equivalent to ([car](#) ([car](#) ([car](#) *x*))).

See also: [car](#), [cdr](#), [caar](#), [cadr](#), [cdar](#), [cddr](#), [caadr](#), [cadar](#), [caddr](#), [cdaar](#), [cdadr](#), [cddar](#), [cdddr](#),  
[nth](#), [1st](#), [2nd](#), [3rd](#). →index →topic

#### 4.93 **caadr**: procedure/1

Usage: ([caadr](#) *x*)=> *any*

Equivalent to ([car](#) ([car](#) ([cdr](#) *x*))).

See also: [car](#), [cdr](#), [caar](#), [cadr](#), [cdar](#), [cddr](#), [caaar](#), [cadar](#), [caddr](#), [cdaar](#), [cdadr](#), [cddar](#), [cdddr](#),  
[nth](#), [1st](#), [2nd](#), [3rd](#). →index →topic

#### 4.94 **caar**: procedure/1

Usage: ([caar](#) *x*)=> *any*

Equivalent to ([car](#) ([car](#) *x*)).

See also: [car](#), [cdr](#), [cadr](#), [cdar](#), [cddr](#), [caaar](#), [caadr](#), [cadar](#), [caddr](#), [cdaar](#), [cdadr](#), [cddar](#), [cdddr](#), [nth](#), [1st](#), [2nd](#), [3rd](#). [→index](#) [→topic](#)

#### 4.95 **cadar** : procedure/1

Usage: ([cadar](#) [x](#))=> [any](#)

Equivalent to ([car](#) ([cdr](#) ([car](#) [x](#)))).

See also: [car](#), [cdr](#), [caar](#), [cadr](#), [cdar](#), [cddr](#), [caaar](#), [caadr](#), [cadar](#), [caddr](#), [cdaar](#), [cdadr](#), [cddar](#), [cdddr](#), [nth](#), [1st](#), [2nd](#), [3rd](#). [→index](#) [→topic](#)

#### 4.96 **caddr** : procedure/1

Usage: ([caddr](#) [x](#))=> [any](#)

Equivalent to ([car](#) ([cdr](#) ([cdr](#) [x](#)))).

See also: [car](#), [cdr](#), [caar](#), [cadr](#), [cdar](#), [cddr](#), [caaar](#), [caadr](#), [cadar](#), [caddr](#), [cdaar](#), [cdadr](#), [cddar](#), [cdddr](#), [nth](#), [1st](#), [2nd](#), [3rd](#). [→index](#) [→topic](#)

#### 4.97 **cadr** : procedure/1

Usage: ([cadr](#) [x](#))=> [any](#)

Equivalent to ([car](#) ([cdr](#) [x](#))).

See also: [car](#), [cdr](#), [caar](#), [cdar](#), [cddr](#), [caaar](#), [caadr](#), [cadar](#), [caddr](#), [cdaar](#), [cdadr](#), [cddar](#), [cdddr](#), [nth](#), [1st](#), [2nd](#), [3rd](#). [→index](#) [→topic](#)

#### 4.98 **call-method** : procedure/3

Usage: ([call-method](#) [obj](#) [mname](#) [args](#))=> [any](#)

Execute method [mname](#) of object [obj](#) with additional arguments in list [args](#). The first argument in the method call is always [obj](#) itself.

See also: [defmethod](#), [defclass](#), [new](#), [isa?](#), [class-of](#). [→index](#) [→topic](#)

### 4.99 `call-super` : procedure/3

Usage: (`call-super` `obj` `mname` `args`)=> `any`

Execute method `mname` of the first superclass of `obj` that has a method with that name.

See also: `call-method`, `supers`. →index →topic

### 4.100 `can-externalize?` : procedure/1

Usage: (`can-externalize?` `datum`)=> `bool`

Recursively determines if `datum` can be externalized and returns true in this case, nil otherwise.

See also: `externalize`, `externalize0`. →index →topic

### 4.101 `car` : procedure/1

Usage: (`car` `li`)=> `any`

Get the first element of a list or pair `li`, an error if there is not first element.

See also: `list`, `list?`, `pair?`. →index →topic

### 4.102 `case` : macro/2 or more

Usage: (`case` `expr` (`clause1` ... `clausen`))=> `any`

Standard case macro, where you should use `t` for the remaining alternative. Example: (`case` (`get` `dict` 'key) ((`a` `b`) (`out` "a or b"))(`t` (`out` "something else!"))).

See also: `cond`. →index →topic

### 4.103 `ccmp` : macro/2

Usage: (`ccmp` `sym` `value`)=> `int`

Compare the integer value of `sym` with the integer `value`, return 0 if `sym` = `value`, -1 if `sym` < `value`, and 1 if `sym` > `value`. This operation is synchronized between tasks and futures.

See also: `cinc!`, `cdec!`, `cwait`, `cst!`. →index →topic

**4.104 cdaar : procedure/1**

Usage: (`cdaar` `x`)=> `any`

Equivalent to (`cdr` (`car` (`car` `x`))).

See also: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index →topic

**4.105 cdadr : procedure/1**

Usage: (`cdadr` `x`)=> `any`

Equivalent to (`cdr` (`car` (`cdr` `x`))).

See also: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index →topic

**4.106 cdar : procedure/1**

Usage: (`cdar` `x`)=> `any`

Equivalent to (`cdr` (`car` `x`)).

See also: `car`, `cdr`, `caar`, `cadr`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index →topic

**4.107 cddar : procedure/1**

Usage: (`cddar` `x`)=> `any`

Equivalent to (`cdr` (`cdr` (`car` `x`))).

See also: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index →topic

**4.108 cdddr : procedure/1**

Usage: (`cdddr` `x`)=> `any`

Equivalent to (`cdr` (`cdr` (`cdr` `x`))).

See also: `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`. →index →topic

#### 4.109 **cddr** : procedure/1

Usage: (**cddr** *x*)=> *any*

Equivalent to (**cdr** (**cdr** *x*)).

See also: **car**, **cdr**, **caar**, **cadr**, **cdar**, **caaar**, **caadr**, **cadar**, **caddr**, **cdaar**, **cdadr**, **cddar**, **cdddr**, **nth**, **1st**, **2nd**, **3rd**. →index →topic

#### 4.110 **cdec!** : macro/1

Usage: (**cdec!** *sym*)=> **int**

Decrease the integer value stored in top-level symbol *sym* by 1 and return the new value. This operation is synchronized between tasks and futures.

See also: **cinc!**, **cwait**, **ccmp**, **cst!**. →index →topic

#### 4.111 **cdr** : procedure/1

Usage: (**cdr** *li*)=> *any*

Get the rest of a list *li*. If the list is proper, the cdr is a list. If it is a pair, then it may be an element. If the list is empty, nil is returned.

See also: **car**, **list**, **list?**, **pair?**. →index →topic

#### 4.112 **center-window-on-screen** : procedure/1

Usage: (**center-window-on-screen** *window*)

As the name implies, this function centers the window on the screen.

See also: **set-window-full-screen**, **window-full-screen?**. →index →topic

#### 4.113 **change-action-prefix** : procedure/2

Usage: (**change-action-prefix** *id* **new-prefix**)=> **bool**

Change the prefix of a registered action with given *id*, or change the prefix of action given by *id*, to **new-prefix**. If the operation succeeds, it returns true, otherwise it returns nil.

See also: **change-all-action-prefixes**, **rename-action**, **get-action**, **action?**, **action**. →index →topic

#### 4.114 `change-all-action-prefixes` : procedure/2

Usage: (`change-all-action-prefixes` `old-prefix` `new-prefix`)

Change the prefixes of all registered actions with `old-prefix` to `new-prefix`.

See also: `change-action-prefix`, `rename-action`, `get-action`, `register-action`, `action?`, `action`. →index →topic

#### 4.115 `char->str` : procedure/1

Usage: (`char->str` `n`)=> `str`

Return a string containing the unicode char based on integer `n`.

See also: `str->char`. →index →topic

#### 4.116 `chars` : procedure/1

Usage: (`chars` `str`)=> `dict`

Return a charset based on `str`, i.e., dict with the chars of `str` as keys and true as value.

See also: `dict`, `get`, `set`, `contains`. →index →topic

#### 4.117 `chars->str` : procedure/1

Usage: (`chars->str` `a`)=> `str`

Convert an array of UTF-8 rune integers `a` into a UTF-8 encoded string.

See also: `str->runes`, `str->char`, `char->str`. →index →topic

#### 4.118 `cinc!` : macro/1

Usage: (`cinc!` `sym`)=> `int`

Increase the integer value stored in top-level symbol `sym` by 1 and return the new value. This operation is synchronized between tasks and futures.

See also: `cdec!`, `cwait`, `ccmp`, `cst!`. →index →topic

#### 4.119 `class-name` : procedure/1

Usage: (`class-name` `c`)=> `sym`

Return the name of a class `c`. An error occurs if `c` is not a valid class.

See also: `class?`, `isa?`. →index →topic

#### 4.120 `class-of` : procedure/1

Usage: (`class-of` `obj`)=> `class` or `nil`

Return the class of object `obj`, nil if `obj` is not a valid object array.

See also: `new`, `isa?`. →index →topic

#### 4.121 `class?` : procedure/1

Usage: (`class?` `c`)=> `bool`

Return true if `c` is a class array (not a name for a class!), nil otherwise.

See also: `object?`, `isa?`. →index →topic

#### 4.122 `close` : procedure/1

Usage: (`close` `p`)

Close the port `p`. Calling close twice on the same port should be avoided.

See also: `open`, `stropen`. →index →topic

#### 4.123 `close-gui` : procedure/0

Usage: (`close-gui`)

Close the GUI, freeing all resources associated with it. After this function has been called, no further GUI functions can be used.

See also: `forget-gui-object`, `close-window`. →index →topic



#### 4.124 `close-window` : procedure/1

Usage: (`close-window` `window`)

Closes `window` and delete it from internal storage. This frees window resources. It cannot be re-used after this operation. Use `window-hide` if you want to close/hide a window only temporarily. Notice that unlike in `Fyne`, there is no mechanism to close an application automatically after its main window has been closed.

See also: `hide-window`. [→index](#) [→topic](#)

#### 4.125 `closure?` : procedure/1

Usage: (`closure?` `x`)=> `bool`

Return true if `x` is a closure, nil otherwise. Use `function?` for texting whether `x` can be executed.

See also: `functional?`, `macro?`, `intrinsic?`, `functional-arity`, `functional-has-rest?`. [→index](#) [→topic](#)

#### 4.126 `collect-garbage` : procedure/0 or more

Usage: (`collect-garbage` [`sort`])

Force a garbage-collection of the system's memory. If `sort` is 'normal, then only a normal incremental garbage collection is performed. If `sort` is 'total, then the garbage collection is more thorough and the system attempts to return unused memory to the host OS. Default is 'normal.

See also: `memstats`. [→index](#)

**Warning: There should rarely be a use for this. Try to use less memory-consuming data structures instead.** [→topic](#)

#### 4.127 `color` : procedure/1

Usage: (`color` `sel`)=> (`r` `g` `b` `a`)

Return the color based on `sel`, which may be 'text for the text color, 'back for the background color, 'textarea for the color of the text area, 'gfx for the current graphics foreground color, and 'frame for the frame color. In standard Z3S5 Lisp only 'text and 'back are available as selectors and implementations are free to ignore these.

See also: `set-color`, `reset-color`, `the-color`, `with-colors`. [→index](#) [→topic](#)

### 4.128 cons : procedure/2

Usage: (cons a b)=> pair

Cons two values into a pair. If *b* is a list, the result is a list. Otherwise the result is a pair.

See also: [cdr](#), [car](#), [list?](#), [pair?](#). →index →topic

### 4.129 cons? : procedure/1

Usage: (cons? x)=> bool

return true if *x* is not an atom, nil otherwise.

See also: [atom?](#). →index →topic

### 4.130 copy-record : procedure/1

Usage: (copy-record r)=> record

Creates a non-recursive, shallow copy of record *r*.

See also: [record?](#). →index →topic

### 4.131 count-partitions : procedure/2

Usage: (count-partitions m k)=> int

Return the number of partitions for dividing *m* items into parts of size *k* or less, where the size of the last partition may be less than *k* but the remaining ones have size *k*.

See also: [nth-partition](#), [get-partitions](#). →index →topic

### 4.132 cpunum : procedure/0

Usage: (cpunum)

Return the number of cpu cores of this machine.

See also: [sys](#). →index

**Warning: This function also counts virtual cores on the emulator. The original Z3S5 machine did not have virtual cpu cores.** →topic

### 4.133 `cst!` : procedure/2

Usage: (`cst!` `sym` `value`)

Set the value of `sym` to integer `value`. This operation is synchronized between tasks and futures.

See also: `cinc!`, `cdec!`, `ccmp`, `cwait`. [→index](#) [→topic](#)

### 4.134 `current-error-handler` : procedure/0

Usage: (`current-error-handler`)=> `proc`

Return the current error handler, a default if there is none.

See also: `default-error-handler`, `push-error-handler`, `pop-error-handler`, `*current-error-handler*`, `*current-error-continuation*`. [→index](#) [→topic](#)

### 4.135 `current-zimage` : procedure/0

Usage: (`current-zimage` [`nonce`])=> `dict`

Obtain a dict of all toplevel bindings. If the `nonce` is provided, procedures are externalized as (`nonce` `proc`) to distinguish them from data. This function may use a lot of memory. Consider saving or loading zimages directly from disk instead. Notice that the dict is not the same format as the one used by `load-zimage` and `save-zimage`.

See also: `load-zimage`, `save-zimage`, `externalize`. [→index](#) [→topic](#)

### 4.136 `cwait` : procedure/3

Usage: (`cwait` `sym` `value` `timeout`)

Wait until integer counter `sym` has `value` or `timeout` milliseconds have passed. If `timeout` is 0, then this routine might wait indefinitely. This operation is synchronized between tasks and futures.

See also: `cinc!`, `cdec!`, `ccmp`, `cst!`. [→index](#) [→topic](#)

### 4.137 `darken` : procedure/1

Usage: (`darken` `color` [`amount`])=> (`r` `g` `b` `a`)

Return a darker version of `color`. The optional positive `amount` specifies the amount of darkening (0-255).

See also: [the-color](#), [\\*colors\\*](#), [lighten](#). [→index](#) [→topic](#)

#### 4.138 [date->epoch-ns](#) : procedure/7

Usage: ([date->epoch-ns](#) *Y M D h m s ns*)=> **int**

Return the Unix epoch nanoseconds based on the given year *Y*, month *M*, day *D*, hour *h*, minute *m*, seconds *s*, and nanosecond fraction of a second *ns*, as it is e.g. returned in a (now) datelist.

See also: [epoch-ns->datelist](#), [datestr->datelist](#), [datestr](#), [datestr\\*](#), [day-of-week](#), [week-of-date](#), [now](#). [→index](#) [→topic](#)

#### 4.139 [datelist->epoch-ns](#) : procedure/1

Usage: ([datelist->epoch-ns](#) *dateli*)=> **int**

Convert a datelist to Unix epoch nanoseconds. This function uses the Unix nanoseconds from the 5th value of the second list in the datelist, as it is provided by functions like (now). However, if the Unix nanoseconds value is not specified in the list, it uses [date->epoch-ns](#) to convert to Unix epoch nanoseconds. Datelists can be incomplete. If the month is not specified, January is assumed. If the day is not specified, the 1st is assumed. If the hour is not specified, 12 is assumed, and corresponding defaults for minutes, seconds, and nanoseconds are 0.

See also: [date->epoch-ns](#), [datestr](#), [datestr\\*](#), [datestr->datelist](#), [epoch-ns->datelist](#), [now](#). [→index](#) [→topic](#)

#### 4.140 [datestr](#) : procedure/1

Usage: ([datestr](#) *datelist*)=> **str**

Return datelist, as it is e.g. returned by (now), as a string in format YYYY-MM-DD HH:mm.

See also: [now](#), [datestr\\*](#), [datestr->datelist](#). [→index](#) [→topic](#)

#### 4.141 [datestr\\*](#) : procedure/1

Usage: ([datestr\\*](#) *datelist*)=> **str**

Return the datelist, as it is e.g. returned by (now), as a string in format YYYY-MM-DD HH:mm:ss.nanoseconds.

See also: [now](#), [datestr](#), [datestr->datelist](#). [→index](#) [→topic](#)

#### 4.142 `datestr->datelist` : procedure/1

Usage: `(datestr->datelist s)=> li`

Convert a date string in the format of `datestr` and `datestr*` into a date list as it is e.g. returned by `(now)`.

See also: `datestr*`, `datestr`, `now`. →index →topic

#### 4.143 `day+ : procedure/2`

Usage: `(day+ dateli n)=> dateli`

Adds `n` days to the given date `dateli` in datelist format and returns the new datelist.

See also: `sec+`, `minute+`, `hour+`, `week+`, `month+`, `year+`, `now`. →index →topic

#### 4.144 `day-of-week` : procedure/3

Usage: `(day-of-week Y M D)=> int`

Return the day of week based on the date with year `Y`, month `M`, and day `D`. The first day number 0 is Sunday, the last day is Saturday with number 6.

See also: `week-of-date`, `datestr->datelist`, `date->epoch-ns`, `epoch-ns->datelist`, `datestr`, `datestr*`, `now`. →index →topic

#### 4.145 `db.blob` : procedure/2

Usage: `(db.blob db-result n)=> fl`

Get the content of column `n` in `db-result` as blob. A blob is a boxed memory area holding binary data.

See also: `db.str`. →index →topic

#### 4.146 `db.close` : procedure/1

Usage: `(db.close db)`

Close the database `db`.

See also: `db.open`, `db.open*`, `db.exec`, `db.query`. →index →topic

#### 4.147 **db.close-result** : procedure/1

Usage: (`db.close-result db-result`)

Close the `db-result`. It is invalid afterwards. This should be done to avoid memory leaks after the result has been used.

See also: `db.reset`, `db.step`, `db.close`. →index →topic

#### 4.148 **db.exec** : procedure/2 or more

Usage: (`db.exec db stmt [args] ...`)

Execute the SQL statement `stmt` in database `db`, binding any optional `args` to the open variable slots in it. This function does not return anything, use `db.query` to execute a query that returns rows as result.

See also: `db.query`, `db.open`, `db.close`, `db.open*`. →index →topic

#### 4.149 **db.float** : procedure/2

Usage: (`db.float db-result n`)=> `fl`

Get the content of column `n` in `db-result` as float.

See also: `db.int`, `db.str`. →index →topic

#### 4.150 **db.int** : procedure/2

Usage: (`db.int db-result n`)=> `int`

Get the content of column `n` in `db-result` as integer.

See also: `db.float`, `db.str`, `db.blob`. →index →topic

#### 4.151 **db.open** : procedure/1

Usage: (`db.open fi`)=> `db`

Opens an sqlite3 DB or creates a new, empty database at file path `fi`.

See also: `db.close`, `db.exec`, `db.query`. →index →topic

#### 4.152 `db.open*` : procedure/1

Usage: `(db.open* sel)=> db`

Open a temporary database if `sel` is 'temp or an in-memory database if `sel` is 'mem.

See also: `db.open`, `db.close`, `db.exec`, `db.query`. →index →topic

#### 4.153 `db.query` : procedure/2 or more

Usage: `(db.query db stmt [args] ...)=> db-result`

Query `db` with SQL statement `stmt`, binding any optional `args` to the open variable slots in it. This function returns a `db-result` that can be used to loop through rows with `db.step` and obtain columns in them using the various accessor methods.

See also: `db.exec`, `db.step`, `db.int`, `db.cname`, `db.float`, `db.str`, `db.expr`, `db.blob`. →index →topic

#### 4.154 `db.result-column-count` : procedure/1

Usage: `(db.result-column-count db-result)=> int`

Get the number of columns in the rows of `db-result`.

See also: `db.result-columns`. →index →topic

#### 4.155 `db.result-columns` : procedure/1

Usage: `(db.result-columns db-result)=> li`

Get a list of column specifications for `db-result`, each consisting of a list with the column name and the column type as string, as these were provided to the query. Since queries support automatic type conversions, this need not reflect the column types in the database schema.

See also: `db.result-column-count`. →index →topic

#### 4.156 `db.row` : procedure/1

Usage: `(db.row db-result)=> li`

Return all columns of the current row in `db-result` as list. They have the respective base types INT, FLOAT, BLOB, and TEXT.

See also: `db.rows`. [→index](#) [→topic](#)

#### 4.157 `db.step : procedure/1`

Usage: `(db.step db-result)=> bool`

Obtain the next result row in `db-result` and return true, or return nil if there is no more row in the result.

See also: `db.query`, `db.row`, `db.rows`. [→index](#) [→topic](#)

#### 4.158 `db.str : procedure/2`

Usage: `(db.str db-result n)=> str`

Get the content of column `n` in `db-result` as string.

See also: `db.blob`, `db.int`, `db.float`. [→index](#) [→topic](#)

#### 4.159 `deactivate-menu-child : procedure/1`

Usage: `(deactivate-menu-child menu)`

Deactivate the currently active menu item and close its submenu if there is one.

See also: `activate-menu-last-submenu`, `activate-menu-next`, `activate-menu-previous`, `new-menu`. [→index](#) [→topic](#)

#### 4.160 `deactivate-menu-last-submenu : procedure/1`

Usage: `(deactivate-menu-last-submenu menu)`

Traverse the menu and deactivate the last open submenu found.

See also: `activate-menu-last-submenu`, `activate-menu-next`, `activate-menu-previous`, `new-menu`. [→index](#) [→topic](#)

#### 4.161 `declare-volatile : procedure/1`

Usage: `(declare-volatile sym)`



Declares `sym`, which has to be quoted, as a volatile toplevel symbol. Volatile toplevel symbols are neither saved to nor loaded from zimages.

See also: `save-zimage`, `load-zimage`, `declare-unprotected`. [→index](#) [→topic](#)

#### 4.162 `def-custom-hook` : procedure/2

Usage: `(def-custom-hook sym proc)`

Define a custom hook point, to be called manually from Lisp. These have IDs starting from 65636.

See also: `add-hook`. [→index](#) [→topic](#)

#### 4.163 `default-error-handler` : procedure/0

Usage: `(default-error-handler)=> proc`

Return the default error handler, irrespectively of the current-error-handler.

See also: `current-error-handler`, `push-error-handler`, `pop-error-handler`, `*current-error-handler*`, `*current-error-continuation*`. [→index](#) [→topic](#)

#### 4.164 `defclass` : macro/2 or more

Usage: `(defclass name supers [props] ...)`

Defines symbol `name` as class with superclasses `supers` and property clauses `props` listed as remaining arguments. A `props` clause is either a symbol for a property or a list of the form `(sym default)` for the property `sym` with `default` value. The class is bound to `name` and a class predicate `name?` is created. Argument `supers` may be a class name or a list of class names.

See also: `defmethod`, `new`. [→index](#) [→topic](#)

#### 4.165 `defmacro` : macro/2 or more

Usage: `(defmacro name args body ...)`

Define a macro `name` with argument list `args` and `body`. Macros are expanded at compile-time.

See also: `macro`. [→index](#) [→topic](#)

#### 4.166 **defmethod** : macro/2 or more

Usage: (defmethod **class-name** **args** [**body**] ...)

Define a method **class-name** for class **class** and method name **name** with a syntax parallel to defun, where **args** are the arguments of the methods and **body** is the rest of the method. The given **class-name** must decompose into a valid class name **class** of a previously created class and method name **name** and is bound to the symbol **class-name**. The remaining arguments are like for defun. So for example (defmethod employee-name (this) (prop this 'last-name)) defines a method **name** for an existing class **employee** which retrieves the property **last-name**. Note that **defmethod** is dynamic: If you define a class B with class A as superclass, then B only inherits methods from A that have already been defined for A at the time of defining B!

See also: **defclass**, **new**, **call-method**. →index →topic

#### 4.167 **defstruct** : macro/1 or more

Usage: (defstruct **name** **props** ...) => **struct**

Binds symbol **name** to a struct with name **name** and with properties **props**. Each clause of **props** must be either a symbol for the property name or a list of the form (prop default-value) where **prop** is the symbol for the property name and **default-value** is the value it has by default. For each property **p**, accessors **name-p** and setters **name-p!** are created, as well as a function **name-p\*** that takes a record **r**, a value **v**, and a procedure **proc** that takes no arguments. When **name-p\*** is called on record **r**, it temporarily sets property **p** of **r** to the provided value **v** and calls the procedure **proc**. Afterwards, the original value of **p** is restored. Since this function mutates the record during the execution of **proc** and does not protect this operation against race conditions, it is not thread-safe. (But you can include a mutex as property and make it thread-safe by wrapping it into **with-mutex-lock**.) The defstruct macro returns the struct that is bound to **name**.

See also: **new-struct**, **make**, **with-mutex-lock**. →index →topic

#### 4.168 **defun** : macro/1 or more

Usage: (defun **ident** (**params** ...) **body** ...)

Define a function with name **ident**, a possibly empty list of **params**, and the remaining **body** expressions. This is a macro for (setq ident (lambda (params ...) body ...)) and binds the lambda-form to the given symbol. Like lambdas, the **params** of **defun** allow for a **&rest** keyword before the last parameter name. This binds all remaining arguments of a variadic function call to this parameter as a list.

See also: [setq](#), [defmacro](#). [→index](#) [→topic](#)

#### 4.169 **delete** : procedure/2

Usage: ([delete](#) *d* *key*)

Remove the value for *key* in dict *d*. This also removes the key.

See also: [dict?](#), [get](#), [set](#). [→index](#) [→topic](#)

#### 4.170 **dequeue!** : macro/1 or more

Usage: ([dequeue!](#) *sym* [*def*])=> *any*

Get the next element from queue *sym*, which must be the unquoted name of a variable, and return it. If a default *def* is given, then this is returned if the queue is empty, otherwise nil is returned.

See also: [make-queue](#), [queue?](#), [enqueue!](#), [glance](#), [queue-empty?](#), [queue-len](#). [→index](#) [→topic](#)

#### 4.171 **dict** : procedure/0 or more

Usage: ([dict](#) [*li*])=> *dict*

Create a dictionary. The option *li* must be a list of the form '(key1 value1 key2 value2 ...). Dictionaries are unordered, hence also not sequences. Dictionaries are safe for concurrent access.

See also: [array](#), [list](#). [→index](#) [→topic](#)

#### 4.172 **dict->alist** : procedure/1

Usage: ([dict->alist](#) *d*)=> *li*

Convert a dictionary into an association list. Note that the resulting alist will be a set of proper pairs of the form '(a . b) if the values in the dictionary are not lists.

See also: [dict](#), [dict-map](#), [dict->list](#). [→index](#) [→topic](#)

#### 4.173 **dict->array** : procedure/1

Usage: ([dict->array](#) *d*)=> *array*

Return an array that contains all key, value pairs of `d`. A key comes directly before its value, but otherwise the order is unspecified.

See also: `dict->list`, `dict.` [→index](#) [→topic](#)

#### 4.174 `dict->keys` : procedure/1

Usage: `(dict->keys d) => li`

Return the keys of dictionary `d` in arbitrary order.

See also: `dict`, `dict->values`, `dict->alist`, `dict->list`. [→index](#) [→topic](#)

#### 4.175 `dict->list` : procedure/1

Usage: `(dict->list d) => li`

Return a list of the form '(key1 value1 key2 value2 ...), where the order of key, value pairs is unspecified.

See also: `dict->array`, `dict.` [→index](#) [→topic](#)

#### 4.176 `dict->values` : procedure/1

Usage: `(dict->values d) => li`

Return the values of dictionary `d` in arbitrary order.

See also: `dict`, `dict->keys`, `dict->alist`, `dict->list`. [→index](#) [→topic](#)

#### 4.177 `dict-copy` : procedure/1

Usage: `(dict-copy d) => dict`

Return a copy of dict `d`.

See also: `dict`, `dict?`. [→index](#) [→topic](#)

#### 4.178 `dict-empty?` : procedure/1

Usage: `(dict-empty? d) => bool`

Return true if dict `d` is empty, nil otherwise. As crazy as this may sound, this can have  $O(n)$  complexity if the dict is not empty, but it is still going to be more efficient than any other method.

See also: `dict.` [→index](#) [→topic](#)

#### 4.179 `dict-foreach` : procedure/2

Usage: `(dict-foreach d proc)`

Call `proc` for side-effects with the key and value for each key, value pair in dict `d`.

See also: `dict-map!`, `dict?`, `dict.` [→index](#) [→topic](#)

#### 4.180 `dict-map` : procedure/2

Usage: `(dict-map dict proc)=> dict`

Returns a copy of `dict` with `proc` applies to each key value pair as arguments. Keys are immutable, so `proc` must take two arguments and return the new value.

See also: `dict-map!`, `map.` [→index](#) [→topic](#)

#### 4.181 `dict-map!` : procedure/2

Usage: `(dict-map! d proc)`

Apply procedure `proc` which takes the key and value as arguments to each key, value pair in dict `d` and set the respective value in `d` to the result of `proc`. Keys are not changed.

See also: `dict`, `dict?`, `dict-foreach.` [→index](#) [→topic](#)

#### 4.182 `dict-merge` : procedure/2

Usage: `(dict-merge a b)=> dict`

Create a new dict that contains all key-value pairs from dicts `a` and `b`. Note that this function is not symmetric. If a key is in both `a` and `b`, then the key value pair in `a` is retained for this key.

See also: `dict`, `dict-map`, `dict-map!`, `dict-foreach.` [→index](#) [→topic](#)

### 4.183 `dict-protect`: procedure/1

Usage: (`dict-protect` `d`)

Protect dict `d` against changes. Attempting to set values in a protected dict will cause an error, but all values can be read and the dict can be copied. This function requires permission 'allow-protect.

See also: `dict-unprotect`, `dict-protected?`, `protect`, `unprotect`, `protected?`, `permissions`, `permission?`. →index

**Warning: Protected dicts are full readable and can be copied, so you may need to use `protect` to also prevent changes to the toplevel symbol storing the dict!** →topic

### 4.184 `dict-protected?`: procedure/1

Usage: (`dict-protected?` `d`)

Return true if the dict `d` is protected against mutation, nil otherwise.

See also: `dict-protect`, `dict-unprotect`, `protect`, `unprotect`, `protected?`, `permissions`, `permission?`. →index →topic

### 4.185 `dict-unprotect`: procedure/1

Usage: (`dict-unprotect` `d`)

Unprotect the dict `d` so it can be mutated again. This function requires permission 'allow-unprotect.

See also: `dict-protect`, `dict-protected?`, `protect`, `unprotect`, `protected?`, `permissions`, `permission?`. →index →topic

### 4.186 `dict?`: procedure/1

Usage: (`dict?` `obj`)=> `bool`

Return true if `obj` is a dict, nil otherwise.

See also: `dict`. →index →topic

### 4.187 `dir`: procedure/1

Usage: (`dir` [`path`])=> `li`

Obtain a directory list for `path`. If `path` is not specified, the current working directory is listed.

See also: `dir?`, `open`, `close`, `read`, `write`. →index →topic

#### 4.188 `dir?`: procedure/1

Usage: `(dir? path) => bool`

Check if the file at `path` is a directory and return true, nil if the file does not exist or is not a directory.

See also: `file-exists?`, `dir`, `open`, `close`, `read`, `write`. →index →topic

#### 4.189 `disable-object`: procedure/1

Usage: `(disable-object obj)`

Disable the canvas object `obj`.

See also: `enable-object`, `hide-object`, `show-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index →topic

#### 4.190 `div`: procedure/2

Usage: `(div n k) => int`

Integer division of `n` by `k`.

See also: `truncate`, `/`, `int`. →index →topic

#### 4.191 `dolist`: macro/1 or more

Usage: `(dolist (name list [result]) body ...)` => `li`

Traverse the list `list` in order, binding `name` to each element subsequently and evaluate the `body` expressions with this binding. The optional `result` is the result of the traversal, nil if it is not provided.

See also: `letrec`, `foreach`, `map`. →index →topic

#### 4.192 **dotimes : macro/1 or more**

Usage: (`dotimes` (`name` `count` [`result`]))`body` ...) => `any`

Iterate `count` times, binding `name` to the counter starting from 0 until the counter has reached `count-1`, and evaluate the `body` expressions each time with this binding. The optional `result` is the result of the iteration, nil if it is not provided.

See also: `letrec`, `dolist`, `while`. →index →topic

#### 4.193 **dump : procedure/0 or more**

Usage: (`dump` [`sym`] [`all?`]) => `li`

Return a list of symbols starting with the characters of `sym` or starting with any characters if `sym` is omitted, sorted alphabetically. When `all?` is true, then all symbols are listed, otherwise only symbols that do not contain "\_" are listed. By convention, the underscore is used for auxiliary functions.

See also: `dump-bindings`, `save-zimage`, `load-zimage`. →index →topic

#### 4.194 **dump-bindings : procedure/0**

Usage: (`dump-bindings`) => `li`

Return a list of all top-level symbols with bound values, including those intended for internal use.

See also: `dump`. →index →topic

#### 4.195 **enable-object : procedure/1**

Usage: (`enable-object` `obj`)

Enable the canvas object `obj`.

See also: `disable-object`, `hide-object`, `show-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index →topic

#### 4.196 **enq : procedure/1**

Usage: (`enq` `proc`)



Put `proc` on a special internal queue for sequential execution and execute it when able. `proc` must be a procedure that takes no arguments. The queue can be used to synchronizing i/o commands but special care must be taken that `proc` terminates, or else the system might be damaged.

See also: `task`, `future`, `synout`, `synouty`. →index

**Warning: Calls to `enq` can never be nested, neither explicitly or implicitly by calling `enq` anywhere else in the call chain!** →topic

#### 4.197 `enqueue!` : macro/2

Usage: (`enqueue!` `sym elem`)

Put `elem` in queue `sym`, where `sym` is the unquoted name of a variable.

See also: `make-queue`, `queue?`, `dequeue!`, `glance`, `queue-empty?`, `queue-len`. →index →topic

#### 4.198 `entry-accepts-tab?` : procedure/1

Usage: (`entry-accepts-tab?` `entry`)=> `bool`

Return `nil` when the entry does not accept tabs, `t` otherwise.

See also: `new-entry`. →index →topic

#### 4.199 `epoch-ns->datelist` : procedure/1

Usage: (`epoch-ns->datelist ns`)=> `li`

Return the date list in UTC time corresponding to the Unix epoch nanoseconds `ns`.

See also: `date->epoch-ns`, `datestr->datelist`, `datestr`, `datestr*`, `day-of-week`, `week-of-date`, `now`. →index →topic

#### 4.200 `eq?` : procedure/2

Usage: (`eq?` `x y`)=> `bool`

Return true if `x` and `y` are equal, nil otherwise. In contrast to other LISPs, `eq?` checks for deep equality of arrays and dicts. However, lists are compared by checking whether they are the same cell in memory. Use `equal?` to check for deep equality of lists and other objects.

See also: `equal?`. →index →topic

#### 4.201 eql? : procedure/2

Usage: (`eql?` *x* *y*)=> *bool*

Returns true if *x* is equal to *y*, nil otherwise. This is currently the same as `equal?` but the behavior might change.

See also: `equal?`. →index

**Warning: Deprecated.** →topic

#### 4.202 equal? : procedure/2

Usage: (`equal?` *x* *y*)=> *bool*

Return true if *x* and *y* are equal, nil otherwise. The equality is tested recursively for containers like lists and arrays.

See also: `eq?`, `eql?`. →index →topic

#### 4.203 error : procedure/0 or more

Usage: (`error` [*msgstr*] [*expr*] ...)

Raise an error, where *msgstr* and the optional expressions *expr*... work as in a call to `fmt`.

See also: `fmt`, `with-final`. →index →topic

#### 4.204 error->str : procedure/1

Usage: (`error->str` *datum*)=> *str*

Convert a special error value to a string.

See also: `*last-error*`, `error`, `error?`. →index →topic

#### 4.205 error? : procedure/1

Usage: (`error?` *datum*)=> *bool*

Return true if *datum* is a special error value, nil otherwise.

See also: `*last-error*`, `error->str`, `error`, `eof?`, `valid?`. →index →topic

#### 4.206 `eval` : procedure/1

Usage: (`eval` `expr`)=> `any`

Evaluate the expression `expr` in the Z3S5 Machine Lisp interpreter and return the result. The evaluation environment is the system's environment at the time of the call.

See also: `break`, `apply`. [→index](#) [→topic](#)

#### 4.207 `even?` : procedure/1

Usage: (`even?` `n`)=> `bool`

Returns true if the integer `n` is even, nil if it is not even.

See also: `odd?`. [→index](#) [→topic](#)

#### 4.208 `exists?` : procedure/2

Usage: (`exists?` `seq` `pred`)=> `bool`

Return true if `pred` returns true for at least one element in sequence `seq`, nil otherwise.

See also: `forall?`, `list-exists?`, `array-exists?`, `str-exists?`, `seq?`. [→index](#) [→topic](#)

#### 4.209 `exit` : procedure/0 or more

Usage: (`exit` [`n`])

Immediately shut down the system and return OS host error code `n`. The shutdown is performed gracefully and exit hooks are executed.

See also: `.` [→index](#) [→topic](#)

#### 4.210 `expand-macros` : procedure/1

Usage: (`expand-macros` `expr`)=> `expr`

Expands the macros in `expr`. This is an ordinary function and will not work on already compiled expressions such as a function bound to a symbol. However, it can be used to expand macros in expressions obtained by `read`.

See also: `internalize`, `externalize`, `load-library`. [→index](#) [→topic](#)

#### 4.211 **expect** : macro/2

Usage: (`expect` `value` `given`)

Registers a test under the current test name that checks that `value` is returned by `given`. The test is only executed when (`run-selftest`) is executed.

See also: `expect-err`, `expect-ok`, `run-selftest`, `testing`. →index →topic

#### 4.212 **expect-err** : macro/1 or more

Usage: (`expect-err` `expr` ...)

Registers a test under the current test name that checks that `expr` produces an error.

See also: `expect`, `expect-ok`, `run-selftest`, `testing`. →index →topic

#### 4.213 **expect-false** : macro/1 or more

Usage: (`expect-false` `expr` ...)

Registers a test under the current test name that checks that `expr` is nil.

See also: `expect`, `expect-ok`, `run-selftest`, `testing`. →index →topic

#### 4.214 **expect-ok** : macro/1 or more

Usage: (`expect-err` `expr` ...)

Registers a test under the current test name that checks that `expr` does not produce an error.

See also: `expect`, `expect-ok`, `run-selftest`, `testing`. →index →topic

#### 4.215 **expect-true** : macro/1 or more

Usage: (`expect-true` `expr` ...)

Registers a test under the current test name that checks that `expr` is true (not nil).

See also: `expect`, `expect-ok`, `run-selftest`, `testing`. →index →topic

#### 4.216 `expr->str` : procedure/1

Usage: `(expr->str expr)=> str`

Convert a Lisp expression `expr` into a string. Does not use a stream port.

See also: `str->expr`, `str->expr*`, `openstr`, `internalize`, `externalize`. [→index](#) [→topic](#)

#### 4.217 `externalize` : procedure/1

Usage: `(externalize sym [nonce])=> sexpr`

Obtain an external representation of top-level symbol `sym`. The optional `nonce` must be a value unique in each system zimage, in order to distinguish data from procedures.

See also: `can-externalize?`, `externalize0`, `current-zimage`, `save-zimage`, `load-zimage`. [→index](#) [→topic](#)

#### 4.218 `externalize0` : procedure/1

Usage: `(externalize0 arg)=> any`

Attempts to externalize `arg` but falls back to the internal expression if `arg` cannot be externalized. This procedure never fails but `can-externalize?` may be false for the result. This function is only used in miscellaneous printing. Use `externalize` to externalize expressions for writing to disk.

See also: `externalize`, `can-externalize?`. [→index](#) [→topic](#)

#### 4.219 `fdelete` : procedure/1

Usage: `(fdelete path)`

Removes the file or directory at `path`.

See also: `file-exists?`, `dir?`, `dir`. [→index](#)

**Warning: This function also deletes directories containing files and all of their subdirectories!** [→topic](#)

#### 4.220 `feature?` : procedure/1

Usage: `(feature? sym)=> bool`

Return true if the Lisp feature identified by symbol `sym` is available, nil otherwise.

See also: `*reflect*`, `on-feature`. →index →topic

#### 4.221 `file-port? : procedure/1`

Usage: `(file-port? p)=> bool`

Return true if `p` is a file port, nil otherwise.

See also: `port?`, `str-port?`, `open`, `stropen`. →index →topic

#### 4.222 `filter : procedure/2`

Usage: `(filter li pred)=> li`

Return the list based on `li` with each element removed for which `pred` returns nil.

See also: `list`. →index →topic

#### 4.223 `find-missing-help-entries : procedure/0`

Usage: `(find-missing-help-entries)=> li`

Return a list of global symbols for which help entries are missing.

See also: `dump`, `dump-bindings`, `find-unneeded-help-entries`. →index →topic

#### 4.224 `find-unneeded-help-entries : procedure/0`

Usage: `(find-unneeded-help-entries)=> li`

Return a list of help entries for which no symbols are defined.

See also: `dump`, `dump-bindings`, `find-missing-help-entries`. →index

**Warning: This function returns false positives! Special forms like `setq` and `macro` are listed even though they clearly are useful and should have a help entry.** →topic

**4.225 fl.abs : procedure/1**

Usage: (fl.abs *x*)=> fl

Return the absolute value of *x*.

See also: [float](#), [\\*](#). [→index](#) [→topic](#)

**4.226 fl.acos : procedure/1**

Usage: (fl.acos *x*)=> fl

Return the arc cosine of *x*.

See also: [fl.cos](#). [→index](#) [→topic](#)

**4.227 fl.asin : procedure/1**

Usage: (fl.asin *x*)=> fl

Return the arc sine of *x*.

See also: [fl.acos](#). [→index](#) [→topic](#)

**4.228 fl.asinh : procedure/1**

Usage: (fl.asinh *x*)=> fl

Return the inverse hyperbolic sine of *x*.

See also: [fl.cosh](#). [→index](#) [→topic](#)

**4.229 fl.atan : procedure/1**

Usage: (fl.atan *x*)=> fl

Return the arctangent of *x* in radians.

See also: [fl.atanh](#), [fl.tan](#). [→index](#) [→topic](#)

**4.230 fl.atan2 : procedure/2**

Usage: (fl.atan2 x y)=> fl

Atan2 returns the arc tangent of  $y/x$ , using the signs of the two to determine the quadrant of the return value.

See also: fl.atan. [→index](#) [→topic](#)

**4.231 fl.atanh : procedure/1**

Usage: (fl.atanh x)=> fl

Return the inverse hyperbolic tangent of  $x$ .

See also: fl.atan. [→index](#) [→topic](#)

**4.232 fl.cbrt : procedure/1**

Usage: (fl.cbrt x)=> fl

Return the cube root of  $x$ .

See also: fl.sqrt. [→index](#) [→topic](#)

**4.233 fl.ceil : procedure/1**

Usage: (fl.ceil x)=> fl

Round  $x$  up to the nearest integer, return it as a floating point number.

See also: fl.floor, truncate, int, fl.round, fl.trunc. [→index](#) [→topic](#)

**4.234 fl.cos : procedure/1**

Usage: (fl.cos x)=> fl

Return the cosine of  $x$ .

See also: fl.sin. [→index](#) [→topic](#)



**4.235 fl.cosh : procedure/1**

Usage: (fl.cosh x)=> fl

Return the hyperbolic cosine of  $x$ .

See also: fl.cos. →index →topic

**4.236 fl.dim : procedure/2**

Usage: (fl.dim x y)=> fl

Return the maximum of  $x$ ,  $y$  or 0.

See also: max. →index →topic

**4.237 fl.erf : procedure/1**

Usage: (fl.erf x)=> fl

Return the result of the error function of  $x$ .

See also: fl.erfc, fl.dim. →index →topic

**4.238 fl.erfc : procedure/1**

Usage: (fl.erfc x)=> fl

Return the result of the complementary error function of  $x$ .

See also: fl.erfcinv, fl.erf. →index →topic

**4.239 fl.erfcinv : procedure/1**

Usage: (fl.erfcinv x)=> fl

Return the inverse of (fl.erfc  $x$ ).

See also: fl.erfc. →index →topic

**4.240 fl.erfinv : procedure/1**

Usage: (fl.erfinv x) => fl

Return the inverse of (fl.erf x).

See also: fl.erf. →index →topic

**4.241 fl.exp : procedure/1**

Usage: (fl.exp x) => fl

Return  $e^x$ , the base-e exponential of x.

See also: fl.exp. →index →topic

**4.242 fl.exp2 : procedure/2**

Usage: (fl.exp2 x) => fl

Return  $2^x$ , the base-2 exponential of x.

See also: fl.exp. →index →topic

**4.243 fl.expm1 : procedure/1**

Usage: (fl.expm1 x) => fl

Return  $e^x - 1$ , the base-e exponential of (sub1 x). This is more accurate than (sub1 (fl.exp x)) when x is very small.

See also: fl.exp. →index →topic

**4.244 fl.floor : procedure/1**

Usage: (fl.floor x) => fl

Return x rounded to the nearest integer below as floating point number.

See also: fl.ceil, truncate, int. →index →topic

**4.245 fl.fma : procedure/3**

Usage: (fl.fma x y z)=> fl

Return the fused multiply-add of  $x$ ,  $y$ ,  $z$ , which is  $x * y + z$ .

See also: \*, +. →index →topic

**4.246 fl.frexp : procedure/1**

Usage: (fl.frexp x)=> li

Break  $x$  into a normalized fraction and an integral power of two. It returns a list of (frac exp) containing a float and an integer satisfying  $x == \text{frac} \times 2^{\text{exp}}$  where the absolute value of  $\text{frac}$  is in the interval  $[0.5, 1)$ .

See also: fl.exp. →index →topic

**4.247 fl.gamma : procedure/1**

Usage: (fl.gamma x)=> fl

Compute the Gamma function of  $x$ .

See also: fl.lgamma. →index →topic

**4.248 fl.hypot : procedure/2**

Usage: (fl.hypot x y)=> fl

Compute the square root of  $x^2$  and  $y^2$ .

See also: fl.sqrt. →index →topic

**4.249 fl.ilogb : procedure/1**

Usage: (fl.ilogb x)=> fl

Return the binary exponent of  $x$  as a floating point number.

See also: fl.exp2. →index →topic

**4.250 fl.inf : procedure/1**

Usage: (fl.inf x) => fl

Return positive 64 bit floating point infinity +INF if  $x \geq 0$  and negative 64 bit floating point finfinity -INF if  $x < 0$ .

See also: fl.is-nan?. →index →topic

**4.251 fl.is-nan? : procedure/1**

Usage: (fl.is-nan? x) => bool

Return true if  $x$  is not a number according to IEEE 754 floating point arithmetics, nil otherwise.

See also: fl.inf. →index →topic

**4.252 fl.j0 : procedure/1**

Usage: (fl.j0 x) => fl

Apply the order-zero Bessel function of the first kind to  $x$ .

See also: fl.j1, fl.jn, fl.y0, fl.y1, fl.yn. →index →topic

**4.253 fl.j1 : procedure/1**

Usage: (fl.j1 x) => fl

Apply the the order-one Bessel function of the first kind  $x$ .

See also: fl.j0, fl.jn, fl.y0, fl.y1, fl.yn. →index →topic

**4.254 fl.jn : procedure/1**

Usage: (fl.jn n x) => fl

Apply the Bessel function of order  $n$  to  $x$ . The number  $n$  must be an integer.

See also: fl.j1, fl.j0, fl.y0, fl.y1, fl.yn. →index →topic

**4.255 fl.lldexp : procedure/2**

Usage: (fl.lldexp x n) => fl

Return the inverse of fl.frexp,  $x * 2^n$ .

See also: fl.frexp. →index →topic

**4.256 fl.lgamma : procedure/1**

Usage: (fl.lgamma x) => li

Return a list containing the natural logarithm and sign (-1 or +1) of the Gamma function applied to  $x$ .

See also: fl.gamma. →index →topic

**4.257 fl.log : procedure/1**

Usage: (fl.log x) => fl

Return the natural logarithm of  $x$ .

See also: fl.log10, fl.log2, fl.logb, fl.log1p. →index →topic

**4.258 fl.log10 : procedure/1**

Usage: (fl.log10 x) => fl

Return the decimal logarithm of  $x$ .

See also: fl.log, fl.log2, fl.logb, fl.log1p. →index →topic

**4.259 fl.log1p : procedure/1**

Usage: (fl.log1p x) => fl

Return the natural logarithm of  $x + 1$ . This function is more accurate than (fl.log (add1 x)) if  $x$  is close to 0.

See also: fl.log, fl.log2, fl.logb, fl.log10. →index →topic

**4.260 fl.log2 : procedure/1**

Usage: (fl.log2 x)=> fl

Return the binary logarithm of *x*. This is important for calculating entropy, for example.

See also: fl.log, fl.log10, fl.log1p, fl.logb. →index →topic

**4.261 fl.logb : procedure/1**

Usage: (fl.logb x)=> fl

Return the binary exponent of *x*.

See also: fl.log, fl.log10, fl.log1p, fl.logb, fl.log2. →index →topic

**4.262 fl.max : procedure/2**

Usage: (fl.max x y)=> fl

Return the larger value of two floating point arguments *x* and *y*.

See also: fl.min, max, min. →index →topic

**4.263 fl.min : procedure/2**

Usage: (fl.min x y)=> fl

Return the smaller value of two floating point arguments *x* and *y*.

See also: fl.min, max, min. →index →topic

**4.264 fl.mod : procedure/2**

Usage: (fl.mod x y)=> fl

Return the floating point remainder of *x* / *y*.

See also: fl.reminder. →index →topic

**4.265 fl.modf : procedure/1**

Usage: (fl.modf *x*) => *li*

Return integer and fractional floating-point numbers that sum to *x*. Both values have the same sign as *x*.

See also: fl.mod. →index →topic

**4.266 fl.nan : procedure/1**

Usage: (fl.nan) => fl

Return the IEEE 754 not-a-number value.

See also: fl.is-nan?, fl.inf. →index →topic

**4.267 fl.next-after : procedure/1**

Usage: (fl.next-after *x*) => fl

Return the next representable floating point number after *x*.

See also: fl.is-nan?, fl.nan, fl.inf. →index →topic

**4.268 fl.pow : procedure/2**

Usage: (fl.pow *x y*) => fl

Return *x* to the power of *y* according to 64 bit floating point arithmetics.

See also: fl.pow10. →index →topic

**4.269 fl.pow10 : procedure/1**

Usage: (fl.pow10 *n*) => fl

Return 10 to the power of integer *n* as a 64 bit floating point number.

See also: fl.pow. →index →topic

**4.270 fl.reminder : procedure/2**

Usage: (fl.reminder x y)=> fl

Return the IEEE 754 floating-point remainder of  $x/y$ .

See also: fl.mod. →index →topic

**4.271 fl.round : procedure/1**

Usage: (fl.round x)=> fl

Round  $x$  to the nearest integer floating point number according to floating point arithmetics.

See also: fl.round-to-even, fl.truncate, int, float. →index →topic

**4.272 fl.round-to-even : procedure/1**

Usage: (fl.round-to-even x)=> fl

Round  $x$  to the nearest even integer floating point number according to floating point arithmetics.

See also: fl.round, fl.truncate, int, float. →index →topic

**4.273 fl.signbit : procedure/1**

Usage: (fl.signbit x)=> bool

Return true if  $x$  is negative, nil otherwise.

See also: fl.abs. →index →topic

**4.274 fl.sin : procedure/1**

Usage: (fl.sin x)=> fl

Return the sine of  $x$ .

See also: fl.cos. →index →topic



**4.275 fl.sinh : procedure/1**

Usage: (fl.sinh x)=> fl

Return the hyperbolic sine of *x*.

See also: fl.sin. →index →topic

**4.276 fl.sqrt : procedure/1**

Usage: (fl.sqrt x)=> fl

Return the square root of *x*.

See also: fl.pow. →index →topic

**4.277 fl.tan : procedure/1**

Usage: (fl.tan x)=> fl

Return the tangent of *x* in radian.

See also: fl.tanh, fl.sin, fl.cos. →index →topic

**4.278 fl.tanh : procedure/1**

Usage: (fl.tanh x)=> fl

Return the hyperbolic tangent of *x*.

See also: fl.tan, flsinh, fl.cosh. →index →topic

**4.279 fl.trunc : procedure/1**

Usage: (fl.trunc x)=> fl

Return the integer value of *x* as floating point number.

See also: truncate, int, fl.floor. →index →topic

**4.280 fl.y0 : procedure/1**

Usage: (fl.y0 x)=> fl

Return the order-zero Bessel function of the second kind applied to x.

See also: fl.y1, fl.yn, fl.j0, fl.j1, fl.jn. →index →topic

**4.281 fl.y1 : procedure/1**

Usage: (fl.y1 x)=> fl

Return the order-one Bessel function of the second kind applied to x.

See also: fl.y0, fl.yn, fl.j0, fl.j1, fl.jn. →index →topic

**4.282 fl.yn : procedure/1**

Usage: (fl.yn n x)=> fl

Return the Bessel function of the second kind of order n applied to x. Argument n must be an integer value.

See also: fl.y0, fl.y1, fl.j0, fl.j1, fl.jn. →index →topic

**4.283 flatten : procedure/1**

Usage: (flatten lst)=> list

Flatten lst, making all elements of sublists elements of the flattened list.

See also: car, cdr, remove-duplicates. →index →topic

**4.284 float : procedure/1**

Usage: (float n)=> float

Convert n to a floating point value.

See also: int. →index →topic

#### 4.285 **fmt : procedure/1 or more**

Usage: (`fmt` `s` [`args`] ...) => `str`

Format string `s` that contains format directives with arbitrary many `args` as arguments. The number of format directives must match the number of arguments. The format directives are the same as those for the esoteric and arcane programming language “Go”, which was used on Earth for some time.

See also: `out`. →index →topic

#### 4.286 **forall? : procedure/2**

Usage: (`forall?` `seq` `pred`) => `bool`

Return true if predicate `pred` returns true for all elements of sequence `seq`, nil otherwise.

See also: `foreach`, `map`, `list-forall?`, `array-forall?`, `str-forall?`, `exists?`, `str-exists?`, `array-exists?`, `list-exists?`. →index →topic

#### 4.287 **force : procedure/1**

Usage: (`force` `fut`) => `any`

Obtain the value of the computation encapsulated by future `fut`, halting the current task until it has been obtained. If the future never ends computation, e.g. in an infinite loop, the program may halt indefinitely.

See also: `future`, `task`, `make-mutex`. →index →topic

#### 4.288 **foreach : procedure/2**

Usage: (`foreach` `seq` `proc`)

Apply `proc` to each element of sequence `seq` in order, for the side effects.

See also: `seq?`, `map`. →index →topic

#### 4.289 **forget : procedure/1**

Usage: (`forget` `key`)

Forget the value associated with `key`. This permanently deletes the value from the persistent record.

See also: `remember`, `recall`, `recollect`, `recall-when`, `recall-info`. →index →topic

#### 4.290 `forget-gui-object` : procedure/1

Usage: (`forget-gui-object` `int`)

Forget the GUI object `int`. This removes any association with the object but does not free internal resources if the object still exists. Internal use only.

See also: `close-window`, `close-gui`. →index →topic

#### 4.291 `functional-arity` : procedure/1

Usage: (`functional-arity` `proc`)=> `int`

Return the arity of a functional `proc`.

See also: `functional?`, `functional-has-rest?`. →index →topic

#### 4.292 `functional-has-rest?` : procedure/1

Usage: (`functional-has-rest?` `proc`)=> `bool`

Return true if the functional `proc` has a `&rest` argument, nil otherwise.

See also: `functional?`, `functional-arity`. →index →topic

#### 4.293 `functional?` : macro/1

Usage: (`functional?` `arg`)=> `bool`

Return true if `arg` is either a builtin function, a closure, or a macro, nil otherwise. This is the right predicate for testing whether the argument is applicable and has an arity.

See also: `closure?`, `proc?`, `functional-arity`, `functional-has-rest?`. →index →topic

#### 4.294 `gensym` : procedure/0

Usage: (`gensym`)=> `sym`

Return a new symbol guaranteed to be unique during runtime.

See also: `nonce`. →index →topic

#### 4.295 **get** : procedure/2 or more

Usage: (`get dict key [default]`)=> `any`

Get the value for `key` in `dict`, return `default` if there is no value for `key`. If `default` is omitted, then `nil` is returned. Provide your own default if you want to store `nil`.

See also: `dict`, `dict?`, `set`. →index →topic

#### 4.296 **get-action** : procedure/1

Usage: (`get-action id`)=> `action`

Return a cloned action based on `id` from the action registry. This action can be run using `action-start` and will get its own taskid.

See also: `action`, `has-action-system?`, `action-start`, `action-stop`, `register-action`. →index →topic

#### 4.297 **get-clipboard-content** : procedure/0

Usage: (`get-clipboard-content`)=> `str`

Return the current content of the operating system clipboard as string. This function might raise an error if clipboard access is prohibited by host security settings.

See also: `set-clipboard-content`. →index →topic

#### 4.298 **get-device-info** : procedure/0

Usage: (`get-device-info`)=> `li`

Return a list with information about the current host device. This returns an association list where 'orientation might be one of '(vertical vertical-upside-down left right unknown), self-explanatory boolean keys 'is-mobile?, 'is-browser, 'has-keyboard?, and 'system-scale with the current scaling factor for graphics as float. The system scale is used to dynamically scale user interface elements to remain legible on hi res displays.

See also: `close-gui`. →index →topic

#### 4.299 `get-entry-cursor` : procedure/1

Usage: `(get-entry-cursor entry)=> sym`

Return a symbol that represents the current cursor of `entry`. Possible values are in `'`(default text crosshair pointer hresize vresize). Curiously, there is no way to set the cursor yet.

See also: `new-entry`. [→index](#) [→topic](#)

#### 4.300 `get-entry-cursor-pos` : procedure/1

Usage: `(get-entry-cursor-pos entry)=> li`

Return a list consisting of row number and column number of the current cursor position of the cursor in `entry`.

See also: `set-entry-cursor-row`, `set-entry-cursor-column`. [→index](#) [→topic](#)

#### 4.301 `get-menu-item-label` : procedure/1

Usage: `(get-menu-item-label item)=> str`

Return the current label of the given menu `item`.

See also: `set-menu-item-label`, `set-menu-item-disabled`, `menu-item-disabled`, `set-menu-item-checked`, `menu-item-checked?`, `new-menu*`, `new-menu`, `new-menu-item`, `new-menu-item-separator`. [→index](#) [→topic](#)

#### 4.302 `get-object-min-size` : procedure/1

Usage: `(get-object-min-size obj)=> li`

Return the minimum size of canvas object `obj` as a list containing the width and height as floats. The minimum size is computed based on various internal criteria and can only be changed for some special widgets.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. [→index](#) [→topic](#)

### 4.303 `get-object-position`: procedure/1

Usage: `(get-object-position obj)=> li`

Return the position of canvas object `obj` as a list containing the x and y coordinates as floats.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. [→index](#) [→topic](#)

### 4.304 `get-object-size`: procedure/1

Usage: `(get-object-size obj)=> li`

Return the size of canvas object `obj` as a list containing the width and height as floats.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. [→index](#) [→topic](#)

### 4.305 `get-or-set`: procedure/3

Usage: `(get-or-set d key value)`

Get the value for `key` in dict `d` if it already exists, otherwise set it to `value`.

See also: `dict?`, `get`, `set`. [→index](#) [→topic](#)

### 4.306 `get-partitions`: procedure/2

Usage: `(get-partitions x n)=> proc/1*`

Return an iterator procedure that returns lists of the form (start-offset end-offset bytes) with 0-index offsets for a given index `k`, or nil if there is no corresponding part, such that the sizes of the partitions returned in `bytes` summed up are `x` and each partition is `n` or lower in size. The last partition will be the smallest partition with a `bytes` value smaller than `n` if `x` is not dividable without rest by `n`. If no argument is provided for the returned iterator, then it returns the number of partitions.

See also: `nth-partition`, `count-partitions`, `get-file-partitions`, `iterate`. [→index](#) [→topic](#)

### 4.307 `get-progress-bar-value`: procedure/1

Usage: `(get-progress-bar-value bar)=> num`

Return the current value of progress-bar `bar`.

See also: `set-progress-bar`, `new-progress-bar`, `new-infinite-progress-bar`. [→index](#)  
[→topic](#)

### 4.308 `get-text-grid-cell`: procedure/3

Usage: `(get-text-grid-cell grid row column)=> li`

Return the cell of `grid` at `row` and `column`. The result is a list consisting of a string containing one unicode rune and a grid style list.

See also: `set-text-grid-cell`, `get-text-grid-row`, `set-text-grid-rune`, `set-text-grid-style-range`. [→index](#) [→topic](#)

### 4.309 `get-text-grid-row`: procedure/2

Usage: `(get-text-grid-row grid row)=> li`

Obtain a `row` of a text `grid`, where `row` is a 0-based index. This function returns a list of the form `'(row style)`, where `style` is a grid style list and `row` is an array of lists consisting each of a unicode string containing one rune and a grid style list. Each entry of the `row` array represents an individual unicode glyph with a style, whereas the `style` list in the return argument represents an optional style of the whole row.

See also: `set-text-grid-row`, `get-text-grid-row-text`, `get-text-grid-cell`, `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `set-text-grid-cell`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. [→index](#) [→topic](#)

### 4.310 `get-text-grid-row-text`: procedure/2

Usage: `(get-text-grid-row-text grid row)=> str`

Return the text of `row` in `grid` as a string without any style information.



See also: `set-text-grid-rune`, `get-text-grid-row`, `get-text-grid-cell`, `set-text-grid-row`, `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `set-text-grid-cell`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. [→index](#) [→topic](#)

#### 4.311 `get-text-grid-tab-width`: procedure/1

Usage: (`get-text-grid-tab-width` *grid*)=> **int**

Return the current tabulator width of *grid* in space characters.

See also: `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. [→index](#) [→topic](#)

#### 4.312 `get-text-grid-text`: procedure/1

Usage: (`get-text-grid-text` *grid*)=> **str**

Return the text of *grid* as a string without style information.

See also: `set-text-grid-text`, `new-text-grid`, `get-text-grid-row`, `get-text-grid-rune`, `get-text-grid-cell`. [→index](#) [→topic](#)

#### 4.313 `get-window-canvas`: procedure/1

Usage: (`get-window-canvas` *window*)=> **int**

Get the canvas object of *window*, which is the area on which window elements are drawn. This is not the same as the `window-content`, which is a widget or other user interface element. The canvas is used for raw-drawing commands, for example for drawing circles and boxes. With a suitable layout that doesn't re-arrange objects, it can e.g. be used to draw overlays.

See also: `get-window-content`, `set-window-content`. [→index](#) [→topic](#)

#### 4.314 `get-window-content` : procedure/2

Usage: (`get-window-content` *window*)=> **int**

Get the canvas object ID that represents the main content of the window. This is usually a widget or a container with layout.

See also: `set-window-content`, `get-window-canvas`. →index →topic

#### 4.315 `get-window-icon` : procedure/1

Usage: (`get-window-icon` *window*)=> **int**

Obtain the icon ID of the icon of *window*. The resource obtained is not guaranteed to be a visible icon or might be a dummy, as not all windows have icons on all platforms.

See also: `set-window-icon`, `new-icon`, `theme-icon`. →index →topic

#### 4.316 `get-window-main-menu` : procedure/2

Usage: (`get-window-main-menu` *window*)=> **int**

Get the main menu ID of *window*.

See also: `set-window-main-menu`, `new-main-menu`. →index →topic

#### 4.317 `get-window-title` : procedure/1

Usage: (`get-window-title` *window*)=> **str**

Return the title of *window* as string.

See also: `set-window-title`. →index →topic

#### 4.318 `getstacked` : procedure/3

Usage: (`getstacked` *dict* *key* **default**)

Get the topmost element from the stack stored at *key* in *dict*. If the stack is empty or no stack is stored at *key*, then **default** is returned.

See also: `pushstacked`, `popstacked`. →index →topic

#### 4.319 `glance` : procedure/1

Usage: (`glance` *s* [*def*])=> *any*

Peek the next element in a stack or queue without changing the data structure. If default `def` is provided, this is returned in case the stack or queue is empty; otherwise nil is returned.

See also: `make-queue`, `make-stack`, `queue?`, `enqueue?`, `dequeue?`, `queue-len`, `stack-len`, `pop!`, `push!`. →index →topic

#### 4.320 `global-sym?` : procedure/1

Usage: (`global-sym?` *sym*)=> *bool*

Returns true if *sym* is a global symbol, nil otherwise. By convention, a symbol counts as global if it starts with a "\*" character. This is used by library functions to determine whether a top-level symbol ought to be treated as local or global to the library.

See also: `load`, `include`, `sym?`. →index →topic

#### 4.321 `has` : procedure/2

Usage: (`has` *dict* *key*)=> *bool*

Return true if the dict *dict* contains an entry for *key*, nil otherwise.

See also: `dict`, `get`, `set`. →index →topic

#### 4.322 `has-action-system?` : procedure/0

Usage: (`has-action-system?`)=> *bool*

This predicate is true if the action system is available, **false** otherwise.

See also: `action`, `init-actions`, `action-start`, `action-stop`, `registered-actions`, `register-action`. →index →topic

#### 4.323 `has-action?` : procedure/1

Usage: (`has-action?` *prefix* *name*)=> *bool*

Return true if an action with the given `prefix` and `name` is registered, nil otherwise. Actions are indexed by id, so this is much slower than using `get-action` to retrieve a registered action by the value of the 'id property.

See also: `get-action`, `action`, `has-action-system?`, `register-action`. →index →topic

#### 4.324 `has-key? : procedure/2`

Usage: `(has-key? d key) => bool`

Return true if `d` has key `key`, nil otherwise.

See also: `dict?`, `get`, `set`, `delete`. →index →topic

#### 4.325 `has-method? : procedure/2`

Usage: `(has-method? obj name) => bool`

Return true if `obj` has a method with name `name`, nil otherwise.

See also: `defmethod`, `has-prop?`, `new`, `props`, `methods`, `prop`, `setprop`. →index →topic

#### 4.326 `has-prop? : procedure/2`

Usage: `(has-prop? obj slot) => bool`

Return true if `obj` has a property named `slot`, nil otherwise.

See also: `has-method?`, `new`, `props`, `methods`, `prop`, `setprop`. →index →topic

#### 4.327 `help : macro/1`

Usage: `(help sym)`

Display help information about `sym` (unquoted).

See also: `defhelp`, `help-topics`, `help-about`, `help-topic-info`, `set-help-topic-info`, `help-entry`, `*help*`, `apropos`. →index →topic

#### 4.328 `help->manual-entry` : nil

Usage: (`help->manual-entry` *key* [*level*] [*link?*])=> *str*

Looks up help for *key* and converts it to a manual section as markdown string. If there is no entry for *key*, then nil is returned. The optional *level* integer indicates the heading nesting. If *link?* is true an anchor is created for the key.

See also: `help`. `→index` `→topic`

#### 4.329 `help-about` : procedure/1 or more

Usage: (`help-about` *topic* [*sel*])=> *li*

Obtain a list of symbols for which help about *topic* is available. If optional *sel* argument is left out or *any*, then any symbols with which the topic is associated are listed. If the optional *sel* argument is *first*, then a symbol is only listed if it has *topic* as first topic entry. This restricts the number of entries returned to a more essential selection.

See also: `help-topics`, `help`, `apropos`. `→index` `→topic`

#### 4.330 `help-entry` : procedure/1

Usage: (`help-entry` *sym*)=> *list*

Get usage and help information for *sym*.

See also: `defhelp`, `help`, `apropos`, `*help*`, `help-topics`, `help-about`, `set-help-topic-info`, `help-topic-info`. `→index` `→topic`

#### 4.331 `help-strings` : procedure/2

Usage: (`help-strings` *sym* *del*)=> *li*

Obtain a string of help strings for a given symbol *sym*. The fields in the string are separated by string *del*.

See also: `help`, `help-entry`, `*help*`. `→index` `→topic`

### 4.332 `help-topic-info`: procedure/1

Usage: (`help-topic-info` *topic*)=> *li*

Return a list containing a heading and an info string for help *topic*, or nil if no info is available.

See also: `set-help-topic-info`, `defhelp`, `help`. →index →topic

### 4.333 `help-topics`: procedure/0

Usage: (`help-topics`)=> *li*

Obtain a list of help topics for commands.

See also: `help`, `help-topic`, `apropos`. →index →topic

### 4.334 `hex->blob`: procedure/1

Usage: (`hex->blob` *str*)=> *blob*

Convert hex string *str* to a blob. This will raise an error if *str* is not a valid hex string.

See also: `blob->hex`, `base64->blob`, `ascii85->blob`, `str->blob`. →index →topic

### 4.335 `hide-object`: procedure/1

Usage: (`hide-object` *obj*)

Hide the canvas object *obj*.

See also: `disable-object`, `enable-object`, `show-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index →topic

### 4.336 `hide-window`: procedure/1

Usage: (`hide-window` *window*)

Hides *window*. It can be shown again using `show-window`.

See also: `show-window`, `close-window`. →index →topic

### 4.337 hook : procedure/1

Usage: (`hook` `symbol`)

Lookup the internal hook number from a symbolic name.

See also: `*hooks*`, `add-hook`, `remove-hook`, `remove-hooks`. [→index](#) [→topic](#)

### 4.338 hour+ : procedure/2

Usage: (`hour+` `date` `i` `n`)=> `date` `i`

Adds `n` hours to the given date `date` `i` in datelist format and returns the new datelist.

See also: `sec+`, `minute+`, `day+`, `week+`, `month+`, `year+`, `now`. [→index](#) [→topic](#)

### 4.339 identity : procedure/1

Usage: (`identity` `x`)

Return `x`.

See also: `apply`, `equal?`. [→index](#) [→topic](#)

### 4.340 if : macro/3

Usage: (`if` `cond` `expr1` `expr2`)=> `any`

Evaluate `expr1` if `cond` is true, otherwise evaluate `expr2`.

See also: `cond`, `when`, `unless`. [→index](#) [→topic](#)

### 4.341 inchars : procedure/2

Usage: (`inchars` `char` `chars`)=> `bool`

Return true if `char` is in the charset `chars`, nil otherwise.

See also: `chars`, `dict`, `get`, `set`, `has`. [→index](#) [→topic](#)

#### 4.342 `include` : procedure/1

Usage: (`include` `fi`)=> `any`

Evaluate the lisp file `fi` one expression after the other in the current environment.

See also: `read`, `write`, `open`, `close`. →index →topic

#### 4.343 `index` : procedure/2 or more

Usage: (`index` `seq` `elem` [`pred`])=> `int`

Return the first index of `elem` in `seq` going from left to right, using equality predicate `pred` for comparisons (default is `eq?`). If `elem` is not in `seq`, -1 is returned.

See also: `nth`, `seq?`. →index →topic

#### 4.344 `init-actions` : procedure/0

Usage: (`init-actions`)

Initialize the action system, signals an error if the action system is not available.

See also: `action`, `has-action-system?`, `action-start`, `action-stop`. →index →topic

#### 4.345 `init-remember` : procedure/0

Usage: (`init-remember`)

Initialize the remember database. This requires the modules 'kvdb and 'db enabled. The database is located at (str+ (sysdir 'z3s5-data) "/remembered.z3kv").

See also: `remember`, `recall-when`, `recall`, `forget`. →index →topic

#### 4.346 `instr` : procedure/2

Usage: (`instr` `s1` `s2`)=> `int`

Return the index of the first occurrence of `s2` in `s1` (from left), or -1 if `s1` does not contain `s2`.

See also: `str?`, `index`. →index →topic



#### 4.347 `int`: procedure/1

Usage: (`int` *n*)=> `int`

Return *n* as an integer, rounding down to the nearest integer if necessary.

See also: `float`. →index

**Warning:** If the number is very large this may result in returning the maximum supported integer number rather than the number as integer. →topic

#### 4.348 `intern`: procedure/1

Usage: (`intern` *s*)=> `sym`

Create a new interned symbol based on string *s*.

See also: `gensym`, `str->sym`, `make-symbol`. →index →topic

#### 4.349 `internalize`: procedure/2

Usage: (`internalize` *arg* *nonce*)

Internalize an external representation of *arg*, using *nonce* for distinguishing between data and code that needs to be evaluated.

See also: `externalize`. →index →topic

#### 4.350 `intrinsic`: procedure/1

Usage: (`intrinsic` *sym*)=> *any*

Attempt to obtain the value that is intrinsically bound to *sym*. Use this function to express the intention to use the pre-defined builtin value of a symbol in the base language.

See also: `bind`, `unbind`. →index

**Warning:** This function currently only returns the binding but this behavior might change in future. →topic

### 4.351 `intrinsic?` : procedure/1

Usage: `(intrinsic? x) => bool`

Return true if `x` is an intrinsic built-in function, nil otherwise. Notice that this function tests the value and not that a symbol has been bound to the intrinsic.

See also: `functional?`, `macro?`, `closure?`. [→index](#)

**Warning: What counts as an intrinsic or not may change from version to version. This is for internal use only.** [→topic](#)

### 4.352 `isa?` : procedure/2

Usage: `(isa? obj class) => bool`

Return true if `obj` is an instance of `class`, nil otherwise.

See also: `supers`. [→index](#) [→topic](#)

### 4.353 `iterate` : procedure/2

Usage: `(iterate it proc)`

Apply `proc` to each argument returned by iterator `it` in sequence, similar to the way `foreach` works. An iterator is a procedure that takes one integer as argument or no argument at all. If no argument is provided, the iterator returns the number of iterations. If an integer is provided, the iterator returns a non-nil value for the given index.

See also: `foreach`, `get-partitions`. [→index](#) [→topic](#)

### 4.354 `kvdb.begin` : procedure/1

Usage: `(kvdb.begin db)`

Begin a key-value database transaction. This can be committed by using `kvdb.commit` and rolled back by `kvdb.rollback`.

See also: `kvdb.commit`, `kvdb.rollback`. [→index](#)

**Warning: Transactions in key-value databases cannot be nested! You have to ensure that there is only one begin...commit pair.** [→topic](#)

**4.355 kvdb.close : procedure/1**

Usage: (`kvdb.close db`)

Close a key-value db.

See also: `kvdb.open`. [→index](#) [→topic](#)

**4.356 kvdb.commit : procedure/1**

Usage: (`kvdb.commit db`)

Commit the current transaction, making any changes made since the transaction started permanent.

See also: `kvdb.rollback`, `kvdb.begin`. [→index](#) [→topic](#)

**4.357 kvdb.db? : procedure/1**

Usage: (`kvdb.db? datum`)=> `bool`

Return true if the given datum is a key-value database, nil otherwise.

See also: `kvdb.open`. [→index](#) [→topic](#)

**4.358 kvdb.forget : procedure/1**

Usage: (`kvdb.forget key`)

Forget the value for `key` if there is one.

See also: `kvdb.set`, `kvdb.get`. [→index](#) [→topic](#)

**4.359 kvdb.forget-everything : procedure/1**

Usage: (`kvdb.forget-everything db`)

Erases all data from the given key-value database `db`, irrecoverably losing ALL data in it.

See also: `kvdb.forget`. [→index](#)

**Warning: This operation cannot be undone! Data for all types of keys is deleted. Permanent data loss is imminent!** [→topic](#)

#### 4.360 kvdb.get : procedure/2 or more

Usage: (kvdb.get db key [other])=> any

Get the value stored at `key` in the key-value database `db`. If the value is found, it is returned. If the value is not found and `other` is specified, then `other` is returned. If the value is not found and `other` is not specified, then nil is returned.

See also: kvdb.set, kvdb.when, kvdb.info, kvdb.open, kvdb.forget, kvdb.close, kvdb.search, remember, recall, forget. →index →topic

#### 4.361 kvdb.info : procedure/2 or more

Usage: (db key [other])=> (str str)

Return a list containing the info string and its fuzzy variant stored for `key` in `db`, `other` when the value for `key` is not found. The default for `other` is nil.

See also: kvdb.get, kvdb.when. →index →topic

#### 4.362 kvdb.open : procedure/1 or more

Usage: (kvdb.open path)=> kvdb-array

Create or open a key-value database at `path`.

See also: kvdb.close. →index →topic

#### 4.363 kvdb.rollback : procedure/1

Usage: (kvdb.rollback db)

Rollback the changes made since the last transaction has been started and return the key-value database to its previous state.

See also: kvdb.commit, kvdb.begin. →index →topic

#### 4.364 kvdb.search : procedure/2 or more

Usage: (kvdb.search db s [keytype] [limit] [fuzzer])=> li

Search the key-value database `db` for search expression string `s` for optional `keytype` and return a list of matching keys. The optional `keytype` may be one of '(all str sym int expr), where the default is 'all for

any kind of key. If the optional `limit` is provided, then only `limit` entries are returned. Default limit is `kvdb.default-search-limit`. If `fuzzer` is a function provided, then a fuzzy string search is performed based on applying fuzzer to the search term; default is `nil`.

See also: `kvdb.get`. →index →topic

#### 4.365 `kvdb.set` : procedure/3 or more

Usage: `(kvdb.set db key value [info] [fuzzer])`

Set the `value` for `key` in key-value database `db`. The optional `info` string contains searchable information about the value that may be retrieved with the search function. The optional `fuzzer` must be a function that takes a string and yields a fuzzy variant of the string that can be used for fuzzy search. If no fuzzer is specified, then the default metaphone algorithm is used. Keys for the database must be externalizable but notice that integer keys may provide faster performance.

See also: `kvdb.get`, `kvdb.forget`, `kvdb.open`, `kvdb.close`, `kvdb.search`. →index →topic

#### 4.366 `kvdb.when` : procedure/2 or more

Usage: `(kvdb.when db key [other])=> str`

Get the date in `db` when the entry for `key` was last modified as a date string. If there is no entry for `key`, then `other` is returned. If `other` is not specified and there is no `key`, then `nil` is returned.

See also: `datestr->datelist`, `kvdb.get`, `kvdb.info`. →index →topic

#### 4.367 `last` : procedure/1 or more

Usage: `(last seq [default])=> any`

Get the last element of sequence `seq` or return `default` if the sequence is empty. If `default` is not given and the sequence is empty, an error is raised.

See also: `nth`, `nthdef`, `car`, `list-ref`, `array-ref`, `string-ref`, `1st`, `2nd`, `3rd`, `4th`, `5th`, `6th`, `7th`, `8th`, `9th`, `10th`. →index →topic

#### 4.368 `lcons` : procedure/2

Usage: `(lcons datum li)> list`

Insert `datum` at the end of the list `li`. There may be a more efficient implementation of this in the future. Or, maybe not. Who knows?

See also: `cons`, `list`, `append`, `nreverse`. →index →topic

#### 4.369 `len` : procedure/1

Usage: `(len seq)` => `int`

Return the length of `seq`. Works for lists, strings, arrays, and dicts.

See also: `seq?`. →index →topic

#### 4.370 `let` : macro/1 or more

Usage: `(let args body ...)` => `any`

Bind each pair of symbol and expression in `args` and evaluate the expressions in `body` with these local bindings. Return the value of the last expression in `body`.

See also: `letrec`. →index →topic

#### 4.371 `letrec` : macro/1 or more

Usage: `(letrec args body ...)` => `any`

Recursive let binds the symbol, expression pairs in `args` in a way that makes prior bindings available to later bindings and allows for recursive definitions in `args`, then evaluates the `body` expressions with these bindings.

See also: `let`. →index →topic

#### 4.372 `lighten` : procedure/1

Usage: `(lighten color [amount])` => `(r g b a)`

Return a lighter version of `color`. The optional positive `amount` specifies the amount of lightening (0-255).

See also: `the-color`, `*colors*`, `darken`. →index →topic

### 4.373 **ling.damerau-levenshtein: procedure/2**

Usage: `(ling.damerau-levenshtein s1 s2)=> num`

Compute the Damerau-Levenshtein distance between `s1` and `s2`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#) [→topic](#)

### 4.374 **ling.hamming: procedure/2**

Usage: `(ling-hamming s1 s2)=> num`

Compute the Hamming distance between `s1` and `s2`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#) [→topic](#)

### 4.375 **ling.jaro: procedure/2**

Usage: `(ling.jaro s1 s2)=> num`

Compute the Jaro distance between `s1` and `s2`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#) [→topic](#)

### 4.376 **ling.jaro-winkler: procedure/2**

Usage: `(ling.jaro-winkler s1 s2)=> num`

Compute the Jaro-Winkler distance between `s1` and `s2`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#) [→topic](#)

#### 4.377 **ling.levenshtein:procedure/2**

Usage: `(ling.levenshtein s1 s2)=> num`

Compute the Levenshtein distance between `s1` and `s2`.

See also: `ling.match-rating-compare`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#) [→topic](#)

#### 4.378 **ling.match-rating-codex:procedure/1**

Usage: `(ling.match-rating-codex s)=> str`

Compute the Match-Rating-Codex of string `s`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#) [→topic](#)

#### 4.379 **ling.match-rating-compare:procedure/2**

Usage: `(ling.match-rating-compare s1 s2)=> bool`

Returns true if `s1` and `s2` are equal according to the Match-rating Comparison algorithm, nil otherwise.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#) [→topic](#)

#### 4.380 **ling.metaphone:procedure/1**

Usage: `(ling.metaphone s)=> str`

Compute the Metaphone representation of string `s`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.soundex`. [→index](#) [→topic](#)



#### 4.381 `ling.nysiis:procedure/1`

Usage: `(ling.nysiis s)=> str`

Compute the Nysiis representation of string `s`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.metaphone`, `ling.soundex`. [→index](#) [→topic](#)

#### 4.382 `ling.porter:procedure/1`

Usage: `(ling.porter s)=> str`

Compute the stem of word string `s` using the Porter stemming algorithm.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#) [→topic](#)

#### 4.383 `ling.soundex:procedure/1`

Usage: `(ling.soundex s)=> str`

Compute the Soundex representation of string `s`.

See also: `ling.match-rating-compare`, `ling.levenshtein`, `ling.jaro-winkler`, `ling.jaro`, `ling.hamming`, `ling.damerau-levenshtein`, `ling.match-rating-codex`, `ling.porter`, `ling.nysiis`, `ling.metaphone`, `ling.soundex`. [→index](#) [→topic](#)

#### 4.384 `list:procedure/0 or more`

Usage: `(list [args] ...)=> li`

Create a list from all `args`. The arguments must be quoted.

See also: `cons`. [→index](#) [→topic](#)

#### 4.385 `list->array:procedure/1`

Usage: `(list->array li)=> array`

Convert the list `li` to an array.

See also: [list](#), [array](#), [string](#), [nth](#), [seq?](#). →index →topic

#### 4.386 **list->set: procedure/1**

Usage: ([list->set](#) [li](#))=> [dict](#)

Create a dict containing true for each element of list [li](#).

See also: [make-set](#), [set-element?](#), [set-union](#), [set-intersection](#), [set-complement](#), [set-difference](#), [set?](#), [set-empty](#). →index →topic

#### 4.387 **list->str: procedure/1**

Usage: ([list->str](#) [li](#))=> [string](#)

Return the string that is composed out of the chars in list [li](#).

See also: [array->str](#), [str->list](#), [chars](#). →index →topic

#### 4.388 **list-exists?: procedure/2**

Usage: ([list-exists?](#) [li](#) [pred](#))=> [bool](#)

Return true if [pred](#) returns true for at least one element in list [li](#), nil otherwise.

See also: [exists?](#), [forall?](#), [array-exists?](#), [str-exists?](#), [seq?](#). →index →topic

#### 4.389 **list-forall?: procedure/2**

Usage: ([list-all?](#) [li](#) [pred](#))=> [bool](#)

Return true if predicate [pred](#) returns true for all elements of list [li](#), nil otherwise.

See also: [foreach](#), [map](#), [forall?](#), [array-forall?](#), [str-forall?](#), [exists?](#). →index →topic

#### 4.390 **list-foreach: procedure/2**

Usage: ([list-foreach](#) [li](#) [proc](#))

Apply [proc](#) to each element of list [li](#) in order, for the side effects.

See also: [mapcar](#), [map](#), [foreach](#). →index →topic

**4.391 list-last: procedure/1**

Usage: (`list-last` `li`)=> `any`

Return the last element of `li`.

See also: `reverse`, `nreverse`, `car`, `1st`, `last`. →index →topic

**4.392 list-ref: procedure/2**

Usage: (`list-ref` `li` `n`)=> `any`

Return the element with index `n` of list `li`. Lists are 0-indexed.

See also: `array-ref`, `nth`. →index →topic

**4.393 list-reverse: procedure/1**

Usage: (`list-reverse` `li`)=> `li`

Create a reversed copy of `li`.

See also: `reverse`, `array-reverse`, `str-reverse`. →index →topic

**4.394 list-slice: procedure/3**

Usage: (`list-slice` `li` `low` `high`)=> `li`

Return the slice of the list `li` starting at index `low` (inclusive) and ending at index `high` (exclusive).

See also: `slice`, `array-slice`. →index →topic

**4.395 list?: procedure/1**

Usage: (`list?` `obj`)=> `bool`

Return true if `obj` is a list, nil otherwise.

See also: `cons?`, `atom?`, `null?`. →index →topic

#### 4.396 `load` : procedure/1 or more

Usage: (`load prefix [fi]`)

Loads the Lisp file at `fi` as a library or program with the given `prefix`. If only a prefix is specified, `load` attempts to find a corresponding file at path (str+ (sysdir 'z3s5-data) "/prg/prefix/prefix.lisp"). Loading binds all non-global toplevel symbols of the definitions in file `fi` to the form `prefix.symbol` and replaces calls to them in the definitions appropriately. Symbols starting with "" *such as* `cancel*` are not modified. To give an example, if `fi` contains a definition (`defun bar ...`) and the prefix is `'foo`, then the result of the import is equivalent to (`defun foo.bar ...`), and so on for any other definitions. The importer preorder-traverses the source and looks for `setq` and `lambdas` after macro expansion has taken place. By convention, the entry point of executable programs is a function (`run`) so the loaded program can be executed with the command (`prefix.run`).

See also: `include`, `global-sym?`. →index →topic

#### 4.397 `load-zimage` : procedure/1 or more

Usage: (`load-zimage fi`)=> `li`

Load the zimage file `fi`, if possible, and return a list containing information about the zimage after it has been loaded. If the zimage fails the semantic version check, then an error is raised.

See also: `save-zimage`, `run-zimage`, `zimage-loadable?`. →index →topic

#### 4.398 `macro?` : procedure/1

Usage: (`macro? x`)=> `bool`

Return true if `x` is a macro, nil otherwise.

See also: `functional?`, `intrinsic?`, `closure?`, `functional-arity`, `functional-has-rest?`. →index →topic

#### 4.399 `make` : macro/2

Usage: (`make name props`)

Create a new record (struct instance) of struct `name` (unquoted) with properties `props`. Each clause in `props` must be a list of property name and initial value.

See also: `make*`, `defstruct`. →index →topic

#### 4.400 **make\*** : macro/1 or more

Usage: (`make*` `name` `prop1` ...)

Create a new record (struct instance) of struct `name` (unquoted) with property clauses `prop-1` ... `prop-n`, where each clause is a list of property name and initial value like in `make`.

See also: `make`, `defstruct`. →index →topic

#### 4.401 **make-blob** : procedure/1

Usage: (`make-blob` `n`)=> `blob`

Make a binary blob of size `n` initialized to zeroes.

See also: `blob-free`, `valid?`, `blob-equal?`. →index →topic

#### 4.402 **make-mutex** : procedure/1

Usage: (`make-mutex`)=> `mutex`

Create a new mutex.

See also: `mutex-lock`, `mutex-unlock`, `mutex-rlock`, `mutex-runlock`. →index →topic

#### 4.403 **make-queue** : procedure/0

Usage: (`make-queue`)=> `array`

Make a synchronized queue.

See also: `queue?`, `enqueue!`, `dequeue!`, `glance`, `queue-empty?`, `queue-len`. →index

**Warning:** Never change the array of a synchronized data structure directly, or your warranty is void! →topic

#### 4.404 **make-set** : procedure/0 or more

Usage: (`make-set` [`arg1`] ... [`argn`])=> `dict`

Create a dictionary out of arguments `arg1` to `argn` that stores true for very argument.

See also: `list->set`, `set->list`, `set-element?`, `set-union`, `set-intersection`, `set-complement`, `set-difference`, `set?`, `set-empty?`. →index →topic

#### 4.405 `make-stack` : procedure/0

Usage: `(make-stack)` => `array`

Make a synchronized stack.

See also: `stack?`, `push!`, `pop!`, `stack-empty?`, `stack-len`, `glance`. →index

**Warning: Never change the array of a synchronized data structure directly, or your warranty is void!** →topic

#### 4.406 `make-symbol` : procedure/1

Usage: `(make-symbol s)` => `sym`

Create a new symbol based on string `s`.

See also: `str->sym`. →index →topic

#### 4.407 `map` : procedure/2

Usage: `(map seq proc)` => `seq`

Return the copy of `seq` that is the result of applying `proc` to each element of `seq`.

See also: `seq?`, `mapcar`, `strmap`. →index →topic

#### 4.408 `map-pairwise` : procedure/2

Usage: `(map-pairwise seq proc)` => `seq`

Applies `proc` in order to subsequent pairs in `seq`, assembling the sequence that results from the results of `proc`. Function `proc` takes two arguments and must return a proper list containing two elements. If the number of elements in `seq` is odd, an error is raised.

See also: `map`. →index →topic

#### 4.409 `mapcar` : procedure/2

Usage: `(mapcar li proc)` => `li`

Return the list obtained from applying `proc` to each elements in `li`.

See also: `map`, `foreach`. →index →topic

#### 4.410 max : procedure/1 or more

Usage: (`max` `x1` `x2` ...) => `num`

Return the maximum of the given numbers.

See also: `min`, `minmax`. →index →topic

#### 4.411 member : procedure/2

Usage: (`member` `key` `li`) => `li`

Return the cdr of `li` starting with `key` if `li` contains an element equal? to `key`, nil otherwise.

See also: `assoc`, `equal?`. →index →topic

#### 4.412 memq : procedure/2

Usage: (`memq` `key` `li`)

Return the cdr of `li` starting with `key` if `li` contains an element eq? to `key`, nil otherwise.

See also: `member`, `eq?`. →index →topic

#### 4.413 memstats : procedure/0

Usage: (`memstats`) => `dict`

Return a dict with detailed memory statistics for the system.

See also: `collect-garbage`. →index →topic

#### 4.414 menu-item-checked? : procedure/1

Usage: (`menu-item-checked?` `item`) => `bool`

Return true if `item` is currently checked, nil otherwise.

See also: `set-menu-item-checked`, `set-menu-item-disabled`, `menu-item-disabled?`, `get-menu-item-label`, `set-menu-item-label`, `new-menu*`, `new-menu`, `new-menu-item`, `new-menu-item-separator`. →index →topic

#### 4.415 menu-item-disabled? : procedure/1

Usage: (menu-item-disabled? item)=> bool

Return true if *item* is currently disabled, nil otherwise.

See also: set-menu-item-disabled, set-menu-item-checked, menu-item-checked?, get-menu-item-label, set-menu-item-label, new-menu\*, new-menu, new-menu-item, new-menu-item-separator. →index →topic

#### 4.416 methods : procedure/1

Usage: (methods obj)=> li

Return the list of methods of *obj*, which must be a class, object, or class name.

See also: has-method?, new, props, prop, setprop, has-prop?. →index →topic

#### 4.417 min : procedure/1 or more

Usage: (min x1 x2 ...)=> num

Return the minimum of the given numbers.

See also: max, minmax. →index →topic

#### 4.418 minmax : procedure/3

Usage: (minmax pred li acc)=> any

Go through *li* and test whether for each *elem* the comparison (pred elem acc) is true. If so, *elem* becomes *acc*. Once all elements of the list have been compared, *acc* is returned. This procedure can be used to implement generalized minimum or maximum procedures.

See also: min, max. →index →topic

#### 4.419 minute+ : procedure/2

Usage: (minute+ date li n)=> date li

Adds *n* minutes to the given date *date li* in datelist format and returns the new datelist.

See also: sec+, hour+, day+, week+, month+, year+, now. →index →topic



#### 4.420 `mod` : procedure/2

Usage: `(mod x y) => num`

Compute `x` modulo `y`.

See also: `%`, `/`. →index →topic

#### 4.421 `month+` : procedure/2

Usage: `(month+ dateli n) => dateli`

Adds `n` months to the given date `dateli` in datelist format and returns the new datelist.

See also: `sec+`, `minute+`, `hour+`, `day+`, `week+`, `year+`, `now`. →index →topic

#### 4.422 `move-object` : procedure/2

Usage: `(move-object obj position)`

Move the canvas object `obj` to the given `position` list, containing its `x` and `y` coordinates as floats.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `object-disabled?`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index →topic

#### 4.423 `mutex-lock` : procedure/1

Usage: `(mutex-lock m)`

Lock the mutex `m` for writing. This may halt the current task until the mutex has been unlocked by another task.

See also: `mutex-unlock`, `make-mutex`, `mutex-rlock`, `mutex-runlock`. →index →topic

#### 4.424 `mutex-rlock` : procedure/1

Usage: `(mutex-rlock m)`

Lock the mutex `m` for reading. This will allow other tasks to read from it, too, but may block if another task is currently locking it for writing.

See also: `mutex-runlock`, `mutex-lock`, `mutex-unlock`, `make-mutex`. →index →topic

#### 4.425 `mutex-runlock` : procedure/1

Usage: `(mutex-runlock m)`

Unlock the mutex `m` from reading.

See also: `mutex-lock`, `mutex-unlock`, `mutex-rlock`, `make-mutex`. [→index](#) [→topic](#)

#### 4.426 `mutex-unlock` : procedure/1

Usage: `(mutex-unlock m)`

Unlock the mutex `m` for writing. This releases ownership of the mutex and allows other tasks to lock it for writing.

See also: `mutex-lock`, `make-mutex`, `mutex-rlock`, `mutex-runlock`. [→index](#) [→topic](#)

#### 4.427 `nconc` : procedure/0 or more

Usage: `(nconc li1 li2 ...)=> li`

Concatenate `li1`, `li2`, and so forth, like with `append`, but destructively modifies `li1`.

See also: `append`. [→index](#) [→topic](#)

#### 4.428 `new` : macro/1 or more

Usage: `(new class [props] ...)`

Create a new object of class `class` with initial property bindings `props` clauses as remaining arguments. Each `props` clause must be a list of the form `(sym value)`, where `sym` is a symbol and `value` is evaluated first before it is assigned to `sym`.

See also: `defclass`. [→index](#) [→topic](#)

#### 4.429 `new-app-tabs` : procedure/0 or more

Usage: `(new-app-tabs tab-item ...)=> int`

Create a new application tabs, which allow users to choose different items within an application.

See also: `new-doc-tabs`, `new-tabitem`, `new-tabitem-with-icon`. [→index](#) [→topic](#)

#### 4.430 `new-border` : procedure/4 or more

Usage: (`new-border` `top` `bottom` `left` `right` [`obj` ...])=> `int`

Create a new border layout, which is one of the most useful layouts. Any of `top`, `bottom`, `left`, and `right` is put in the respective place (with minimum size) and might also be `nil` for no widget. The remaining canvas objects `obj` are arranged in the center and take maximum size. This allows you e.g. to put a list on the left side of a window, a panel of buttons on the top, and the main content in another container in the center.

See also: `new-container`, `new-container-without-layout`. →index →topic

#### 4.431 `new-button` : procedure/2

Usage: (`new-button` `label` `proc`)=> `int`

Return a new button with the given `label` and without an icon. The callback `proc` is called without arguments when the button is pressed or tapped.

See also: `new-button-with-icon`, `new-hyperlink`, `new-label`. →index →topic

#### 4.432 `new-button-with-icon` : procedure/3

Usage: (`new-button-with-icon` `label` `icon` `proc`)=> `int`

Return a new button the given `label` and `icon`. The callback `proc` is called without arguments when the button is pressed.

See also: `new-button`, `new-icon`, `theme-icon`. →index →topic

#### 4.433 `new-center-layout` : procedure/0

Usage: (`new-center-layout`)=> `int`

Create a new center layout, which centers container elements (possibly overlapping). This may be used for drawing centered on the window, for example.

See also: `new-form`, `append-form`, `new-spacer`, `new-hbox-layout`, `new-vbox-layout`, `new-grid-layout`, `new-grid-wrap-layout`, `new-form-layout`, `new-stack-layout`, `new-container`. →index →topic

#### 4.434 `new-check` : procedure/2

Usage: (`new-check` `title` `proc`)=> `int`

Create and return a new check box with the given `title` string and a callback procedure `proc`. The callback `proc` is called with the new state of the check box as bool when it has changed.

See also: `new-choice`. [→index](#) [→topic](#)

#### 4.435 `new-choice` : procedure/3

Usage: (`new-choice` `selector` `string-list` `proc`)=> `int`

Create and return a new choice representing choices in `string-list`. If `selector` is 'radio-group, a group of radio buttons is created with options in `string-list`. If `selector` is 'select, a more compact selection menu is created with the options in `string-list`. The callback `proc` takes a string that represents the choice that has been selected.

See also: `new-check`. [→index](#) [→topic](#)

#### 4.436 `new-circle` : procedure/1 or more

Usage: (`new-circle` `fill-color` [`pos1`] [`pos2`] [`stroke-color`] [`stroke-width`])=> `int`

Draw and return a circle with the given NRGBA `fill-color`. If the optional `pos1` and `pos2` position lists of `x` and `y` coordinates in floats are given, then the circle is drawn inside the rectangle defined by these positions. The optional `stroke-color` and `stroke-width` arguments determine the outline of the circle. Notice that circle's size and position may be set by the layout of the container, so to set these manually using `pos1` and `pos2` you need to make sure the underlying container has no such layout.

See also: `new-rectangle`, `new-line-new-text`. [→index](#) [→topic](#)

#### 4.437 `new-combined-string-validator` : procedure/1 or more

Usage: (`new-combined-string-validator` `validator-1` [...`validator-n`])=> `int`

Combine validators `validator-1` to `validator-n` into a combined string validator and return it.

See also: `set-entry-validator`, `new-validator`, `new-regexp-validator`, `new-time-validator`, `set-object-on-validation-change-callback`, `validate-object`. [→index](#) [→topic](#)

#### 4.438 **new-container : procedure/1 or more**

Usage: (**new-container** *layout* *obj* ...) => **int**

Create a new container with the given *layout* and various canvas objects *obj* arranged by the layout.

See also: **new-container-without-layout**, **new-border**. →index →topic

#### 4.439 **new-container-without-layout : procedure/0 or more**

Usage: (**new-container-without-layout** *obj* ...) => **int**

Create a new container without a layout (overlapping objects) with the given canvas objects *obj*.

See also: **new-container**, **new-border**. →index →topic

#### 4.440 **new-doc-tabs : procedure/0 or more**

Usage: (**new-doc-tabs** *tab-item* ...) => **int**

Create new document tabs, which allow users to choose different items in a window (not the application as a whole like *app-tabs*).

See also: **new-app-tabs**, **new-tabitem**, **new-tabitem-with-icon**. →index →topic

#### 4.441 **new-entry : procedure/1**

Usage: (**new-entry** [*selector*]) => **int**

Create a new text entry field based on the optional *selector* symbol. *selector* can be a symbol in '(single-line multi-line password). The default is 'single-line.

See also: *set-entry-on-change-callback*, *set-entry-validator*, *entry-accepts-tab?*, *get-entry-cursor-pos*, *set-entry-cursor-row*, *set-entry-cursor-column*, *set-entry-on-cursor-change-callback*, *get-entry-cursor*, *get-entry-selected-text*, *set-entry-min-rows-visible*, *set-entry-place-holder*, *set-entry-text*. →index →topic

#### 4.442 **new-form : procedure/0**

Usage: (**new-form**)

Return a new form container, which orders widgets in rows, where each row has a label and a widget whose columns are aligned with the other rows. Use *append-form* to add label and widgets.

See also: [append-form](#). [→index](#) [→topic](#)

#### 4.443 new-form-layout : procedure/0

Usage: ([new-form-layout](#))=> [int](#)

Create a form layout, which arranges elements in two columns per row, where the columns are aligned.

See also: [new-form](#), [append-form](#), [new-spacer](#), [new-hbox-layout](#), [new-vbox-layout](#), [new-grid-layout](#), [new-grid-wrap-layout](#), [new-center-layout](#), [new-stack-layout](#), [new-container](#). [→index](#) [→topic](#)

#### 4.444 new-grid-layout : procedure/1

Usage: ([new-grid-layout](#) [n](#))=> [int](#)

Create a new grid layout, which arranges elements in [n](#) columns.

See also: [new-spacer](#), [new-hbox-layout](#), [new-vbox-layout](#), [new-grid-wrap-layout](#), [new-form-layout](#), [new-center-layout](#), [new-stack-layout](#), [new-container](#). [→index](#) [→topic](#)

#### 4.445 new-grid-wrap-layout : procedure/2

Usage: ([new-grid-wrap-layout](#) [width](#) [height](#))=> [int](#)

Create a new grid wrap layout, which arranges elements such that each element has the given [width](#) and [height](#), and wraps lines based on the size of the parent container.

See also: [new-spacer](#), [new-hbox-layout](#), [new-vbox-layout](#), [new-grid-layout](#), [new-form-layout](#), [new-center-layout](#), [new-stack-layout](#), [new-container](#). [→index](#) [→topic](#)

#### 4.446 new-hbox-layout : procedure/0

Usage: ([new-hbox-layout](#))=> [int](#)

Create a new horizontal box layout, which lays out container elements horizontally.

See also: [new-spacer](#), [new-vbox-layout](#), [new-grid-layout](#), [new-grid-wrap-layout](#), [new-form-layout](#), [new-center-layout](#), [new-stack-layout](#), [new-container](#). [→index](#) [→topic](#)

#### 4.447 `new-hyperlink`: procedure/2

Usage: (`new-hyperlink` `label` `url`)=> `int`

Create a new hyperlink with given `label` string and an `url` string as embedded link. A hyperlink looks like a label with link style; when it is clicked, the `url` is called by the default operating system mechanism for opening urls. Using hyperlinks might be disallowed by the host system configuration `HyperlinksAllowed` and may be re-written by the host system using the `CheckHyperlinks` function. If `HyperlinksAllowed` is false in the active GUI config of the host, this function raises an error. It also parses the given `url` and will raise an error if it does not represent a valid URL.

See also: `new-button`, `new-label`. →index

**Warning: Allowing the host to open hyperlinks usually launches a web browser and the details depend on the operating system. There is an added security risk!** →topic

#### 4.448 `new-icon`: procedure/1

Usage: (`new-icon` `resource`)=> `int`

Create a new icon from `resource`, which must be suitable to create an image.

See also: `theme-icon`. →index →topic

#### 4.449 `new-image-from-file`: procedure/1

Usage: (`new-image-from-file` `path`)=> `int`

Create and return a new image from the image file at `path`, which must be a PNG file.

See also: `new-image-from-resource`. →index →topic

#### 4.450 `new-image-from-resource`: procedure/1

Usage: (`new-image-from-resource` `resource`)=> `int`

Create and return a new image from the given `resource`.

See also: `new-image-from-file`, `theme-icon`. →index →topic

#### 4.451 `new-label`: procedure/1

Usage: (`new-label` `str`)=> `int`

Creates a new text label with string `str`.

See also: `set-label-text`. →index →topic

#### 4.452 `new-line`: procedure/1 or more

Usage: (`new-line` `fill-color` [`pos1`] [`pos2`] [`stroke-color`] [`stroke-width`])=> `int`

Draw and return a line with the given NRGBA `fill-color` from optional position `pos1` to position `pos2`, where these are lists of `x` and `y` coordinates as floats. The optional `stroke-color` and `stroke-width` determines the outer edges of the line.

See also: `new-circle`, `new-rectangle`, `new-text`. →index →topic

#### 4.453 `new-list`: procedure/3

Usage: (`new-list` `len-proc` `prep-proc` `update-proc`)=> `int`

Create a new list display. A list consists of rows of simple items like labels. The `len-proc` must be a procedure without arguments returning the length of the list as integer. The `prep-proc` must be a procedure without arguments that returns a canvas object (i.e. a label or other widgets) representing a template for a single list item. The `update-proc` must be a procedure that receives the ID of a canvas object (given by the template) and the 0-based list index as arguments. This procedure then should modify the canvas object with `ID` to display the given list item at the index. See the GUI examples on how to use this function.

See also: `new-table`, `new-tree`. →index →topic

#### 4.454 `new-main-menu`: procedure/1 or more

Usage: (`new-main-menu` `menu` ...)=> `int`

Return a new main menu with the given menus. A main menu displays a menubar for a window on some desktop platforms but it may also be displayed in other ways.

See also: `new-menu`, `new-menu*`. →index →topic



#### 4.455 `new-menu` : procedure/1

Usage: (`new-menu menu*`)=> `int`

Create a new visible menu widget from the abstract `menu*` created by `new-menu*`.

See also: `new-menu*`, `new-main-menu`. →index →topic

#### 4.456 `new-menu*` : procedure/1 or more

Usage: (`new-menu* label [item...]`)=> `int`

Make a new abstract menu with given `label` and arbitrary menu items `item...` following. The starred function is used to define a menu but is not bound to any particular way of displaying it (popup-menu, normal menu, main menu). Use `new-menu` and `new-main-menu` to create visible menus and menu bars based on such abstract menus.

See also: `refresh-menu*`, `new-menu`, `new-main-menu`. →index →topic

#### 4.457 `new-menu-item` : procedure/2 or more

Usage: (`new-menu-item str proc [selector...]`)=> `int`

Create a new menu item with given label `str` and callback `proc`, which takes no arguments. The optional `selector` symbol may be one of: 'is-quit - the item is the application Quit menu item (this is dealt with differently by operating system requirements), 'is-separator - the item is a menu item separator and the label string is ignored (redundent, use `new-menu-item-separator` instead), 'disabled - the menu item is disabled, or 'checked - the menu item is checked.

See also: `set-menu-item-checked`, `menu-item-checked?`, `set-menu-item-disabled`, `menu-item-disabled?`, `get-menu-item-label`, `set-menu-item-label`, `new-menu*`, `new-menu`, `new-menu-item-separator`. →index →topic

#### 4.458 `new-menu-item-separator` : procedure/0

Usage: (`new-menu-item-separator`)=> `int`

Return a new menu item separator, which is a menu item without callback and label that displays a separator between menu items in menus.

See also: `new-menu-item`. →index →topic

#### 4.459 `new-progress-bar` : procedure/0

Usage: (`new-progress-bar`)=> `int`

Create a new progress bar whose default minimum is 0.0 and maximum is 1.0.

See also: `set-progress-bar`, `new-infinite-progress-bar`, `get-progress-bar-value`.  
→index →topic

#### 4.460 `new-raster-with-pixels` : procedure/1

Usage: (`new-raster-with-pixels pixel-proc`)=> `int`

Create a new raster image generated dynamically by the given `pixel-proc`. The `pixel-proc` takes `x` and `y` pixel coordinates and the `width` and `height` of the image in pixels, and returns the color of the pixel `x, y` as a color list of the form '(red green blue [alpha]) where `alpha` is optional. Notice that specifying the color of each pixel can be very CPU-intensive for larger images, so optimizations might be necessary.

See also: `new-image-from-file`. →index →topic

#### 4.461 `new-rectangle` : procedure/1 or more

Usage: (`new-rectangle fill-color [width height] [position] [stroke-color] [stroke-width] [corner-radius]`)=> `int`

Draw and return a rectangle with the given NRGBA `fill-color`. The optional int `width` and `height` arguments set the width and height of the rectangle explicitly (otherwise they are 1). The optional `position` argument must be a list of `x` and `y` coordinates as floats. The optional `stroke-color` and `stroke-width` arguments determine the color and width of the outline of the rectangle, and the optional `corner-radius` defines how rounded the rectangle is. Notice that the rectangle's size and position can be set by the layout of the container, so to set it manually you need to make sure the underlying container has no layout that positions or resizes the rectangle.

See also: `new-circle`, `new-line`, `new-text`. →index →topic

#### 4.462 `new-regexp-validator` : procedure/2

Usage: (`new-regexp-validator regexp reason`)=> `int`

Create a new string validator from the `regexp` string, which must be a valid regular expression in Go's regexp syntax. The `reason` string is displayed to the user by widgets like `entry` when the validation fails.

See also: `set-entry-validator`, `new-validator`, `new-combined-string-validator`, `new-time-validator`, `set-object-on-validation-change-callback`, `validate-object`.  
→index →topic

#### 4.463 `new-shortcut` : procedure/1 or more

Usage: (`new-shortcut` `key` [`mod...`])=> `int`

Create a new keyboard shortcut based on symbol or string `key` and a number of modifier keys `mod`. If multiple non-modifier keys are present, only the last one is taken. However, multiple modifier keys are possible. Possible modifiers are symbols or corresponding strings in '(shift control alt suprt). Possible keys are in '(escape return tab backspace insert delete right left down up page-up page-down home end f1 f2 f3 f4 f5 f6 f7 f8 f9 f10 f11 f12 enter 0 1 2 3 4 5 6 7 8 9 key0 key1 key2 key3 key4 key5 key6 key7 key8 key9 a b c d e f g h i j k l m n o p q r s t u v w x y z space tick comma minus period slash backslash left-bracket right-bracket semicolon equal asterisk plus back-tick) and their string variants.

See also: `new-window`. →index →topic

#### 4.464 `new-slider` : procedure/3

Usage: (`new-slider` `min` `max` `proc`)=> `int`

Create a new slider that allows users to adjust numerical values. The `min` and `max` arguments must be floats. The procedure `proc` takes the current slider float value as argument and is called when the slider changes.

See also: `set-slider-value`. →index →topic

#### 4.465 `new-spacer` : procedure/0

Usage: (`new-spacer`)=> `int`

Create a new spacer, which adjusts size dynamically by taking up space and displaying nothing. Use this to fill containers e.g. to right align a widget.

See also: `new-hbox-layout`, `new-vbox-layout`, `new-grid-layout`, `new-grid-wrap-layout`, `new-form-layout`, `new-center-layout`, `new-stack-layout`, `new-container`. →index →topic

#### 4.466 `new-stack-layout`: procedure/0

Usage: (`new-stack-layout`)=> `int`

Create a new stack layout that stacks container elements on top of each other, overlapping. This may be used for drawing, for example.

See also: `new-form`, `append-form`, `new-spacer`, `new-hbox-layout`, `new-vbox-layout`, `new-grid-layout`, `new-grid-wrap-layout`, `new-form-layout`, `new-center-layout`, `new-container`. →index →topic

#### 4.467 `new-struct`: procedure/2

Usage: (`new-struct` `name` `li`)

Defines a new structure `name` with the properties in the a-list `li`. Structs are more lightweight than classes and do not allow for inheritance. Instances of structs (“records”) are arrays.

See also: `defstruct`. →index →topic

#### 4.468 `new-tabitem`: procedure/2

Usage: (`new-tabitem` `title` `obj`)=> `int`

Create a new tab item for use in app-tabs and doc-tabs with a `title` and an embedded canvas object `obj` shown when the tab item is selected in the tabs.

See also: `new-tabitem-with-icon`, `new-app-tabs`, `new-doc-tabs`. →index →topic

#### 4.469 `new-tabitem-with-icon`: procedure/3

Usage: (`new-tabitem-with-icon` `title` `icon` `obj`)=> `int`

Create a new tab item for use in app-tabs and doc-tabs with given `title` string, `icon` resource, and embedded canvas object `obj` that shown when the tab item is selected in the tabs.

See also: `new-tabitem`, `new-app-tabs`, `new-doc-tabs`. →index →topic

#### 4.470 `new-table`: procedure/3

Usage: (`new-table` `len-proc` `prep-proc` `update-proc`)=> `int`

Create a new table display. A table consists of a number of rows, each of which has a fixed number of columns such as labels. The `len-proc` must be a procedure without arguments returning the length of the table as integer. The `prep-proc` must be a procedure without arguments that returns a canvas object that represents the table row with updatable columns. The `update-proc` takes the row, column, and ID of a canvas object and updates a table template with the right display for the table cell at `row` and `column`.

See also: `new-list`, `new-tree`. [→index](#) [→topic](#)

#### 4.471 `new-text`: procedure/2

Usage: (`new-text` `str` `color`)=> `int`

Draw and return text with the given string `str` and foreground NRGBA `color`.

See also: `set-text-alignment`, `set-text-size`, `set-text-style`, `new-line`, `new-circle`, `new-rectangle`. [→index](#) [→topic](#)

#### 4.472 `new-text-grid`: procedure/0 or more

Usage: (`new-text-grid` [`<string>`] [`show-line-numbers`|`show-whitespace`|`tab-width` `<int>`])=> `int`

Create a new text grid widget, which displays multiline text with custom background and foreground colors. The optional `string` argument is the initial text of the grid without formatting. The following symbols might be `'show-line-numbers` to turn the line number display on and `'show-whitespace` to display white space characters by special unicode symbols. If the selector `'tab-width` occurs, then it must be immediately followed by an integer for the tabulator width of the text grid in space characters.

See also: `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. [→index](#) [→topic](#)

#### 4.473 `new-time-validator`: procedure/1

Usage: (`new-time-validator` `format-str`)=> `int`

Create a new string validator for time and date based on the given template `format-str`. This validator uses Go's data parsing function and therefore is quite restrictive. Only datetimes entered in exactly the format given (including timezones) validate successfully. To obtain a more relaxed date and time validator, use `new-validator` to create a custom validator with your own parsing or try a `new-regexp-validator`.

See also: `set-entry-validator`, `new-validator`, `new-combined-string-validator`, `new-time-validator`, `new-regexp-validator`, `set-object-on-validation-change-callback`, `validate-object`. →index →topic

#### 4.474 `new-tree`: procedure/4

Usage: (`new-tree` `child-uid-proc` `is-branch-proc` `create-node-proc` `update-node-proc`)=> `int`

Create a new tree display. A tree displays nested branches and leaf nodes. The `child-uid-proc` is a procedure that takes an id string as argument. If the string is empty, it should return a list of top-level branch uid strings. If the string is not empty, it represents an uid; the procedure should then return a list of all child id strings of that branch. This defines the tree's structure. All id strings must be unique to the tree. The `is-branch-proc` takes an id string as argument and should return non-nil if the id represents a branch, nil if it has no children. The `create-node-proc` takes a bool `b` as argument and should return a branch template if `b` is non-nil and a leaf template object if `b` is nil. Finally, the `update-node-proc` is a procedure that takes a node id string, a boolean that is true if the node is a branch, and a node template canvas-object as it is returned by `create-node-proc`. The procedure should fill the template with the display values for the respective node id.

See also: `new-list`, `new-table`. →index →topic

#### 4.475 `new-validator`: procedure/1

Usage: (`new-validator` `proc`)=> `int`

Create a new string validator based on validation procedure `proc`. The procedure `proc` takes a string as argument and returns a string. If the string returned is not the empty string "", then validation fails and the returned string is given as a reason for validation failure. If the empty string is returned, then validation succeeds. If an error occurs in `proc`, then validation fails with the error's error message as reason. Notice that validators are fairly limited and can only be attached to a few validatable objects such as text entry fields. For a more general approach, it might make sense to implement your own validation system based on key press, focus change, and change callbacks of various GUI objects.

See also: [set-entry-validator](#), [new-combined-string-validator](#), [new-regexp-validator](#), [new-time-validator](#), [set-object-on-validation-change-callback](#), [validate-object](#). [→index](#) [→topic](#)

#### 4.476 `new-vbox-layout` : procedure/0

Usage: (`new-vbox-layout`)=> `int`

Create a new vertical box layout, which lays out container elements vertically.

See also: [new-spacer](#), [new-hbox-layout](#), [new-grid-layout](#), [new-grid-wrap-layout](#), [new-form-layout](#), [new-center-layout](#), [new-stack-layout](#), [new-container](#). [→index](#) [→topic](#)

#### 4.477 `new-window` : procedure/1

Usage: (`new-window title`)=> `int`

Create a new window with `title` string and return the window ID. This function raises an error if the host configuration `WindowsAllowed` is not true. In certain embedded uses, creating new windows is not allowed and you should check the documentation how to find a pre-configured window and add user interface elements to it.

See also: [set-window-content](#), [close-window](#), [show-window](#). [→index](#) [→topic](#)

#### 4.478 `nl` : procedure/0

Usage: (`nl`)

Display a newline, advancing the cursor to the next line.

See also: [out](#), [outy](#), [output-at](#). [→index](#) [→topic](#)

#### 4.479 `nonce` : procedure/0

Usage: (`nonce`)=> `str`

Return a unique random string. This is not cryptographically secure but the string satisfies reasonable GUID requirements.

See also: [externalize](#), [internalize](#). [→index](#) [→topic](#)

**4.480 not : procedure/1**

Usage: (`not` `x`)=> `bool`

Return true if `x` is nil, nil otherwise.

See also: `and`, `or`. →index →topic

**4.481 now : procedure/0**

Usage: (`now`)=> `li`

Return the current datetime in UTC format as a list of values in the form '(year month day weekday iso-week) (hour minute second nanosecond unix-nano-second)).

See also: `now-ns`, `datestr`, `time`, `date`→`epoch-ns`, `epoch-ns`→`date``list`. →index →topic

**4.482 now-ms : procedure/0**

Usage: (`now-ms`)=> `num`

Return the relative system time as a call to (`now-ns`) but in milliseconds.

See also: `now-ns`, `now`. →index →topic

**4.483 now-ns : procedure/0**

Usage: (`now-ns`)=> `int`

Return the current time in Unix nanoseconds.

See also: `now`, `time`. →index →topic

**4.484 nreverse : procedure/1**

Usage: (`nreverse` `li`)=> `li`

Destructively reverse `li`.

See also: `reverse`. →index →topic



**4.485 nrgba : procedure/4**

Usage: (`nrgba red green blue alpha`)=> `int`

Create an RGBA color where `red`, `green`, `blue`, and `alpha` are 8-bit uint integers, i.e., values between 0 and 255 (inclusive). Notice that some GUI functions require NRGBA color returned by this function, whereas others require a color list of int values '(red green blue alpha). This is for performance reasons, since it sometimes faster to convert a list to a color on-the-fly and sometimes more convenient to store pre-defined colors for later re-use.

See also: `new-rectangle`, `new-circle`, `new-line`, `new-text`. →index →topic

**4.486 nth : procedure/2**

Usage: (`nth seq n`)=> `any`

Get the `n-th` element of sequence `seq`. Sequences are 0-indexed.

See also: `nthdef`, `list`, `array`, `string`, `1st`, `2nd`, `3rd`, `4th`, `5th`, `6th`, `7th`, `8th`, `9th`, `10th`. →index →topic

**4.487 nth-partition : procedure/3**

Usage: (`nth-partition m k idx`)=> `li`

Return a list of the form (start-offset end-offset bytes) for the partition with index `idx` of `m` into parts of size `k`. The index `idx` as well as the start- and end-offsets are 0-based.

See also: `count-partitions`, `get-partitions`. →index →topic

**4.488 nthdef : procedure/3**

Usage: (`nthdef seq n default`)=> `any`

Return the `n-th` element of sequence `seq` (0-indexed) if `seq` is a sequence and has at least `n+1` elements, default otherwise.

See also: `nth`, `seq?`, `1st`, `2nd`, `3rd`, `4th`, `5th`, `6th`, `7th`, `8th`, `9th`, `10th`. →index →topic

**4.489 null? : procedure/1**

Usage: (**null?** *li*)=> *bool*

Return true if *li* is nil, nil otherwise.

See also: [not](#), [list?](#), [cons?](#). →index →topic

**4.490 num? : procedure/1**

Usage: (**num?** *n*)=> *bool*

Return true if *n* is a number (exact or inexact), nil otherwise.

See also: [str?](#), [atom?](#), [sym?](#), [closure?](#), [intrinsic?](#), [macro?](#). →index →topic

**4.491 object-disabled? : procedure/1**

Usage: (**object-disabled?** *obj*)=> *bool*

Return true if the canvas object *obj* is disabled, nil otherwise.

See also: [disable-object](#), [enable-object](#), [show-object](#), [hide-object](#), [move-object](#), [resize-object](#), [get-object-size](#), [get-object-min-size](#), [get-object-position](#), [object-visible?](#), [refresh-object](#), [new-entry](#), [new-label](#). →index →topic

**4.492 object? : procedure/1**

Usage: (**object?** *obj*)=> *bool*

Return true of *obj* is an object array, nil otherwise.

See also: [class?](#), [isa?](#). →index →topic

**4.493 odd? : procedure/1**

Usage: (**odd?** *n*)=> *bool*

Returns true if the integer *n* is odd, nil otherwise.

See also: [even?](#). →index →topic

#### 4.494 `on-feature` : macro/1 or more

Usage: (`on-feature` `sym` `body` ...) => `any`

Evaluate the expressions of `body` if the Lisp feature `sym` is supported by this implementation, do nothing otherwise.

See also: `feature?`, `*reflect*`. →index →topic

#### 4.495 `open` : procedure/1 or more

Usage: (`open` `file-path` [`modes`] [`permissions`]) => `int`

Open the file at `file-path` for reading and writing, and return the stream ID. The optional `modes` argument must be a list containing one of '(read write read-write) for read, write, or read-write access respectively, and may contain any of the following symbols: 'append to append to an existing file, 'create for creating the file if it doesn't exist, 'exclusive for exclusive file access, 'truncate for truncating the file if it exists, and 'sync for attempting to sync file access. The optional `permissions` argument must be a numeric value specifying the Unix file permissions of the file. If these are omitted, then default values '(read-write append create) and 0640 are used.

See also: `stopen`, `close`, `read`, `write`. →index →topic

#### 4.496 `or` : macro/0 or more

Usage: (`or` `expr1` `expr2` ...) => `any`

Evaluate the expressions until one of them is not nil. This is a logical shortcut or.

See also: `and`. →index →topic

#### 4.497 `out` : procedure/1

Usage: (`out` `expr`)

Output `expr` on the console with current default background and foreground color.

See also: `outy`, `synout`, `synouty`, `output-at`. →index →topic

#### 4.498 outy : procedure/1

Usage: (outy spec)

Output styled text specified in `spec`. A specification is a list of lists starting with 'fg for foreground, 'bg for background, or 'text for unstyled text. If the list starts with 'fg or 'bg then the next element must be a color suitable for (the-color spec). Following may be a string to print or another color specification. If a list starts with 'text then one or more strings may follow.

See also: `*colors*`, `the-color`, `set-color`, `color`, `gfx.color`, `output-at`, `out`. →index →topic

#### 4.499 peek : procedure/4

Usage: (peek b pos end sel)=> num

Read a numeric value determined by selector `sel` from binary blob `b` at position `pos` with endianness `end`. Possible values for endianness are 'little and 'big, and possible values for `sel` must be one of '(bool int8 uint8 int16 uint16 int32 uint32 int64 uint64 float32 float64).

See also: `poke`, `read-binary`. →index →topic

#### 4.500 permission? : procedure/1

Usage: (permission? sym [`default`])=> bool

Return true if the permission for `sym` is set, nil otherwise. If the permission flag is unknown, then `default` is returned. The default for `default` is nil.

See also: `permissions`, `set-permissions`, `when-permission`, `sys`. →index →topic

#### 4.501 permissions : procedure/0

Usage: (permissions)

Return a list of all active permissions of the current interpreter. Permissions are: `load-prelude` - load the init file on start; `load-user-init` - load the local user init on startup, file if present; `allow-unprotect` - allow the user to unprotect protected symbols (for redefining them); `allow-protect` - allow the user to protect symbols from redefinition or unbinding; `interactive` - make the session interactive, this is particularly used during startup to determine whether hooks are installed and feedback is given. Permissions have to generally be set or removed in careful combination with `revoke-permissions`, which redefines symbols and functions.

See also: [set-permissions](#), [permission?](#), [when-permission](#), [sys.](#) →index →topic

#### 4.502 **poke** : procedure/5

Usage: ([poke](#) *b pos end sel n*)

Write numeric value *n* as type *sel* with endianness *end* into the binary blob *b* at position *pos*. Possible values for endianness are 'little and 'big, and possible values for *sel* must be one of '(bool int8 uint8 int16 uint16 int32 uint32 int64 uint64 float32 float64).

See also: [peek](#), [write-binary](#). →index →topic

#### 4.503 **pop!** : macro/1 or more

Usage: ([pop!](#) *sym [def]*)=> *any*

Get the next element from stack *sym*, which must be the unquoted name of a variable, and return it. If a default *def* is given, then this is returned if the queue is empty, otherwise nil is returned.

See also: [make-stack](#), [stack?](#), [push!](#), [stack-len](#), [stack-empty?](#), [glance](#). →index →topic

#### 4.504 **pop-error-handler** : procedure/0

Usage: ([pop-error-handler](#))=> *proc*

Remove the topmost error handler from the error handler stack and return it. For internal use only.

See also: [with-error-handler](#). →index →topic

#### 4.505 **pop-finalizer** : procedure/0

Usage: ([pop-finalizer](#))=> *proc*

Remove a finalizer from the finalizer stack and return it. For internal use only.

See also: [push-finalizer](#), [with-final](#). →index →topic

#### 4.506 **popstacked** : procedure/3

Usage: ([popstacked](#) *dict key default*)

Get the topmost element from the stack stored at `key` in `dict` and remove it from the stack. If the stack is empty or no stack is stored at `key`, then **default** is returned.

See also: `pushstacked`, `getstacked`. →index →topic

#### 4.507 `prin1`: procedure/1

Usage: `(prin1 s)`

Print `s` to the host OS terminal, where strings are quoted.

See also: `princ`, `terpri`, `out`, `outy`. →index →topic

#### 4.508 `princ`: procedure/1

Usage: `(princ s)`

Print `s` to the host OS terminal without quoting strings.

See also: `prin1`, `terpri`, `out`, `outy`. →index →topic

#### 4.509 `print`: procedure/1

Usage: `(print x)`

Output `x` on the host OS console and end it with a newline.

See also: `prin1`, `princ`. →index →topic

#### 4.510 `proc?`: macro/1

Usage: `(proc? arg)=> bool`

Return true if `arg` is a procedure, nil otherwise.

See also: `functional?`, `closure?`, `functional-arity`, `functional-has-rest?`. →index →topic

#### 4.511 `prop`: procedure/2

Usage: `(prop obj slot)=> any`

Return the value in `obj` for property `slot`, or an error if the object does not have a property with that name.

See also: `new`, `isa?`, `setslot`, `object?`, `class-name`, `supers`, `props`, `methods`, `has-slot?`.  
→index →topic

#### 4.512 `props` : procedure/1

Usage: (`props obj`)=> `li`

Return the list of properties of `obj`. An error occurs if `obj` is not a valid object.

See also: `methods`, `has-prop?`, `new`, `prop`, `setprop`. →index →topic

#### 4.513 `protect` : procedure/0 or more

Usage: (`protect [sym] ...`)

Protect symbols `sym`... against changes or rebinding. The symbols need to be quoted. This operation requires the permission 'allow-protect to be set.

See also: `protected?`, `unprotect`, `dict-protect`, `dict-unprotect`, `dict-protected?`, `permissions`, `permission?`, `setq`, `bind`, `interpret`. →index →topic

#### 4.514 `protect-toplevel-symbols` : procedure/0

Usage: (`protect-toplevel-symbols`)

Protect all toplevel symbols that are not yet protected and aren't in the *mutable-toplevel-symbols* dict.

See also: `protected?`, `protect`, `unprotect`, `declare-unprotected`, `declare-volatile`, `when-permission?`, `dict-protect`, `dict-protected?`, `dict-unprotect`. →index →topic

#### 4.515 `protected?` : procedure/1

Usage: (`protected? sym`)

Return true if `sym` is protected, nil otherwise.

See also: `protect`, `unprotect`, `dict-unprotect`, `dict-protected?`, `permission`, `permission?`, `setq`, `bind`, `interpret`. →index →topic

#### 4.516 `prune-task-table` : procedure/0

Usage: (`prune-task-table`)

Remove tasks that are finished from the task table. This includes tasks for which an error has occurred.

See also: `task-remove`, `task`, `task?`, `task-run`. [→index](#) [→topic](#)

#### 4.517 `prune-unneeded-help-entries` : procedure/0

Usage: (`prune-unneeded-help-entries`)

Remove help entries for which no toplevel symbol is defined. This function may need to be called when a module is not being used (e.g. because of a missing build tag) and it is desirable that only help for existing symbols is available.

See also: `find-unneeded-help-entries`, `find-missing-help-entries`, `help`, `*help*`. [→index](#) [→topic](#)

#### 4.518 `push!` : macro/2

Usage: (`push!` `sym elem`)

Put `elem` in stack `sym`, where `sym` is the unquoted name of a variable.

See also: `make-stack`, `stack?`, `pop!`, `stack-len`, `stack-empty?`, `glance`. [→index](#) [→topic](#)

#### 4.519 `push-error-handler` : procedure/1

Usage: (`push-error-handler proc`)

Push an error handler `proc` on the error handler stack. For internal use only.

See also: `with-error-handler`. [→index](#) [→topic](#)

#### 4.520 `push-finalizer` : procedure/1

Usage: (`push-finalizer proc`)

Push a finalizer procedure `proc` on the finalizer stack. For internal use only.

See also: `with-final`, `pop-finalizer`. [→index](#) [→topic](#)



#### 4.521 `pushstacked` : procedure/3

Usage: `(pushstacked dict key datum)`

Push `datum` onto the stack maintained under `key` in the `dict`.

See also: `getstacked`, `popstacked`. →index →topic

#### 4.522 `queue-empty?` : procedure/1

Usage: `(queue-empty? q)=> bool`

Return true if the queue `q` is empty, nil otherwise.

See also: `make-queue`, `queue?`, `enqueue!`, `dequeue!`, `glance`, `queue-len`. →index →topic

#### 4.523 `queue-len` : procedure/1

Usage: `(queue-len q)=> int`

Return the length of the queue `q`.

See also: `make-queue`, `queue?`, `enqueue!`, `dequeue!`, `glance`, `queue-len`. →index

**Warning:** Be advised that this is of limited use in some concurrent contexts, since the length of the queue might have changed already once you've obtained it! →topic

#### 4.524 `queue?` : procedure/1

Usage: `(queue? q)=> bool`

Return true if `q` is a queue, nil otherwise.

See also: `make-queue`, `enqueue!`, `dequeue`, `glance`, `queue-empty?`, `queue-len`. →index →topic

#### 4.525 `rand` : procedure/2

Usage: `(rand prng lower upper)=> int`

Return a random integer in the interval [`lower` .. `upper`], both inclusive, from pseudo-random number generator `prng`. The `prng` argument must be an integer from 0 to 9 (inclusive).

See also: `rnd`, `rndseed`. →index →topic

#### 4.526 `random-color` : procedure/0 or more

Usage: (`random-color` [`alpha`])

Return a random color with optional `alpha` value. If `alpha` is not specified, it is 255.

See also: `the-color`, `*colors*`, `darken`, `lighten`. →index →topic

#### 4.527 `read` : procedure/1

Usage: (`read` `p`)=> `any`

Read an expression from input port `p`.

See also: `input`, `write`. →index →topic

#### 4.528 `read-binary` : procedure/3

Usage: (`read-binary` `p` `buff` `n`)=> `int`

Read `n` or less bytes from input port `p` into binary blob `buff`. If `buff` is smaller than `n`, then an error is raised. If less than `n` bytes are available before the end of file is reached, then the amount `k` of bytes is read into `buff` and `k` is returned. If the end of file is reached and no byte has been read, then 0 is returned. So to loop through this, read into the buffer and do something with it while the amount of bytes returned is larger than 0.

See also: `write-binary`, `read`, `close`, `open`. →index →topic

#### 4.529 `read-string` : procedure/2

Usage: (`read-string` `p` `delstr`)=> `str`

Reads a string from port `p` until the single-byte delimiter character in `delstr` is encountered, and returns the string including the delimiter. If the input ends before the delimiter is encountered, it returns the string up until EOF. Notice that if the empty string is returned then the end of file must have been encountered, since otherwise the string would contain the delimiter.

See also: `read`, `read-binary`, `write-string`, `write`, `read`, `close`, `open`. →index →topic

#### 4.530 read-zimage : procedure/2

Usage: (read-zimage in fi)

Reads and evaluates the zimage in stream `in` from file `fi`. The file `fi` argument is used in error messages. This procedure raises errors when the zimage is malformed or the version check fails.

See also: `load-zimage`, `run-zimage`, `zimage-header`. →index →topic

#### 4.531 readall : procedure/1

Usage: (readall stream)=> sexpr

Read all data from `stream` and return it as an sexpr.

See also: `read`, `write`, `open`, `close`. →index

#### 4.532 readall-str : procedure/1 or more

Usage: (readall-str p [buffsize])=> str

Read all content from port `p` as string. This method may trigger an error if the content in the stream is not a valid UTF-8 string. The optional `buffsize` argument determines the size of the internal buffer.

See also: `readall`, `read-binary`, `read`. →index

#### 4.533 recall : procedure/1 or more

Usage: (recall key [notfound])=> any

Obtain the value remembered for `key`, `notfound` if it doesn't exist. If `notfound` is not provided, then `nil` is returned in case the value for `key` doesn't exist.

See also: `recall-when`, `recall-info`, `recollect`, `remember`, `forget`. →index →topic

#### 4.534 recall-info : procedure/1 or more

Usage: (recall-info key [notfound])=> (str str)

Return a list containing the info string and its fuzzy version for a remembered value with the given `key`, `notfound` if no value for `key` was found. The default for `notfound` is `nil`.

See also: `recall-when`, `recall`, `recall-when`, `recollect`, `remember`, `forget`. →index →topic

#### 4.535 `recall-when` : procedure/1 or more

Usage: (`recall-when` *key* [*notfound*])=> *datestr*

Obtain the date string when the value for *key* was last modified by `remember` (`set`), *notfound* if it doesn't exist. If *notfound* is not provided, then `nil` is returned in case there is no value for *key*.

See also: `recall`, `datestr->datelist`, `recall-info`, `remember`, `forget`. →index →topic

#### 4.536 `recollect` : procedure/1 or more

Usage: (`recollect` *s* [*keytype*] [*limit*] [*fuzzer*])=> *li*

Search for remembered items based on search query *s* and return a list of matching keys. The optional *keytype* parameter must be one of '(all str sym int expr), where the default is 'all for all kinds of keys. Up to *limit* results are returned, default is `kvdb.default-search-limit`. The optional *fuzzer* procedure takes a word string and yields a 'fuzzy' version of it. If *fuzzer* is specified and a procedure, then a fuzzy search is performed.

See also: `kvdb.search`, `recall`, `recall-info`, `recall-when`, `remember`. →index →topic

#### 4.537 `record?` : procedure/1

Usage: (`record?` *s*)=> *bool*

Returns true if *s* is a struct record, i.e., an instance of a struct; `nil` otherwise. Notice that records are not really types distinct from arrays, they simply contain a marker '%record as first element. With normal use no confusion should arise. Since the internal representation might change, you ought not use ordinary array procedures for records.

See also: `struct?`, `defstruct`. →index →topic

#### 4.538 `refresh-main-menu` : procedure/1

Usage: (`refresh-main-menu` *main-menu*)

Refresh the given *main-menu* display. This should be called after some submenus or menu items in the main menu have changed.

See also: `new-main-menu`, `refresh-menu*`. →index →topic

#### 4.539 `refresh-menu*` : procedure/1

Usage: (`refresh-menu*` `menu`)

Refresh the given `menu` after a change was made that has a visual impact. This will refresh the menu widget in which this abstract menu occurs.

See also: `refresh-main-menu`, `new-menu*`. →index →topic

#### 4.540 `refresh-object` : procedure/1

Usage: (`refresh-object` `obj`)

Refresh the canvas object `obj`, causing the graphical display to be re-drawn as soon as possible. This may be needed if the object's state has changed.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `object-visible?`, `get-object-position`, `new-entry`, `new-label`. →index →topic

#### 4.541 `register-action` : procedure/1

Usage: (`register-action` `action`)

Register the `action` which makes it available for processing by the host system. Use `get-action` to obtain an action clone that can be started.

See also: `action`, `has-action-system?`, `action-start`, `action-stop`. →index →topic

#### 4.542 `remember` : procedure/2

Usage: (`remember` `key` `value` [`info`] [`fuzzer`])

Persistently remember `value` by given `key`. See `kvdb.set` for the optional `info` and `fuzzer` arguments.

See also: `recall`, `forget`, `kvdb.set`, `recall-when`, `recall-info`, `recollect`. →index →topic

#### 4.543 `remove-duplicates` : procedure/1

Usage: (`remove-duplicates` `seq`)=> `seq`

Remove all duplicates in sequence `seq`, return a new sequence with the duplicates removed.

See also: `seq?`, `map`, `foreach`, `nth`. [→index](#) [→topic](#)

#### 4.544 `remove-hook` : procedure/2

Usage: (`remove-hook` `hook` `id`)=> `bool`

Remove the symbolic or numeric `hook` with `id` and return true if the hook was removed, nil otherwise.

See also: `add-hook`, `remove-hooks`, `replace-hook`. [→index](#) [→topic](#)

#### 4.545 `remove-hook-internal` : procedure/2

Usage: (`remove-hook-internal` `hook` `id`)

Remove the hook with ID `id` from numeric `hook`.

See also: `remove-hook`. [→index](#)

**Warning: Internal use only.** [→topic](#)

#### 4.546 `remove-hooks` : procedure/1

Usage: (`remove-hooks` `hook`)=> `bool`

Remove all hooks for symbolic or numeric `hook`, return true if the hook exists and the associated procedures were removed, nil otherwise.

See also: `add-hook`, `remove-hook`, `replace-hook`. [→index](#) [→topic](#)

#### 4.547 `rename-action` : procedure/2

Usage: (`rename-action` `id` `new-name`)=> `bool`

Rename a registered action with given `id`, or rename the action given as `id`, to `new-name`. If the operation succeeds, it returns true, otherwise it returns nil.

See also: `change-action-prefix`, `change-all-action-prefixes`, `get-action`, `has-action?`, `action`. [→index](#) [→topic](#)

#### 4.548 `replace-hook` : procedure/2

Usage: (`replace-hook` `hook` `proc`)

Remove all hooks for symbolic or numeric `hook` and install the given `proc` as the only hook procedure.

See also: `add-hook`, `remove-hook`, `remove-hooks`. →index →topic

#### 4.549 `reset-color` : procedure/0

Usage: (`reset-color`)

Reset the 'text and 'back colors of the display to default values. These values are not specified in the color database and depend on the runtime implementation. Other colors like 'gfx or 'frame are not affected.

See also: `set-color`, `color`, `the-color`, `with-colors`. →index →topic

#### 4.550 `resize-object` : procedure/2

Usage: (`resize-object` `obj` `width` `height`)

Resize canvas object `obj` to the given `width` and `height` as floats.

See also: `disable-object`, `enable-object`, `show-object`, `hide-object`, `object-disabled?`, `move-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index →topic

#### 4.551 `reverse` : procedure/1

Usage: (`reverse` `seq`)=> `sequence`

Reverse a sequence non-destructively, i.e., return a copy of the reversed sequence.

See also: `nth`, `seq?`, `1st`, `2nd`, `3rd`, `4th`, `6th`, `7th`, `8th`, `9th`, `10th`, `last`. →index →topic

#### 4.552 `rnd` : procedure/0

Usage: (`rnd` `prng`)=> `num`

Return a random value in the interval [0, 1] from pseudo-random number generator `prng`. The `prng` argument must be an integer from 0 to 9 (inclusive).

See also: [rand](#), [rndseed](#). [→index](#) [→topic](#)

#### 4.553 [rndseed](#) : procedure/1

Usage: ([rndseed](#) [prng](#) [n](#))

Seed the pseudo-random number generator [prng](#) (0 to 9) with 64 bit integer value [n](#). Larger values will be truncated. Seeding affects both the [rnd](#) and the [rand](#) function for the given [prng](#).

See also: [rnd](#), [rand](#). [→index](#) [→topic](#)

#### 4.554 [rplaca](#) : procedure/2

Usage: ([rplaca](#) [li](#) [a](#))=> [li](#)

Destructively mutate [li](#) such that its car is [a](#), return the list afterwards.

See also: [rplacd](#). [→index](#) [→topic](#)

#### 4.555 [run-at](#) : procedure/2

Usage: ([run-at](#) [date](#) [repeater](#) [proc](#))=> [int](#)

Run procedure [proc](#) with no arguments as task periodically according to the specification in [spec](#) and return the task ID for the periodic task. Herbey, [date](#) is either a datetime specification or one of '(now skip next-minute next-quarter next-halfhour next-hour in-2-hours in-3-hours tomorrow next-week next-month next-year)', and [repeater](#) is nil or a procedure that takes a task ID and unix-epoch-nanoseconds and yields a new unix-epoch-nanoseconds value for the next time the procedure shall be run. While the other names are self-explanatory, the 'skip' specification means that the task is not run immediately but rather that it is first run at ([repeater](#) -1 (now)). Timing resolution for the scheduler is about 1 minute. Consider using interrupts for periodic events with smaller time resolutions. The scheduler uses relative intervals and has 'drift'.

See also: [task](#), [task-send](#). [→index](#)

**Warning: Tasks scheduled by [run-at](#) are not persistent! They are only run until the system is shutdown.** [→topic](#)

#### 4.556 [run-hook](#) : procedure/1

Usage: ([run-hook](#) [hook](#))



Manually run the hook, executing all procedures for the hook.

See also: [add-hook](#), [remove-hook](#). [→index](#) [→topic](#)

#### 4.557 `run-hook-internal` : procedure/1 or more

Usage: (`run-hook-internal` `hook` [`args`] ...)

Run all hooks for numeric hook ID `hook` with `args...` as arguments.

See also: [run-hook](#). [→index](#)

**Warning: Internal use only.** [→topic](#)

#### 4.558 `run-selftest` : procedure/0

Usage: (`run-selftest`)

Run a self test of the Z3S5 Lisp system and report errors to standard output.

See also: [help](#), [testing](#). [→index](#) [→topic](#)

#### 4.559 `run-zimage` : procedure/1 or more

Usage: (`run-zimage` `fi`)

Load the zimage file `fi` and start it at the designated entry point. Raises an error if the zimage version is not compatible or the zimage cannot be run.

See also: [load-zimage](#), [save-zimage](#), [zimage-runable?](#), [zimage-loadable?](#). [→index](#) [→topic](#)

#### 4.560 `save-zimage` : procedure/1 or more

Usage: (`save-zimage` `min-version` `info` `entry-point` `fi`)=> `int`

Write the current state of the system as a zimage to file `fi`. If the file already exists, it is overwritten. The `min-version` argument designates the minimum system version required to load the zimage. The `info` argument should be a list whose first argument is a human-readable string explaining the purpose of the zimage and remainder is user data. The `entry-point` is either nil or an expression that can be evaluated to start the zimage after it has been loaded with `run-zimage`.

See also: [load-zimage](#), [current-zimage](#), [dump](#), [run-zimage](#), [zimage-loadable?](#), [zimage-runable?](#), [externalize](#). [→index](#) [→topic](#)

**4.561 sec+ : procedure/2**

Usage: (`sec+ dateli n`)=> `dateli`

Adds `n` seconds to the given date `dateli` in datelist format and returns the new datelist.

See also: `minute+`, `hour+`, `day+`, `week+`, `month+`, `year+`, `now`. →index →topic

**4.562 semver.build : procedure/1**

Usage: (`semver.build s`)=> `str`

Return the build part of a semantic versioning string.

See also: `semver.canonical`, `semver.major`, `semver.major-minor`. →index →topic

**4.563 semver.canonical : procedure/1**

Usage: (`semver.canonical s`)=> `str`

Return a canonical semver string based on a valid, yet possibly not canonical version string `s`.

See also: `semver.major`. →index →topic

**4.564 semver.compare : procedure/2**

Usage: (`semver.compare s1 s2`)=> `int`

Compare two semantic version strings `s1` and `s2`. The result is 0 if `s1` and `s2` are the same version, -1 if `s1 < s2` and 1 if `s1 > s2`.

See also: `semver.major`, `semver.major-minor`. →index →topic

**4.565 semver.is-valid? : procedure/1**

Usage: (`semver.is-valid? s`)=> `bool`

Return true if `s` is a valid semantic versioning string, nil otherwise.

See also: `semver.major`, `semver.major-minor`, `semver.compare`. →index →topic

**4.566 `semver.major` : procedure/1**

Usage: (`semver.major` *s*)=> *str*

Return the major part of the semantic versioning string.

See also: `semver.major-minor`, `semver.build`. →index →topic

**4.567 `semver.major-minor` : procedure/1**

Usage: (`semver.major-minor` *s*)=> *str*

Return the major.minor prefix of a semantic versioning string. For example, (`semver.major-minor` “v2.1.4”) returns “v2.1”.

See also: `semver.major`, `semver.build`. →index →topic

**4.568 `semver.max` : procedure/2**

Usage: (`semver.max` *s1 s2*)=> *str*

Canonicalize *s1* and *s2* and return the larger version of them.

See also: `semver.compare`. →index →topic

**4.569 `semver.prerelease` : procedure/1**

Usage: (`semver.prerelease` *s*)=> *str*

Return the prerelease part of a version string, or the empty string if there is none. For example, (`semver.prerelease` “v2.1.0-pre+build”) returns “-pre”.

See also: `semver.build`, `semver.major`, `semver.major-minor`. →index →topic

**4.570 `seq?` : procedure/1**

Usage: (`seq?` *seq*)=> *bool*

Return true if *seq* is a sequence, nil otherwise.

See also: `list`, `array`, `string`, `slice`, `nth`. →index →topic

**4.571 set : procedure/3**

Usage: (`set` `d` `key` `value`)

Set `value` for `key` in dict `d`.

See also: `dict`, `get`, `get-or-set`. →index →topic

**4.572 set\* : procedure/2**

Usage: (`set*` `d` `li`)

Set in dict `d` the keys and values in list `li`. The list `li` must be of the form (key-1 value-1 key-2 value-2 ... key-n value-n). This function may be slightly faster than using individual `set` operations.

See also: `dict`, `set`. →index →topic

**4.573 set->list : procedure/1**

Usage: (`set->list` `s`)=> `li`

Convert set `s` to a list of set elements.

See also: `list->set`, `make-set`, `set-element?`, `set-union`, `set-intersection`, `set-complement`, `set-difference`, `set?`, `set-empty`. →index →topic

**4.574 set-clipboard-content : procedure/1**

Usage: (`set-clipboard-content` `str`)

Set the operating system clipboard content to string `str`. This function might raise an error if clipboard access is prohibited by host security settings.

See also: `get-clipboard-content`. →index →topic

**4.575 set-color : procedure/1**

Usage: (`set-color` `sel` `colorlist`)

Set the color according to `sel` to the color `colorlist` of the form '(r g b a). See `color` for information about `sel`.

See also: `color`, `reset-color`, `the-color`, `with-colors`. →index →topic

**4.576 set-complement : procedure/2**

Usage: (set-complement a domain)=> set

Return all elements in `domain` that are not elements of `a`.

See also: [list->set](#), [set->list](#), [make-set](#), [set-element?](#), [set-union](#), [set-difference](#), [set-intersection](#), [set?](#), [set-empty?](#), [set-subset?](#), [set-equal?](#). [→index](#) [→topic](#)

**4.577 set-difference : procedure/2**

Usage: (set-difference a b)=> set

Return the set-theoretic difference of set `a` minus set `b`, i.e., all elements in `a` that are not in `b`.

See also: [list->set](#), [set->list](#), [make-set](#), [set-element?](#), [set-union](#), [set-intersection](#), [set-complement](#), [set?](#), [set-empty?](#), [set-subset?](#), [set-equal?](#). [→index](#) [→topic](#)

**4.578 set-element? : procedure/2**

Usage: (set-element? s elem)=> bool

Return true if set `s` has element `elem`, nil otherwise.

See also: [make-set](#), [list->set](#), [set->list](#), [set-union](#), [set-intersection](#), [set-complement](#), [set-difference](#), [set?](#), [set-empty?](#). [→index](#) [→topic](#)

**4.579 set-empty? : procedure/1**

Usage: (set-empty? s)=> bool

Return true if set `s` is empty, nil otherwise.

See also: [make-set](#), [list->set](#), [set->list](#), [set-union](#), [set-intersection](#), [set-complement](#), [set-difference](#), [set?](#). [→index](#) [→topic](#)

**4.580 set-entry-cursor-column : procedure/2**

Usage: (set-entry-cursor-column entry column)

Set the column position of the cursor in `entry` to integer `column`.

See also: [get-entry-cursor-pos](#), [set-entry-cursor-row](#). [→index](#) [→topic](#)

#### 4.581 `set-entry-cursor-row`: procedure/2

Usage: (`set-entry-cursor-row` *entry* *row*)

Set the row position of the cursor in *entry* to integer *row*.

See also: `get-entry-cursor-pos`, `set-entry-cursor-column`. [→index](#) [→topic](#)

#### 4.582 `set-entry-min-rows-visible`: procedure/2

Usage: (`set-entry-min-rows-visible` *entry* *rows*)

Set the minimum number of rows of *entry* that are visible. This ensures that *rows* text rows are visible and is a way of setting the entry's minimum size. Curiously, there is no corresponding `set-entry-min-columns-visible` function yet.

See also: `new-entry`. [→index](#) [→topic](#)

#### 4.583 `set-entry-on-change-callback`: procedure/2

Usage: (`set-entry-on-change-callback` *entry* *proc*)

Set the callback of *entry* that is triggered when the entry text changes. *proc* must be a procedure that takes the entry text as string.

See also: `new-entry`, `set-entry-cursor-change-callback`. [→index](#) [→topic](#)

#### 4.584 `set-entry-on-cursor-change-callback`: procedure/2

Usage: (`set-entry-cursor-change-callback` *entry* *proc*)

Set the cursor change callback of *entry* to *proc*, which is a procedure that takes the entry ID as argument.

See also: `new-entry`, `set-entry-on-change-callback`. [→index](#) [→topic](#)

#### 4.585 `set-entry-place-holder`: procedure/2

Usage: (`set-entry-place-holder` *entry* *str*)

Set the place holder string of *entry* to *str*. This is displayed as a prompt when no text is entered.

See also: `new-entry`, `set-entry-text`. [→index](#) [→topic](#)

**4.586 set-entry-text : procedure/2**

Usage: (set-entry-text entry str)

Set the text of `entry` to string `str`.

See also: `new-entry`, `set-entry-place-holder`. →index →topic

**4.587 set-entry-validator : procedure/2**

Usage: (set-entry-validator entry validator)

Set the `validator` of `entry`. A validator must be created first from a special procedure or a regular expression.

See also: `new-entry`, `new-validator`, `new-combined-string-validator`, `new-time-validator`, `new-regexp-validator`, `validate-object`. →index →topic

**4.588 set-equal? : procedure/2**

Usage: (set-equal? a b) => bool

Return true if `a` and `b` contain the same elements.

See also: `set-subset?`, `list->set`, `set-element?`, `set->list`, `set-union`, `set-difference`, `set-intersection`, `set-complement`, `set?`, `set-empty?`. →index →topic

**4.589 set-help-topic-info : procedure/3**

Usage: (set-help-topic-info topic header info)

Set a human-readable information entry for help `topic` with human-readable `header` and `info` strings.

See also: `defhelp`, `help-topic-info`. →index →topic

**4.590 set-intersection : procedure/2**

Usage: (set-intersection a b) => set

Return the intersection of sets `a` and `b`, i.e., the set of elements that are both in `a` and in `b`.

See also: `list->set`, `set->list`, `make-set`, `set-element?`, `set-union`, `set-complement`, `set-difference`, `set?`, `set-empty?`, `set-subset?`, `set-equal?`. →index →topic

#### 4.591 **set-label-text**: procedure/2

Usage: (**set-label-text** **label** **str**)

Sets the text of **label** to string **str**. This might resize the label depending on the layout in which the label is put.

See also: **new-label**. [→index](#) [→topic](#)

#### 4.592 **set-menu-item-checked**: procedure/2

Usage: (**set-menu-item-checked** **item** **checked?**)

Set the menu item check mark display if **checked?** is non-nil, remove it otherwise.

See also: **menu-item-checked?**, **set-menu-item-disabled**, **menu-item-disabled?**, **get-menu-item-label**, **set-menu-item-label**, **new-menu\***, **new-menu**, **new-menu-item**, **new-menu-item-separator**. [→index](#) [→topic](#)

#### 4.593 **set-menu-item-disabled**: procedure/2

Usage: (**set-menu-item-disabled** **item** **disabled?**)

Disable the menu item if **disabled?** is non-nil, enable it otherwise.

See also: **menu-item-disabled?**, **set-menu-item-checked**, **menu-item-checked?**, **get-menu-item-label**, **set-menu-item-label**, **new-menu\***, **new-menu**, **new-menu-item**, **new-menu-item-separator**. [→index](#) [→topic](#)

#### 4.594 **set-menu-item-label**: procedure/2

Usage: (**set-menu-item-label** **item** **str**)

Set the label of menu **item** to **str**.

See also: **get-menu-item-label**, **set-menu-item-disabled**, **menu-item-disabled?**, **set-menu-item-checked**, **menu-item-checked?**, **new-menu\***, **new-menu**, **new-menu-item**, **new-menu-item-separator**. [→index](#) [→topic](#)



#### 4.595 **set-object-on-validation-change-callback : procedure/2**

Usage: ([set-object-on-validation-change-callback](#) *obj* *proc*)

Set a validatable object's *obj* validation change callback, which is called when the object's validation changes. The callback *proc* takes a string or nil as argument. When it is nil, the validation was successful. When it is a string, then the validation failed with the string as reason. This can be used to track validation changes of any validatable object (such as a text entry) to e.g. display custom messages or icons when the validation fails or succeeds.

See also: [validate-object](#), [new-validator](#), [set-entry-validator](#). [→index](#) [→topic](#)

#### 4.596 **set-permissions : nil**

Usage: ([set-permissions](#) *li*)

Set the permissions for the current interpreter. This will trigger an error when the permission cannot be set due to a security violation. Generally, permissions can only be downgraded (made more stringent) and never relaxed. See the information for [permissions](#) for an overview of symbolic flags.

See also: [permissions](#), [permission?](#), [when-permission](#), [sys](#). [→index](#) [→topic](#)

#### 4.597 **set-progress-bar : procedure/1 or more**

Usage: ([set-progress-bar](#) *bar* *value* [*selector* *value*])

Set the value of progress-bar *bar* as follows. If a single number is provided, then the current value of the progress-bar is set to this number. If a selector symbol is provided, then if it is 'value, the progress-bar value is set to the following number, if it is 'max or 'min, then the progress-bar maximum or minimum values are set to the respective following number. If it is 'formatter, then the following value must be a procedure that takes the progress-bar ID as argument and returns a string that represents the display of the progress-bar at the given time.

See also: [get-progress-bar-value](#), [new-progress-bar](#), [new-infinite-progress-bar](#). [→index](#) [→topic](#)

#### 4.598 **set-slider-value : procedure/2**

Usage: ([set-slider-value](#) *slider* *fl*)

Set the value of *slider* to float *fl*.

See also: [new-slider](#). [→index](#) [→topic](#)

**4.599 set-subset? : procedure/2**

Usage: `(set-subset? a b) => bool`

Return true if `a` is a subset of `b`, nil otherwise.

See also: `set-equal?`, `list->set`, `set->list`, `make-set`, `set-element?`, `set-union`, `set-difference`, `set-intersection`, `set-complement`, `set?`, `set-empty?`. [→index](#) [→topic](#)

**4.600 set-text-alignment : procedure/2**

Usage: `(set-text-alignment text sym)`

Set the alignment of `text` to `sym`, which must be one of '(leading center trailing).

See also: `new-text`, `set-text-size`, `set-text-style`. [→index](#) [→topic](#)

**4.601 set-text-grid-cell : procedure/4**

Usage: `(set-text-grid-cell grid row column li)`

Set the text `grid` cell at `row` and `column` (both 0-indexed) to the list `li`, where `li` must consist of a unicode string containing one rune and a valid grid style list.

See also: `get-text-grid-cell`, `set-text-grid-rune`, `get-text-grid-row`, `set-text-grid-row`. [→index](#) [→topic](#)

**4.602 set-text-grid-row : procedure/3**

Usage: `(set-text-grid-row grid row row-spec)`

Set the `row` of `grid` to the given `row-spec`, which is a list containing an array of grid cells like in the return value of `get-text-grid-row` and a grid style for the row as a whole.

See also: `get-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-cell`, `set-text-grid-rune`, `set-text-grid-style-range`. [→index](#) [→topic](#)

**4.603 set-text-grid-row-style : procedure/3**

Usage: `(set-text-grid-row-style grid row style)`

Set the style of text `grid` at `row` to the given grid `style`.

See also: `set-text-grid-row`, `set-text-grid-cell`, `get-text-grid-row`, `set-text-grid-rune`, `set-text-grid-style-range`. [→index](#) [→topic](#)

#### 4.604 `set-text-grid-rune`: procedure/4

Usage: (`set-text-grid-rune` `grid` `row` `column` `str`)

Set the rune of `grid` at `row` and `column` to the unicode glyph in string `str`.

See also: `set-text-grid-style`, `set-text-grid-cell`, `get-text-grid-cell`. [→index](#) [→topic](#)

#### 4.605 `set-text-grid-show-line-numbers`: procedure/2

Usage: (`set-text-grid-show-line-numbers` `grid` `show?`)

Set whether `grid` shows line numbers. If `show?` is not nil, then line numbers are shown, otherwise they are not shown.

See also: `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. [→index](#) [→topic](#)

#### 4.606 `set-text-grid-show-whitespace`: procedure/2

Usage: (`set-text-grid-show-whitespace` `grid` `show?`)

Set whether `grid` shows whitespace characters. If `show?` is not nil, then whitespace characters are shown, otherwise they are not shown.

See also: `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. [→index](#) [→topic](#)

#### 4.607 `set-text-grid-style`: procedure/4

Usage: (`set-text-grid-style` `grid` `row` `column` `style`)

Set the grid style of `grid` at `row` and `column` to the list `style`.

See also: `set-text-grid-cell`, `set-text-grid-rune`, `set-text-grid-style-range`. →index  
→topic

#### 4.608 `set-text-grid-style-range`: procedure/6

Usage: (`set-text-grid-style-range` `grid` `start-row` `start-column` `end-row` `end-column` `style`)

Set the grid style of `grid` starting at `start-row` and `start-column` and ending at `end-row` and `end-column` (all inclusive) to the grid `style`.

See also: `set-text-grid-style`, `set-text-grid-cell`, `set-text-grid-row-style`. →index  
→topic

#### 4.609 `set-text-grid-tab-width`: procedure/2

Usage: (`set-text-grid-tab-width` `grid` `width`)

Set the tabulator width of `grid` to integer `width` space characters.

See also: `new-text-grid`, `text-grid-show-line-numbers?`, `text-grid-show-whitespace?`, `get-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index →topic

#### 4.610 `set-text-grid-text`: procedure/2

Usage: (`set-text-grid-text` `grid` `str`)

Set the text of the text `grid` to the given `str`.

See also: `get-text-grid-text`, `new-text-grid`, `set-text-grid-rune`, `set-text-grid-row`. →index →topic

#### 4.611 `set-text-size`: procedure/2

Usage: (`set-text-size` *text* *size*)

Set the size of *text* to float *size*.

See also: `new-text`, `set-text-alignment`, `set-text-style`. →index →topic

#### 4.612 `set-text-style`: procedure/2

Usage: (`set-text-style` *text* *li*

Set the style of *text* to the specification in list *li*, which must contain symbols in '(bold italic monospace symbol tab-width). If a symbol in the list is `tab-width`, it must be followed by an integer. `bold` sets boldface, `italic` makes the style italic, `monospace` selects the monospace/typewriter font, and `symbol` selects the `symbol` font. `tab-width` followed by an integer sets the width of tabulator in terms of the number of space characters.

See also: `new-text`, `set-text-alignment`, `set-text-size`. →index →topic

#### 4.613 `set-union`: procedure/2

Usage: (`set-union` *a* *b*)=> *set*

Return the union of sets *a* and *b* containing all elements that are in *a* or in *b* (or both).

See also: `list->set`, `set->list`, `make-set`, `set-element?`, `set-intersection`, `set-complement`, `set-difference`, `set?`, `set-empty?`. →index →topic

#### 4.614 `set-volume`: procedure/1

Usage: (`set-volume` *fl*)

Set the master volume for all sound to *fl*, a value between 0.0 and 1.0.

See also: `beep`. →index →topic

#### 4.615 `set-window-content`: procedure/2

Usage: (`set-window-content` *window* *canvas-object*)

Set the main content of the window, which must be an existing canvas object such as a widget or container with layout.

See also: [get-window-content](#), [get-window-canvas](#), [new-window](#). [→index](#) [→topic](#)

#### 4.616 **set-window-full-screen** : procedure/2

Usage: ([set-window-full-screen](#) [window](#) [full-screen?](#))

If [full-screen?](#) is not nil, then [window](#) is set to full screen mode, otherwise the window is set to normal mode. In full screen mode the window is centered and fills the whole screen of the main monitor (multiple monitors are currently not supported).

See also: [window-full-screen?](#), [center-window-on-screen](#). [→index](#) [→topic](#)

#### 4.617 **set-window-icon** : procedure/1

Usage: ([set-window-icon](#) [window](#) [resource](#))

Set the icon of [window](#) to the given icon [resource](#). Setting the icon does not guarantee that it is displayed, since this is platform-dependent.

See also: [get-window-icon](#), [new-icon](#), [theme-icon](#). [→index](#) [→topic](#)

#### 4.618 **set-window-main-menu** : procedure/2

Usage: ([set-window-main-menu](#) [window](#) [main-menu](#))

Set the main menu of [window](#) to [main-menu](#).

See also: [new-main-menu](#), [new-menu](#), [new-menu\\*](#). [→index](#) [→topic](#)

#### 4.619 **set-window-on-close-callback** : procedure/2

Usage: ([set-window-on-close-callback](#) [window](#) [proc](#))

Sets the close callback of [window](#). [proc](#) must be a function that takes no arguments and is evaluated when the window is closed.

See also: [show-window](#), [close-window](#), [hide-window](#). [→index](#) [→topic](#)

#### 4.620 `set-window-padded` : procedure/2

Usage: (`set-window-padded` `window` `padded?`)

If `padded?` is not nil, then `window` is set to being padded. This is the default of new window. If `padded?` is nil, then the window's padding is removed, which means that the whole content area of the window can be filled with user interface elements and draw commands. This would be used for a game display, for instance.

See also: `window-padded?`, `set-window-full-screen`, `window-full-screen?`, `center-window-on-screen`. →index →topic

#### 4.621 `set-window-size` : procedure/2

Usage: (`set-window-size` `window` `width` `height`)

Set the size of `window` to `width` and `height` as floats. Sizes and positions are generally given as floats whose accuracy is only guaranteed to 32 bit.

See also: `new-window`, `show-window`, `hide-window`. →index →topic

#### 4.622 `set-window-title` : procedure/2

Usage: (`set-window-title` `window` `title`)

Set the title of `window` to string `title`.

See also: `get-window-title`. →index →topic

#### 4.623 `set?` : procedure/1

Usage: (`set?` `x`)=> `bool`

Return true if `x` can be used as a set, nil otherwise.

See also: `list->set`, `make-set`, `set->list`, `set-element?`, `set-union`, `set-intersection`, `set-complement`, `set-difference`, `set-empty?`. →index →topic

#### 4.624 `setcar` : procedure/1

Usage: (`setcar` `li` `elem`)=> `li`

Mutate `li` such that its car is `elem`. Same as `rplaca`.

See also: `rplaca`, `rplacd`, `setcdr`. →index →topic

#### 4.625 `setcdr` : procedure/1

Usage: `(setcdr li1 li2)=> li`

Mutate `li1` such that its `cdr` is `li2`. Same as `rplacd`.

See also: `rplacd`, `rplaca`, `setcar`. →index →topic

#### 4.626 `setprop` : procedure/3

Usage: `(setprop obj slot value)`

Set property `slot` in `obj` to `value`. An error occurs if the object does not have a property with that name.

See also: `new`, `isa?`, `prop`, `object?`, `class`-name, `supers`, `props`, `methods`, `has-prop?`. →index →topic

#### 4.627 `shorten` : procedure/2

Usage: `(shorten s n)=> str`

Shorten string `s` to length `n` in a smart way if possible, leave it untouched if the length of `s` is smaller than `n`.

See also: `substr`. →index →topic

#### 4.628 `show-object` : procedure/1

Usage: `(show-object obj)`

Show the canvas object `obj`.

See also: `disable-object`, `enable-object`, `hide-object`, `object-disabled?`, `move-object`, `resize-object`, `get-object-size`, `get-object-min-size`, `get-object-position`, `object-visible?`, `refresh-object`, `new-entry`, `new-label`. →index →topic



**4.629 `sleep` : procedure/1**

Usage: (`sleep` `ms`)

Halt the current task execution for `ms` milliseconds.

See also: `sleep-ns`, `time`, `now`, `now-ns`. →index →topic

**4.630 `sleep-ns` : procedure/1**

Usage: (`sleep-ns` `n`)

Halt the current task execution for `n` nanoseconds.

See also: `sleep`, `time`, `now`, `now-ns`. →index →topic

**4.631 `slice` : procedure/3**

Usage: (`slice` `seq` `low` `high`)=> `seq`

Return the subsequence of `seq` starting from `low` inclusive and ending at `high` exclusive. Sequences are 0-indexed.

See also: `list`, `array`, `string`, `nth`, `seq?`. →index →topic

**4.632 `sort` : procedure/2**

Usage: (`sort` `li` `proc`)=> `li`

Sort the list `li` by the given less-than procedure `proc`, which takes two arguments and returns true if the first one is less than the second, nil otherwise.

See also: `array-sort`. →index →topic

**4.633 `sort-symbols` : nil**

Usage: (`sort-symbols` `li`)=> `list`

Sort the list of symbols `li` alphabetically.

See also: `out`, `dp`, `du`, `dump`. →index →topic

**4.634 spaces : procedure/1**

Usage: (`spaces` `n`)=> `str`

Create a string consisting of `n` spaces.

See also: `strbuild`, `strleft`, `strright`. →index →topic

**4.635 stack-empty? : procedure/1**

Usage: (`queue-empty?` `s`)=> `bool`

Return true if the stack `s` is empty, nil otherwise.

See also: `make-stack`, `stack?`, `push!`, `pop!`, `stack-len`, `glance`. →index →topic

**4.636 stack-len : procedure/1**

Usage: (`stack-len` `s`)=> `int`

Return the length of the stack `s`.

See also: `make-queue`, `queue?`, `enqueue!`, `dequeue!`, `glance`, `queue-len`. →index

**Warning:** Be advised that this is of limited use in some concurrent contexts, since the length of the queue might have changed already once you've obtained it! →topic

**4.637 stack? : procedure/1**

Usage: (`stack?` `q`)=> `bool`

Return true if `q` is a stack, nil otherwise.

See also: `make-stack`, `push!`, `pop!`, `stack-empty?`, `stack-len`, `glance`. →index →topic

**4.638 str+ : procedure/0 or more**

Usage: (`str+` [`s`] ...)=> `str`

Append all strings given to the function.

See also: `str?`. →index →topic

**4.639 str->array : procedure/1**

Usage: (str->array s)=> array

Return the string *s* as an array of unicode glyph integer values.

See also: array->str. →index →topic

**4.640 str->blob : procedure/1**

Usage: (str->blob s)=> blob

Convert string *s* into a blob.

See also: blob->str. →index →topic

**4.641 str->char : procedure/1**

Usage: (str->char s)

Return the first character of *s* as unicode integer.

See also: char->str. →index →topic

**4.642 str->chars : procedure/1**

Usage: (str->chars s)=> array

Convert the UTF-8 string *s* into an array of UTF-8 rune integers. An error may occur if the string is not a valid UTF-8 string.

See also: runes->str, str->char, char->str. →index →topic

**4.643 str->expr : procedure/0 or more**

Usage: (str->expr s [default])=> any

Convert a string *s* into a Lisp expression. If **default** is provided, it is returned if an error occurs, otherwise an error is raised.

See also: expr->str, str->expr\*, openstr, externalize, internalize. →index →topic

**4.644 str->expr\* : procedure/0 or more**

Usage: (str->expr\* s [default])=> li

Convert a string *s* into a list consisting of the Lisp expressions in *s*. If **default** is provided, then this value is put in the result list whenever an error occurs. Otherwise an error is raised. Notice that it might not always be obvious what expression in *s* triggers an error, since this hinges on the way the internal expression parser works.

See also: [str->expr](#), [expr->str](#), [openstr](#), [internalize](#), [externalize](#). [→index](#) [→topic](#)

**4.645 str->list : procedure/1**

Usage: (str->list s)=> list

Return the sequence of numeric chars that make up string *s*.

See also: [str->array](#), [list->str](#), [array->str](#), [chars](#). [→index](#) [→topic](#)

**4.646 str->sym : procedure/1**

Usage: (str->sym s)=> sym

Convert a string into a symbol.

See also: [sym->str](#), [intern](#), [make-symbol](#). [→index](#) [→topic](#)

**4.647 str-count-substr : procedure/2**

Usage: (str-count-substr s1 s2)=> int

Count the number of non-overlapping occurrences of substring *s2* in string *s1*.

See also: [str-replace](#), [str-replace\\*](#), [instr](#). [→index](#) [→topic](#)

**4.648 str-empty? : procedure/1**

Usage: (str-empty? s)=> bool

Return true if the string *s* is empty, nil otherwise.

See also: [strlen](#). [→index](#) [→topic](#)

**4.649 str-exists? : procedure/2**

Usage: (str-exists? s pred)=> bool

Return true if `pred` returns true for at least one character in string `s`, nil otherwise.

See also: `exists?`, `forall?`, `list-exists?`, `array-exists?`, `seq?`. →index →topic

**4.650 str-forall? : procedure/2**

Usage: (str-forall? s pred)=> bool

Return true if predicate `pred` returns true for all characters in string `s`, nil otherwise.

See also: `foreach`, `map`, `forall?`, `array-forall?`, `list-forall`, `exists?`. →index →topic

**4.651 str-foreach : procedure/2**

Usage: (str-foreach s proc)

Apply `proc` to each element of string `s` in order, for the side effects.

See also: `foreach`, `list-foreach`, `array-foreach`, `map`. →index →topic

**4.652 str-index : procedure/2 or more**

Usage: (str-index s chars [pos])=> int

Find the first char in `s` that is in the charset `chars`, starting from the optional `pos` in `s`, and return its index in the string. If no matching char is found, nil is returned.

See also: `strsplit`, `chars`, `inchars`. →index →topic

**4.653 str-join : procedure/2**

Usage: (str-join li del)=> str

Join a list of strings `li` where each of the strings is separated by string `del`, and return the result string.

See also: `strlen`, `strsplit`, `str-slice`. →index →topic

**4.654 str-port? : procedure/1**

Usage: (`str-port?` *p*)=> *bool*

Return true if *p* is a string port, nil otherwise.

See also: `port?`, `file-port?`, `stropen`, `open`. →index →topic

**4.655 str-ref : procedure/2**

Usage: (`str-ref` *s* *n*)=> *n*

Return the unicode char as integer at position *n* in *s*. Strings are 0-indexed.

See also: `nth`. →index →topic

**4.656 str-remove-number : procedure/1**

Usage: (`str-remove-number` *s* [*del*])=> *str*

Remove the suffix number in *s*, provided there is one and it is separated from the rest of the string by *del*, where the default is a space character. For instance, “Test 29” will be converted to “Test”, “User-Name1-23-99” with delimiter “-” will be converted to “User-Name1-23”. This function will remove intermediate delimiters in the middle of the string, since it disassembles and reassembles the string, so be aware that this is not preserving inputs in that respect.

See also: `strsplit`. →index →topic

**4.657 str-remove-prefix : procedure/1**

Usage: (`str-remove-prefix` *s* *prefix*)=> *str*

Remove the prefix *prefix* from string *s*, return the string without the prefix. If the prefix does not match, *s* is returned. If *prefix* is longer than *s* and matches, the empty string is returned.

See also: `str-remove-suffix`. →index →topic

**4.658 str-remove-suffix : procedure/1**

Usage: (`str-remove-suffix` *s* *suffix*)=> *str*

remove the suffix *suffix* from string *s*, return the string without the suffix. If the suffix does not match, *s* is returned. If *suffix* is longer than *s* and matches, the empty string is returned.

See also: `str-remove-prefix`. [→index](#) [→topic](#)

#### 4.659 `str-replace` : procedure/4

Usage: `(str-replace s t1 t2 n)=> str`

Replace the first `n` instances of substring `t1` in `s` by `t2`.

See also: `str-replace*`, `str-count-substr`. [→index](#) [→topic](#)

#### 4.660 `str-replace*` : procedure/3

Usage: `(str-replace* s t1 t2)=> str`

Replace all non-overlapping substrings `t1` in `s` by `t2`.

See also: `str-replace`, `str-count-substr`. [→index](#) [→topic](#)

#### 4.661 `str-reverse` : procedure/1

Usage: `(str-reverse s)=> str`

Reverse string `s`.

See also: `reverse`, `array-reverse`, `list-reverse`. [→index](#) [→topic](#)

#### 4.662 `str-segment` : procedure/3

Usage: `(str-segment str start end)=> list`

Parse a string `str` into words that start with one of the characters in string `start` and end in one of the characters in string `end` and return a list consisting of lists of the form `(bool s)` where `bool` is true if the string starts with a character in `start`, nil otherwise, and `s` is the extracted string including start and end characters.

See also: `str+`, `strsplit`, `fmt`, `strbuild`. [→index](#) [→topic](#)

#### 4.663 `str-slice` : procedure/3

Usage: `(str-slice s low high)=> s`

Return a slice of string `s` starting at character with index `low` (inclusive) and ending at character with index `high` (exclusive).

See also: `slice`. →index →topic

#### 4.664 `str?` : procedure/1

Usage: `(str? s) => bool`

Return true if `s` is a string, nil otherwise.

See also: `num?`, `atom?`, `sym?`, `closure?`, `intrinsic?`, `macro?`. →index

#### 4.665 `strbuild` : procedure/2

Usage: `(strbuild s n) => str`

Build a string by repeating string `s` `n` times.

See also: `str+`. →index →topic

#### 4.666 `strcase` : procedure/2

Usage: `(strcase s sel) => str`

Change the case of the string `s` according to selector `sel` and return a copy. Valid values for `sel` are 'lower for conversion to lower-case, 'upper for uppercase, 'title for title case and 'utf-8 for utf-8 normalization (which replaces unprintable characters with "?").

See also: `strmap`. →index →topic

#### 4.667 `strcenter` : procedure/2

Usage: `(strcenter s n) => str`

Center string `s` by wrapping space characters around it, such that the total length the result string is `n`.

See also: `strleft`, `strright`, `strlimit`. →index →topic



**4.668 strcnt : procedure/2**

Usage: (`strcnt s del`)=> `int`

Return the number of non-overlapping substrings `del` in `s`.

See also: `strsplit`, `str-index`. →index →topic

**4.669 strleft : procedure/2**

Usage: (`strleft s n`)=> `str`

Align string `s` left by adding space characters to the right of it, such that the total length the result string is `n`.

See also: `strcenter`, `strright`, `strlimit`. →index →topic

**4.670 strlen : procedure/1**

Usage: (`strlen s`)=> `int`

Return the length of `s`.

See also: `len`, `seq?`, `str?`. →index →topic

**4.671 strless : procedure/2**

Usage: (`strless s1 s2`)=> `bool`

Return true if string `s1` < `s2` in lexicographic comparison, nil otherwise.

See also: `sort`, `array-sort`, `strcase`. →index →topic

**4.672 strlimit : procedure/2**

Usage: (`strlimit s n`)=> `str`

Return a string based on `s` cropped to a maximal length of `n` (or less if `s` is shorter).

See also: `strcenter`, `strleft`, `strright`. →index →topic

**4.673 strmap : procedure/2**

Usage: (`strmap s proc`)=> `str`

Map function `proc`, which takes a number and returns a number, over all unicode characters in `s` and return the result as new string.

See also: `map`. →index →topic

**4.674 stropen : procedure/1**

Usage: (`stropen s`)=> `streamport`

Open the string `s` as input stream.

See also: `open`, `close`. →index →topic

**4.675 strright : procedure/2**

Usage: (`strright s n`)=> `str`

Align string `s` right by adding space characters in front of it, such that the total length the result string is `n`.

See also: `strcenter`, `strleft`, `strlimit`. →index →topic

**4.676 strsplit : procedure/2**

Usage: (`strsplit s del`)=> `array`

Return an array of strings obtained from `s` by splitting `s` at each occurrence of string `del`.

See also: `str?`. →index →topic

**4.677 struct-index : procedure/1**

Usage: (`struct-index s`)=> `dict`

Returns the index of struct `s` as a dict. This dict is an internal representation of the struct's instance data.

See also: `defstruct`. →index →topic

**4.678 struct-instantiate : procedure/2**

Usage: (`struct-instantiate` *s* *li*)=> *record*

Instantiates the struct *s* with property a-list *li* as values for its properties and return the record. If a property is not in *li*, its value is set to nil.

See also: `make`, `defstruct`, `struct?`, `record?`. [→index](#) [→topic](#)

**4.679 struct-name : procedure/1**

Usage: (`struct-name` *s*)=> *sym*

Returns the name of a struct *s*. This is rarely needed since the struct is bound to a symbol with the same name.

See also: `defstruct`. [→index](#) [→topic](#)

**4.680 struct-props : procedure/1**

Usage: (`struct-props` *s*)=> *dict*

Returns the properties of structure *s* as dict.

See also: `defstruct`. [→index](#) [→topic](#)

**4.681 struct-size : procedure/1**

Usage: (`struct-size` *s*)=> *int*

Returns the number of properties of struct *s*.

See also: `defstruct`. [→index](#) [→topic](#)

**4.682 struct? : procedure/1**

Usage: (`struct?` *datum*)=> *boo*

Returns true if *datum* is a struct, nil otherwise.

See also: `defstruct`. [→index](#) [→topic](#)

**4.683 sub1 : procedure/1**

Usage: (`sub1` `n`)=> `num`

Subtract 1 from `n`.

See also: `add1`, `+`, `-`. →index →topic

**4.684 supers : procedure/1**

Usage: (`supers` `c`)=> `li`

Return the list of superclasses of class `c`. An error occurs if `c` is not a valid class.

See also: `class?`, `isa?`, `class-name`. →index →topic

**4.685 sym->str : procedure/1**

Usage: (`sym->str` `sym`)=> `str`

Convert a symbol into a string.

See also: `str->sym`, `intern`, `make-symbol`. →index →topic

**4.686 sym? : procedure/1**

Usage: (`sym?` `sym`)=> `bool`

Return true if `sym` is a symbol, nil otherwise.

See also: `str?`, `atom?`. →index →topic

**4.687 synout : procedure/1**

Usage: (`synout` `arg`)

Like out, but enforcing a new input line afterwards. This needs to be used when outputting concurrently in a future or task.

See also: `out`, `outy`, `synouty`. →index

**Warning:** Concurrent display output can lead to unexpected visual results and ought to be avoided. →topic

#### 4.688 **synouty** : procedure/1

Usage: (`synouty` `li`)

Like `outy`, but enforcing a new input line afterwards. This needs to be used when outputting concurrently in a future or task.

See also: `synout`, `out`, `outy`. →index

**Warning: Concurrent display output can lead to unexpected visual results and ought to be avoided.**

#### 4.689 **sys-key?** : procedure/1

Usage: (`sys-key?` `key`)=> `bool`

Return true if the given sys key `key` exists, nil otherwise.

See also: `sys`, `setsys`. →index →topic

#### 4.690 **sysmsg** : procedure/1

Usage: (`sysmsg` `msg`)

Asynchronously display a system message string `msg` if in console or page mode, otherwise the message is logged.

See also: `sysmsg*`, `synout`, `synouty`, `out`, `outy`. →index →topic

#### 4.691 **sysmsg\*** : procedure/1

Usage: (`sysmsg*` `msg`)

Display a system message string `msg` if in console or page mode, otherwise the message is logged.

See also: `sysmsg`, `synout`, `synouty`, `out`, `outy`. →index →topic

#### 4.692 **take** : procedure/3

Usage: (`take` `seq` `n`)=> `seq`

Return the sequence consisting of the `n` first elements of `seq`.

See also: `list`, `array`, `string`, `nth`, `seq?`. →index →topic

### 4.693 `task` : procedure/1

Usage: (`task sel proc`)=> `int`

Create a new task for concurrently running `proc`, a procedure that takes its own ID as argument. The `sel` argument must be a symbol in '(auto manual remove). If `sel` is 'remove, then the task is always removed from the task table after it has finished, even if an error has occurred. If `sel` is 'auto, then the task is removed from the task table if it ends without producing an error. If `sel` is 'manual then the task is not removed from the task table, its state is either 'canceled, 'finished, or 'error, and it must be removed manually with `task-remove` or `prune-task-table`. Broadcast messages are never removed. Tasks are more heavy-weight than futures and allow for message-passing.

See also: `task?`, `task-run`, `task-state`, `task-broadcast`, `task-send`, `task-recv`, `task-remove`, `prune-task-table`. →index →topic

### 4.694 `task-broadcast` : procedure/2

Usage: (`task-broadcast id msg`)

Send a message from task `id` to the blackboard. Tasks automatically send the message 'finished to the blackboard when they are finished.

See also: `task`, `task?`, `task-run`, `task-state`, `task-send`, `task-recv`. →index →topic

### 4.695 `task-recv` : procedure/1

Usage: (`task-recv id`)=> `any`

Receive a message for task `id`, or nil if there is no message. This is typically used by the task with `id` itself to periodically check for new messages while doing other work. By convention, if a task receives the message 'end it ought to terminate at the next convenient occasion, whereas upon receiving 'cancel it ought to terminate in an expedited manner.

See also: `task-send`, `task`, `task?`, `task-run`, `task-state`, `task-broadcast`. →index

**Warning: Busy polling for new messages in a tight loop is inefficient and ought to be avoided.**  
→topic

### 4.696 `task-remove` : procedure/1

Usage: (`task-remove id`)

Remove task `id` from the task table. The task can no longer be interacted with.

See also: [task](#), [task?](#), [task-state](#). [→index](#) [→topic](#)

#### 4.697 **task-run : procedure/1**

Usage: ([task-run](#) `id`)

Run task `id`, which must have been previously created with `task`. Attempting to run a task that is already running results in an error unless `silent?` is true. If `silent?` is true, the function does never produce an error.

See also: [task](#), [task?](#), [task-state](#), [task-send](#), [task-recv](#), [task-broadcast-](#). [→index](#) [→topic](#)

#### 4.698 **task-schedule : procedure/1**

Usage: ([task-schedule](#) `sel` `id`)

Schedule task `id` for running, starting it as soon as other tasks have finished. The scheduler attempts to avoid running more than (`cpunum`) tasks at once.

See also: [task](#), [task-run](#). [→index](#) [→topic](#)

#### 4.699 **task-send : procedure/2**

Usage: ([task-send](#) `id` `msg`)

Send a message `msg` to task `id`. The task needs to cooperatively use `task-recv` to reply to the message. It is up to the receiving task what to do with the message once it has been received, or how often to check for new messages.

See also: [task-broadcast](#), [task-recv](#), [task](#), [task?](#), [task-run](#), [task-state](#). [→index](#) [→topic](#)

#### 4.700 **task-state : procedure/1**

Usage: ([task-state](#) `id`)=> `sym`

Return the state of the task, which is a symbol in '(finished error stopped new waiting running).

See also: [task](#), [task?](#), [task-run](#), [task-broadcast](#), [task-recv](#), [task-send](#). [→index](#) [→topic](#)

#### 4.701 **task? : procedure/1**

Usage: ([task? id](#))=> [bool](#)

Check whether the given [id](#) is for a valid task, return true if it is valid, nil otherwise.

See also: [task](#), [task-run](#), [task-state](#), [task-broadcast](#), [task-send](#), [task-recv](#). →index →topic

#### 4.702 **terpri : procedure/0**

Usage: ([terpri](#))

Advance the host OS terminal to the next line.

See also: [princ](#), [out](#), [outy](#). →index →topic

#### 4.703 **testing : macro/1**

Usage: ([testing name](#))

Registers the string [name](#) as the name of the tests that are next registered with expect.

See also: [expect](#), [expect-err](#), [expect-ok](#), [run-selftest](#). →index →topic

#### 4.704 **text-grid-show-line-numbers? : procedure/1**

Usage: ([text-grid-show-line-numbers? grid](#))=> [bool](#)

Return true if the text [grid](#) shows line numbers, nil otherwise.

See also: [new-text-grid](#), [text-grid-show-whitespace?](#), [get-text-grid-tab-width](#), [set-text-grid-tab-width](#), [set-text-grid-show-line-numbers](#), [set-text-grid-show-whitespace](#), [get-text-grid-row](#), [get-text-grid-row-text](#), [set-text-grid-cell](#), [get-text-grid-cell](#), [set-text-grid-row](#), [set-text-grid-row-style](#), [set-text-grid-rune](#), [set-text-grid-style](#), [set-text-grid-style-range](#), [set-text-grid-text](#), [get-text-grid-text](#). →index →topic

#### 4.705 **text-grid-show-whitespace? : procedure/1**

Usage: ([text-grid-show-whitespace? grid](#))=> [bool](#)

Return true if the text [grid](#) shows whitespace glyphs, nil otherwise.



See also: `new-text-grid`, `text-grid-show-line-numbers?`, `get-text-grid-tab-width`, `set-text-grid-tab-width`, `set-text-grid-show-line-numbers`, `set-text-grid-show-whitespace`, `get-text-grid-row`, `get-text-grid-row-text`, `set-text-grid-cell`, `get-text-grid-cell`, `set-text-grid-row`, `set-text-grid-row-style`, `set-text-grid-rune`, `set-text-grid-style`, `set-text-grid-style-range`, `set-text-grid-text`, `get-text-grid-text`. →index →topic

#### 4.706 `the-color`: procedure/1

Usage: (`the-color` *colors-spec*)=> (*r g b a*)

Return the color list (*r g b a*) based on a color specification, which may be a color list (*r g b*), a color selector for (*color selector*) or a color name such as 'dark-blue.

See also: `*colors*`, `color`, `set-color`, `outy`. →index →topic

#### 4.707 `the-color-names`: procedure/0

Usage: (`the-color-names`)=> *li*

Return the list of color names in *colors*.

See also: `*colors*`, `the-color`. →index →topic

#### 4.708 `theme-icon`: procedure/1

Usage: (`theme-icon` *selector*)=> *int*

Obtain a pre-defined icon from the application's theme based on the symbol *selector*, which may be one of '(cancel check-button check-button-checked color-achromatic color-chromatic color-palette computer confirm content-add content-clear content-copy content-cut content-paste content-redo content-remove content-undo delete document-create document-print document-download error file-application file-audio file-image file-text file-video file-folder-new folder-open folder-grid help history home info list login logout mail-attachment mail-compose mail-forward mail-reply-all mail-reply mail-send media-fast-forward media-fast-rewind media-music media-pause media-photo media-play media-record media-replay media-skip-next media-skip-previous media-stop media-video media-expand menu more-horizontal more-vertical move-down move-up navigate-back navigate-next question radio-button radio-button-checked search-replace search settings storage upload view-full-screen view-refresh view-restore visibility-off visibility volume-down volume-mute volume-up warning).

See also: `new-icon`, `new-image-from-`, `new-image-from-resource`. →index →topic

#### 4.709 **time: procedure/1**

Usage: (`time proc`)=> `int`

Return the time in nanoseconds that it takes to execute the procedure with no arguments `proc`.

See also: `now-ns`, `now`. →index →topic

#### 4.710 **trigger-menu-last: procedure/1**

Usage: (`trigger-menu-last menu`)

Find the last active menu or submenu item and trigger it.

See also: `activate-menu-last-submenu`, `activate-menu-next`, `activate-menu-previous`, `new-menu`. →index →topic

#### 4.711 **truncate: procedure/1 or more**

Usage: (`truncate x [y]`)=> `int`

Round down to nearest integer of `x`. If `y` is present, divide `x` by `y` and round down to the nearest integer.

See also: `div`, `/`, `int`. →index →topic

#### 4.712 **try: macro/2 or more**

Usage: (`try (finals ...) body ...`)

Evaluate the forms of the `body` and afterwards the forms in `finals`. If during the execution of `body` an error occurs, first all `finals` are executed and then the error is printed by the default error printer.

See also: `with-final`, `with-error-handler`. →index →topic

#### 4.713 **type-of: macro/1**

Usage: (`type-of datum`)=> `sym`

Returns the type of `datum` as symbol like `type-of*` but without having to quote the argument. If `datum` is an unbound symbol, then this macro returns 'unbound. Otherwise the type of a given symbol's value or the type of a given literal is returned.

See also: `type-of*`. →index →topic

#### 4.714 `type-of*` : procedure/1

Usage: `(type-of* datum)=> sym`

Return the type of `datum` as a symbol. This uses existing predicates and therefore is not faster than testing with predicates directly.

See also: `num?`, `str?`, `sym?`, `list?`, `array?`, `bool?`, `eof?`, `boxed?`, `intrinsic?`, `closure?`, `macro?`, `blob?`. [→index](#) [→topic](#)

#### 4.715 `unless` : macro/1 or more

Usage: `(unless cond expr ...)=> any`

Evaluate expressions `expr` if `cond` is not true, returns void otherwise.

See also: `if`, `when`, `cond`. [→index](#) [→topic](#)

#### 4.716 `unprotect` : procedure/0 or more

Usage: `(unprotect [sym] ...)`

Unprotect symbols `sym` ..., allowing mutation or rebinding them. The symbols need to be quoted. This operation requires the permission 'allow-unprotect to be set, or else an error is caused.

See also: `protect`, `protected?`, `dict-unprotect`, `dict-protected?`, `permissions`, `permission?`, `setq`, `bind`, `interpret`. [→index](#) [→topic](#)

#### 4.717 `unprotect-toplevel-symbols` : procedure/0

Usage: `(unprotect-toplevel-symbols)`

Attempts to unprotect all toplevel symbols.

See also: `protect-toplevel-symbols`, `protect`, `unprotect`, `declare-unprotected`. [→index](#) [→topic](#)

#### 4.718 `valid?` : procedure/1

Usage: `(valid? obj)=> bool`

Return true if `obj` is a valid object, nil otherwise. What exactly object validity means is undefined, but certain kind of objects such as graphics objects may be marked invalid when they can no longer

be used because they have been disposed off by a subsystem and cannot be automatically garbage collected. Generally, invalid objects ought no longer be used and need to be discarded.

See also: [blob?](#). [→index](#) [→topic](#)

#### 4.719 `validate-object`: procedure/2

Usage: (`validate-object` *obj*)=> *str*

Validate the validatable object *obj* programmatically and return the validation failure as string, or the empty string if validation succeeded. It sometimes makes sense to call this explicitly in order to force the object to display its validation state.

See also: [set-object-on-validation-change-callback](#), [new-validator](#), [set-entry-validator](#). [→index](#) [→topic](#)

#### 4.720 `void`: procedure/0 or more

Usage: (`void` [*any*] ...)

Always returns void, no matter what values are given to it. Void is a special value that is not printed in the console.

See also: [void?](#). [→index](#) [→topic](#)

#### 4.721 `void?`: procedure/1

Usage: (`void?` *datum*)

Return true if *datum* is the special symbol void, nil otherwise.

See also: [void](#). [→index](#) [→topic](#)

#### 4.722 `wait-for`: procedure/2

Usage: (`wait-for` *dict* *key*)

Block execution until the value for *key* in *dict* is not-nil. This function may wait indefinitely if no other thread sets the value for *key* to not-nil.

See also: [wait-for\\*](#), [future](#), [force](#), [wait-until](#), [wait-until\\*](#). [→index](#)

**Warning:** This cannot be used for synchronization of multiple tasks due to potential race-conditions. [→topic](#)

### 4.723 `wait-for*` : procedure/3

Usage: (`wait-for*` dict key timeout)

Blocks execution until the value for `key` in `dict` is not-nil or `timeout` nanoseconds have passed, and returns that value or nil if waiting timed out. If `timeout` is negative, then the function waits potentially indefinitely without any timeout. If a non-nil key is not found, the function sleeps at least *sync-wait-lower-bound* nanoseconds and up to *sync-wait-upper-bound* nanoseconds until it looks for the key again.

See also: `future`, `force`, `wait-for`, `wait-until`, `wait-until*`. →index

**Warning:** This cannot be used for synchronization of multiple tasks due to potential race-conditions. →topic

### 4.724 `wait-for-empty*` : procedure/3

Usage: (`wait-for-empty*` dict key timeout)

Blocks execution until the `key` is no longer present in `dict` or `timeout` nanoseconds have passed. If `timeout` is negative, then the function waits potentially indefinitely without any timeout.

See also: `future`, `force`, `wait-for`, `wait-until`, `wait-until*`. →index

**Warning:** This cannot be used for synchronization of multiple tasks due to potential race-conditions. →topic

### 4.725 `wait-until` : procedure/2

Usage: (`wait-until` dict key pred)

Blocks execution until the unary predicate `pred` returns true for the value at `key` in `dict`. This function may wait indefinitely if no other thread sets the value in such a way that `pred` returns true when applied to it.

See also: `wait-for`, `future`, `force`, `wait-until*`. →index

**Warning:** This cannot be used for synchronization of multiple tasks due to potential race-conditions. →topic

### 4.726 `wait-until*` : procedure/4

Usage: (`wait-until*` dict key pred timeout)

Blocks execution until the unary predicate `pred` returns true for the value at `key` in `dict`, or `timeout` nanoseconds have passed, and returns the value or nil if waiting timed out. If `timeout` is negative, then the function waits potentially indefinitely without any timeout. If a non-nil key is not found, the function sleeps at least *sync-wait-lower-bound* nanoseconds and up to *sync-wait-upper-bound* nanoseconds until it looks for the key again.

See also: `future`, `force`, `wait-for`, `wait-until*`, `wait-until`. →index

**Warning: This cannot be used for synchronization of multiple tasks due to potential race-conditions.** →topic

#### 4.727 `warn` : procedure/1 or more

Usage: (`warn` `msg` [`args...`])

Output the warning message `msg` in error colors. The optional `args` are applied to the message as in `fmt`. The message should not end with a newline.

See also: `error`. →index →topic

#### 4.728 `week+` : procedure/2

Usage: (`week+` `date- n)` => `date`

Adds `n` weeks to the given date `date in datelist format and returns the new datelist.`

See also: `sec+`, `minute+`, `hour+`, `day+`, `month+`, `year+`, `now`. →index →topic

#### 4.729 `week-of-date` : procedure/3

Usage: (`week-of-date` `Y` `M` `D`) => `int`

Return the week of the date in the year given by year `Y`, month `M`, and day `D`.

See also: `day-of-week`, `datestr->datelist`, `date->epoch-ns`, `epoch-ns->datelist`, `datestr`, `datestr*`, `now`. →index →topic

#### 4.730 `when` : macro/1 or more

Usage: (`when` `cond` `expr` ...) => `any`

Evaluate the expressions `expr` if `cond` is true, returns void otherwise.

See also: `if`, `cond`, `unless`. →index →topic

#### 4.731 `when-permission` : macro/1 or more

Usage: (`when-permission` `perm` `body` ...) => `any`

Execute the expressions in `body` if and only if the symbolic permission `perm` is available.

See also: `permission?`. →index →topic

#### 4.732 `while` : macro/1 or more

Usage: (`while` `test` `body` ...) => `any`

Evaluate the expressions in `body` while `test` is not nil.

See also: `letrec`, `dotimes`, `dolist`. →index →topic

#### 4.733 `window-fixed-size?` : procedure/1

Usage: (`window-fixed-size?` `window`) => `bool`

Return nil if `window` is fixed size, true otherwise.

See also: `set-window-fixed-size`. →index →topic

#### 4.734 `window-full-screen?` : procedure/1

Usage: (`window-full-screen?` `window`) => `bool`

Return nil if `window` is full screen, true otherwise.

See also: `set-window-full-screen`, `center-window-on-screen`. →index →topic

#### 4.735 `window-padded?` : procedure/1

Usage: (`window-padded?` `window`) => `bool`

Return nil if `window` is padded, true otherwise.

See also: `set-window-padded`, `set-window-full-screen`, `center-window-on-screen`.  
→index →topic

### 4.736 `with-colors` : procedure/3

Usage: (`with-colors` `textcolor` `backcolor` `proc`)

Execute `proc` for display side effects, where the default colors are set to `textcolor` and `backcolor`. These are color specifications like in `the-color`. After `proc` has finished or if an error occurs, the default colors are restored to their original state.

See also: `the-color`, `color`, `set-color`, `with-final`. →index →topic

### 4.737 `with-error-handler` : macro/2 or more

Usage: (`with-error-handler` `handler` `body` ...)

Evaluate the forms of the `body` with error handler `handler` in place. The handler is a procedure that takes the error as argument and handles it. If an error occurs in `handler`, a default error handler is used. Handlers are only active within the same thread.

See also: `with-final`. →index →topic

### 4.738 `with-final` : macro/2 or more

Usage: (`with-final` `finalizer` `body` ...)

Evaluate the forms of the `body` with the given finalizer as error handler. If an error occurs, then `finalizer` is called with that error and nil. If no error occurs, `finalizer` is called with nil as first argument and the result of evaluating all forms of `body` as second argument.

See also: `with-error-handler`. →index →topic

### 4.739 `with-mutex-lock` : macro/1 or more

Usage: (`with-mutex-lock` `m` ...) => `any`

Execute the body with mutex `m` locked for writing and unlock the mutex afterwards.

See also: `with-mutex-rlock`, `make-mutex`, `mutex-lock`, `mutex-rlock`, `mutex-unlock`, `mutex-runlock`. →index

**Warning: Make sure to never lock the same mutex twice from the same task, otherwise a deadlock will occur!**



#### 4.740 `with-mutex-rlock`: macro/1 or more

Usage: (`with-mutex-rlock` *m* ...) => *any*

Execute the body with mutex *m* locked for reading and unlock the mutex afterwards.

See also: `with-mutex-lock`, `make-mutex`, `mutex-lock`, `mutex-rlock`, `mutex-unlock`, `mutex-runlock`. →index →topic

#### 4.741 `write`: procedure/2

Usage: (`write` *p* *datum*) => *int*

Write *datum* to output port *p* and return the number of bytes written.

See also: `write-binary`, `write-binary-at`, `read`, `close`, `open`. →index →topic

#### 4.742 `write-binary`: procedure/4

Usage: (`write-binary` *p* *buff* *n* *offset*) => *int*

Write *n* bytes starting at *offset* in binary blob *buff* to the stream port *p*. This function returns the number of bytes actually written.

See also: `write-binary-at`, `read-binary`, `write`, `close`, `open`. →index →topic

#### 4.743 `write-binary-at`: procedure/5

Usage: (`write-binary-at` *p* *buff* *n* *offset* *fpos*) => *int*

Write *n* bytes starting at *offset* in binary blob *buff* to the seekable stream port *p* at the stream position *fpos*. If there is not enough data in *p* to overwrite at position *fpos*, then an error is caused and only part of the data might be written. The function returns the number of bytes actually written.

See also: `read-binary`, `write-binary`, `write`, `close`, `open`. →index →topic

#### 4.744 `write-string`: procedure/2

Usage: (`write-string` *p* *s*) => *int*

Write string *s* to output port *p* and return the number of bytes written. LF are *not* automatically converted to CR LF sequences on windows.

See also: `write`, `write-binary`, `write-binary-at`, `read`, `close`, `open`. →index →topic

#### 4.745 `write-zimage` : procedure/4

Usage: (`write-zimage` `out` `min-version` `info` `entry-point`)=> `list`

Write the current state of the system as an zimage to stream `out`. The `min-version` argument designates the minimum system version required to load the zimage. The `info` argument should be a list whose first argument is a human-readable string explaining the purpose of the zimage and remainder is user data. The `entry-point` is either nil or an expression that can be evaluated to start the zimage after it has been loaded with `run-zimage`. The procedure returns a header with information of the zimage.

See also: `save-zimage`, `read-zimage`, `load-zimage`, `current-zimage`, `externalize`. →index →topic

#### 4.746 `year+` : procedure/2

Usage: (`month+` `date` `n`)=> `date`

Adds `n` years to the given date `date` in datelist format and returns the new datelist.

See also: `sec+`, `minute+`, `hour+`, `day+`, `week+`, `month+`, `now`. →index →topic

#### 4.747 `zimage-header` : procedure/1

Usage: (`zimage-header` `fi`)=> `li`

Return the zimage header from file `fi`.

See also: `load-zimage`, `run-zimage`. →index →topic

#### 4.748 `zimage-loadable?` : procedure/1 or more

Usage: (`zimage-loadable?` `fi`)

Checks whether the file `fi` is loadable. This does not check whether the file actually is an zimage file, so you can only use this on readable lisp files.

See also: `zimage-runable?`, `load-zimage`, `save-zimage`, `current-zimage`. →index →topic

**4.749 `zimage-runable?` : procedure/1 or more**

Usage: (`zimage-runable?` [`sel`] `fi`)

Returns the non-nil entry-point of the zimage if the the zimage in file `fi` can be run, nil otherwise.

See also: `load-zimage`, `zimage-loadable?`, `save-zimage`, `current-zimage`. →index →topic