Z3S5 Lisp Reference Manual

by Erich Rast and all Help system contributors

2022-7-27 11:08

For Z3S5 Lisp Version 2.3.2+3f2a3c7 with installed modules (tasks help beep 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 By Topics

2.1 Arrays

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

2.1.1 `array` : procedure/0 or more

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

Create an array containing the arguments given to it.

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

2.1.2 `array-copy` : procedure/1

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

Return a copy of `arr`.

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

2.1.3 `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?`.

2.1.4 `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?`.

2.1.5 `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`.

2.1.6 `array-len` : procedure/1

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

Return the length of array `arr`.

See also: `len`.

2.1.7 `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?`.

2.1.8 `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?`.

2.1.9 `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?`.

2.1.10 `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`.

2.1.11 `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`.

2.1.12 `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.

See also: `array-ref`, `array-len`.

2.1.13 `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`.

2.1.14 `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?`.

2.1.15 `array?` : procedure/1

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

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

See also: `seq?`, `array`.

2.1.16 `build-array` : procedure/2

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

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

See also: `array`, `array?`.

2.2 Binary Manipulation

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

2.2.1 `bitand` : procedure/2

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

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

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

2.2.2 `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`.

2.2.3 `bitor` : procedure/2

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

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

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

2.2.4 `bitshl` : procedure/2

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

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

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

2.2.5 `bitshr` : procedure/2

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

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

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

2.2.6 `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`.

2.2.7 `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`.

2.2.8 `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`.

2.2.9 `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?`.

2.2.10 `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?`.

2.2.11 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`.

2.2.12 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`.

2.3 Boxed Data Structures

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

2.3.1 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: `gfx.reset`.

2.4 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.

2.4.1 `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!`.

2.4.2 `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!`.

2.4.3 `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!`.

2.4.4 `cpunum` : procedure/0

Usage: `(cpunum)`

Return the number of cpu cores of this machine.

See also: `sys`.

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

2.4.5 `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`.

2.4.6 `cwait` : procedure/3

Usage: `(cwait sym value timeout)`

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

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

2.4.7 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 prodedure 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`.

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

2.4.8 `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`.

2.4.9 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`.

2.4.10 `make-mutex` : procedure/1

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

Create a new mutex.

See also: `mutex-lock`, `mutex-unlock`, `mutex-rlock`, `mutex-runlock`.

2.4.11 `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`.

2.4.12 `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`.

2.4.13 `mutex-runlock` : procedure/1

Usage: `(mutex-runlock m)`

Unlock the mutex `m` from reading.

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

2.4.14 `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`.

2.4.15 `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`.

2.4.16 `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`.

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

2.4.17 `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`.

2.4.18 `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`.

2.4.19 `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`.

2.4.20 `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`.

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

2.4.21 `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`.

2.4.22 `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-`.

2.4.23 `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`.

2.4.24 `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`.

2.4.25 `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`.

2.4.26 `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`.

2.4.27 `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*`.

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

2.4.28 `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*`.

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

2.4.29 `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*`.

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

2.4.30 `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*`.

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

2.4.31 `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`.

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

2.4.32 `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`.

2.5 Console Input & Output

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

2.5.1 `nl` : procedure/0

Usage: `(nl)`

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

See also: `out`, `outy`, `output-at`.

2.5.2 `prin1` : procedure/1

Usage: `(prin1 s)`

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

See also: `princ`, `terpri`, `out`, `outy`.

2.5.3 `princ` : procedure/1

Usage: `(princ s)`

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

See also: `prin1`, `terpri`, `out`, `outy`.

2.5.4 `print` : procedure/1

Usage: `(print x)`

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

See also: `prin1`, `princ`.

2.5.5 `terpri` : procedure/0

Usage: `(terpri)`

Advance the host OS terminal to the next line.

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

2.6 Data Type Conversion

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

2.6.1 `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`.

2.6.2 `array->list` : procedure/1

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

Convert array `arr` into a list.

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

2.6.3 `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`.

2.6.4 `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`.

2.6.5 `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`.

2.6.6 `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?`.

2.6.7 `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`.

2.6.8 `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`.

2.6.9 `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?`.

2.6.10 `char->str` : procedure/1

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

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

See also: `str->char`.

2.6.11 `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`.

2.6.12 `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`.

2.6.13 `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`.

2.6.14 `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`.

2.6.15 `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`.

2.6.16 `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`.

2.6.17 `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`.

2.6.18 `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`.

2.6.19 `list->array` : procedure/1

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

Convert the list `li` to an array.

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

2.6.20 `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`.

2.6.21 `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`.

2.6.22 `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`.

2.6.23 `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`.

2.6.24 `str->blob` : procedure/1

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

Convert string `s` into a blob.

See also: `blob->str`.

2.6.25 `str->char` : procedure/1

Usage: `(str->char s)`

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

See also: `char->str`.

2.6.26 `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`.

2.6.27 `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`.

2.6.28 `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 expession parser works.

See also: `str->expr`, `expr->str`, `openstr`, `internalize`, `externalize`.

2.6.29 `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`.

2.6.30 `str->sym` : procedure/1

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

Convert a string into a symbol.

See also: `sym->str`, `intern`, `make-symbol`.

2.6.31 `sym->str` : procedure/1

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

Convert a symbol into a string.

See also: `str->sym`, `intern`, `make-symbol`.

2.7 Special Data Structures

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

2.7.1 `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`.

2.7.2 `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`.

2.7.3 `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`.

2.7.4 `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!`.

2.7.5 `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`.

2.7.6 `make-queue` : procedure/0

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

Make a synchronized queue.

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

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

2.7.7 `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 very argument.

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

2.7.8 `make-stack` : procedure/0

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

Make a synchronized stack.

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

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

2.7.9 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`.

2.7.10 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`.

2.7.11 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`.

2.7.12 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`.

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!

2.7.13 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`.

2.7.14 `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?`.

2.7.15 `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?`.

2.7.16 `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?`.

2.7.17 `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?`.

2.7.18 `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?`.

2.7.19 `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?`.

2.7.20 `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?`.

2.7.21 `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?`.

2.7.22 `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?`.

2.7.23 `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`.

2.7.24 `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`.

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!

2.7.25 `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`.

2.8 Dictionaries

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

2.8.1 `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`.

2.8.2 `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`.

2.8.3 `dict-copy` : procedure/1

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

Return a copy of dict `d`.

See also: `dict`, `dict?`.

2.8.4 `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`.

2.8.5 `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`.

2.8.6 `dict-map` : procedure/2

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

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

See also: `dict-map!`, `map`.

2.8.7 `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`.

2.8.8 `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`.

2.8.9 `dict?` : procedure/1

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

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

See also: `dict`.

2.8.10 `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`.

2.8.11 `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`.

2.8.12 `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`.

2.8.13 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`.

2.8.14 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`.

2.8.15 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`.

2.8.16 pushstacked : procedure/3

Usage: `(pushstacked dict key datum)`

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

See also: `getstacked`, `popstacked`.

2.8.17 set : procedure/3

Usage: `(set d key value)`

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

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

2.8.18 `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`.

2.9 Equality Predicates

Equality predicates are used to test whether two values are equal in some sense.

2.9.1 `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?`.

2.9.2 `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?`.

Warning: Deprecated.

2.10 File Input & Output

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

2.10.1 `close` : procedure/1

Usage: `(close p)`

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

See also: `open`, `stropen`.

2.10.2 `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`.

2.10.3 `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`.

2.10.4 `fdelete` : procedure/1

Usage: `(fdelete path)`

Removes the file or directory at `path`.

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

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

2.10.5 `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`.

2.10.6 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`.

2.10.7 read : procedure/1

Usage: `(read p)=> any`

Read an expression from input port `p`.

See also: `input`, `write`.

2.10.8 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`.

2.10.9 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`.

2.10.10 `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`.

2.10.11 `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`.

2.10.12 `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`.

2.10.13 `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`.

2.10.14 `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`.

2.11 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.

2.11.1 `fl.abs` : procedure/1

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

Return the absolute value of `x`.

See also: **`float`**, `*`.

2.11.2 `fl.acos` : procedure/1

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

Return the arc cosine of `x`.

See also: `fl.cos`.

2.11.3 `fl.asin` : procedure/1

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

Return the arc sine of `x`.

See also: `fl.acos`.

2.11.4 `fl.asinh` : procedure/1

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

Return the inverse hyperbolic sine of `x`.

See also: `fl.cosh`.

2.11.5 `fl.atan` : procedure/1

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

Return the arctangent of `x` in radians.

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

2.11.6 `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`.

2.11.7 `fl.atanh` : procedure/1

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

Return the inverse hyperbolic tangent of `x`.

See also: `fl.atan`.

2.11.8 `fl.cbrt` : procedure/1

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

Return the cube root of `x`.

See also: `fl.sqrt`.

2.11.9 `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`.

2.11.10 `fl.cos` : procedure/1

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

Return the cosine of `x`.

See also: `fl.sin`.

2.11.11 `fl.cosh` : procedure/1

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

Return the hyperbolic cosine of `x`.

See also: `fl.cos`.

2.11.12 `fl.dim` : procedure/2

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

Return the maximum of x, y or 0.

See also: `max`.

2.11.13 `fl.erf` : procedure/1

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

Return the result of the error function of `x`.

See also: `fl.erfc`, `fl.dim`.

2.11.14 `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`.

2.11.15 `fl.erfcinv` : procedure/1

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

Return the inverse of (fl.erfc `x`).

See also: `fl.erfc`.

2.11.16 `fl.erfinv` : procedure/1

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

Return the inverse of (fl.erf `x`).

See also: `fl.erf`.

2.11.17 `fl.exp` : procedure/1

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

Return e^`x`, the base-e exponential of `x`.

See also: `fl.exp`.

2.11.18 `fl.exp2` : procedure/2

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

Return 2^`x`, the base-2 exponential of `x`.

See also: `fl.exp`.

2.11.19 `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`.

2.11.20 `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`**.

2.11.21 `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: `*`, `+`.

2.11.22 `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` == `frac` × 2^`exp` where the absolute value of `frac` is in the interval [0.5, 1).

See also: `fl.exp`.

2.11.23 `fl.gamma` : procedure/1

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

Compute the Gamma function of `x`.

See also: `fl.lgamma`.

2.11.24 `fl.hypot` : procedure/2

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

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

See also: `fl.sqrt`.

2.11.25 `fl.ilogb` : procedure/1

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

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

See also: `fl.exp2`.

2.11.26 `fl.inf` : procedure/1

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

Return positive 64 bit floating point infinity +INF if `x` >= 0 and negative 64 bit floating point finfinity -INF if `x` < 0.

See also: `fl.is-nan?`.

2.11.27 `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`.

2.11.28 `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`.

2.11.29 `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`.

2.11.30 `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`.

2.11.31 `fl.ldexp` : procedure/2

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

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

See also: `fl.frexp`.

2.11.32 `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`.

2.11.33 `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`.

2.11.34 `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`.

2.11.35 `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`.

2.11.36 `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`.

2.11.37 `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`.

2.11.38 `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`.

2.11.39 `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`.

2.11.40 `fl.mod` : procedure/2

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

Return the floating point remainder of `x` / `y`.

See also: `fl.remainder`.

2.11.41 `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`.

2.11.42 `fl.nan` : procedure/1

Usage: `(fl.nan)=> fl`

Return the IEEE 754 not-a-number value.

See also: `fl.is-nan?`, `fl.inf`.

2.11.43 `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`.

2.11.44 `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`.

2.11.45 `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`.

2.11.46 `fl.remainder` : procedure/2

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

Return the IEEE 754 floating-point remainder of `x` / `y`.

See also: `fl.mod`.

2.11.47 `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`**.

2.11.48 `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`**.

2.11.49 `fl.signbit` : procedure/1

Usage: `(fl.signbit x)=> bool`

Return true if `x` is negative, nil otherwise.

See also: `fl.abs`.

2.11.50 `fl.sin` : procedure/1

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

Return the sine of `x`.

See also: `fl.cos`.

2.11.51 `fl.sinh` : procedure/1

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

Return the hyperbolic sine of `x`.

See also: `fl.sin`.

2.11.52 `fl.sqrt` : procedure/1

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

Return the square root of `x`.

See also: `fl.pow`.

2.11.53 `fl.tan` : procedure/1

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

Return the tangent of `x` in radian.

See also: `fl.tanh`, `fl.sin`, `fl.cos`.

2.11.54 `fl.tanh` : procedure/1

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

Return the hyperbolic tangent of `x`.

See also: `fl.tan`, `flsinh`, `fl.cosh`.

2.11.55 `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`.

2.11.56 `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`.

2.11.57 `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`.

2.11.58 `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`.

2.12 Help System

This section lists functions related to the built-in help system.

2.12.1 *`help`* : dict

Usage: `*help*`

Dict containing all help information for symbols.

See also: `help`, `defhelp`, `apropos`.

2.12.2 `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*`.

2.12.3 `help` : macro/1

Usage: `(help sym)`

Display help information about `sym` (unquoted).

See also: `defhelp`, `help-entry`, `*help*`, `apropos`.

2.12.4 help->manual-entry : nil

Usage: `(help->manual-entry key [level])=> 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.

See also: `help`.

2.12.5 `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`.

2.12.6 `help-entry` : procedure/1

Usage: `(help-entry sym)=> list`

Get usage and help information for `sym`.

See also: `defhelp`, `help`, `apropos`, `*help*`.

2.12.7 `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`.

2.12.8 `help-topics` : procedure/0

Usage: `(help-topics)=> li`

Obtain a list of help topics for commands.

See also: `help`, `help-topic`, `apropos`.

2.12.9 `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`.

2.13 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.

2.13.1 `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`.

2.13.2 `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`.

2.13.3 `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`.

2.13.4 `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`.

2.13.5 `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`.

2.13.6 `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`.

2.13.7 `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`.

2.13.8 `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`.

2.13.9 `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`.

2.13.10 `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`.

2.13.11 `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`.

2.14 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.

2.14.1 `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`.

2.14.2 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`.

2.14.3 append : procedure/1 or more

Usage: `(append li1 li2 ...)=> li`

Concatenate the lists given as arguments.

See also: `cons`.

2.14.4 apply : procedure/2

Usage: `(apply proc arg)=> any`

Apply function `proc` to argument list `arg`.

See also: `functional?`.

2.14.5 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?`.

2.14.6 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?`.

2.14.7 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?`.

2.14.8 `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?`.

2.14.9 `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`.

2.14.10 `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`.

2.14.11 `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`.

2.14.12 `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`.

2.14.13 `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`.

2.14.14 `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`.

2.14.15 `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`.

2.14.16 `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?`.

2.14.17 `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`.

2.14.18 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`.

2.14.19 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`.

2.14.20 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`.

2.14.21 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`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`.

2.14.22 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`, `nth`, `1st`, `2nd`, `3rd`.

2.14.23 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`.

2.14.24 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?`.

2.14.25 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`.

2.14.26 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?`.

2.14.27 cons? : procedure/1

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

return true if `x` is not an atom, nil otherwise.

See also: `atom?`.

2.14.28 `count-partitions` : procedure/2

Usage: `(count-partitions m k)=> int`

Return the number of partitions for divding `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`.

2.14.29 `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`.

2.14.30 `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`.

2.14.31 `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`**.

2.14.32 `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?`.

2.14.33 `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`.

2.14.34 `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`.

2.14.35 `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 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`.

2.14.36 `identity` : procedure/1

Usage: `(identity x)`

Return `x`.

See also: `apply`, `equal?`.

2.14.37 `if` : macro/3

Usage: `(if cond expr1 expr2)=> any`

Evaluate `expr1` if `cond` is true, otherwise evaluate `expr2`.

See also: `cond`, `when`, `unless`.

2.14.38 `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`.

2.14.39 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?`.

2.14.40 `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`.

2.14.41 `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`.

2.14.42 `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`.

2.14.43 `list` : procedure/0 or more

Usage: `(list [args] ...)=> li`

Create a list from all `args`. The arguments must be quoted.

See also: `cons`.

2.14.44 `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?`.

2.14.45 `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?`.

2.14.46 `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`.

2.14.47 `list-last` : procedure/1

Usage: `(list-last li)=> any`

Return the last element of `li`.

See also: `reverse`, `nreverse`, `car`, `1st`, `last`.

2.14.48 `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`.

2.14.49 `list-reverse` : procedure/1

Usage: `(list-reverse li)=> li`

Create a reversed copy of `li`.

See also: `reverse`, `array-reverse`, `str-reverse`.

2.14.50 `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`.

2.14.51 `list?` : procedure/1

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

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

See also: `cons?`, `atom?`, **`null`**`?`.

2.14.52 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`, `funcional?`, `macro?`, `closure?`.

2.14.53 `mapcar` : procedure/2

Usage: `(mapcar li proc)=> li`

Return the list obtained from applying `proc` to each elements in `li`.

See also: `map`, `foreach`.

2.14.54 `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?`.

2.14.55 `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?`.

2.14.56 `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`.

2.14.57 `not` : procedure/1

Usage: `(not x)=> bool`

Return true if `x` is nil, nil otherwise.

See also: `and`, `or`.

2.14.58 `nreverse` : procedure/1

Usage: `(nreverse li)=> li`

Destructively reverse `li`.

See also: `reverse`.

2.14.59 `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`.

2.14.60 `null?` : procedure/1

Usage: `(null? li)=> bool`

Return true if `li` is nil, nil otherwise.

See also: `not`, `list?`, `cons?`.

2.14.61 `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?`.

2.14.62 `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`.

2.14.63 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`.

2.14.64 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-spliced 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`.

2.14.65 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`.

2.14.66 replacd : procedure/2

Usage: `(rplacd li1 li2)=> li`

Destructively replace the cdr of `li1` with `li2` and return the result afterwards.

See also: `rplaca`.

2.14.67 rplaca : procedure/2

Usage: `(rplaca li a)=> li`

Destructively mutate `li` such that its car is `a`, return the list afterwards.

See also: `rplacd`.

2.14.68 `setcar` : procedure/1

Usage: `(setcar li elem)=> li`

Mutate `li` such that its car is `elem`. Same as rplaca.

See also: `rplaca`, `rplacd`, `setcdr`.

2.14.69 `setcdr` : procedure/1

Usage: `(setcdr li1 li2)=> li`

Mutate `li1` such that its cdr is `li2`. Same as rplacd.

See also: `rplacd`, `rplaca`, `setcar`.

2.14.70 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`.

2.14.71 `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 otheriwse.

See also: `array-sort`.

2.14.72 sort-symbols : nil

Usage: `(sort-symbols li)=> list`

Sort the list of symbols `li` alphabetically.

See also: `out`, `dp`, `du`, `dump`.

2.14.73 `sym?` : procedure/1

Usage: `(sym? sym)=> bool`

Return true if `sym` is a symbol, nil otherwise.

See also: `str?`, `atom?`.

2.14.74 `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`.

2.14.75 `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`**?.

2.14.76 `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`.

2.14.77 `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`.

2.15 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.

2.15.1 % : procedure/2

Usage: `(% x y)=> num`

Compute the remainder of dividing number `x` by `y`.

See also: `mod`, `/`.

2.15.2 * : procedure/0 or more

Usage: `(* [args] ...)=> num`

Multiply all `args`. Special cases: *() is 1 and (* x) is x.

See also: `+`, `-`, `/`.

2.15.3 + : procedure/0 or more

Usage: `(+ [args] ...)=> num`

Sum up all `args`. Special cases: (+) is 0 and (+ x) is x.

See also: `-`, `*`, `/`.

2.15.4 - : procedure/1 or more

Usage: `(- x [y1] [y2] ...)=> num`

Subtract `y1`, `y2`, ..., from `x`. Special case: (- x) is -x.

See also: `+`, `*`, `/`.

2.15.5 / : 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: `+`, `*`, `-`.

2.15.6 /= : procedure/2

Usage: `(/= x y)=> bool`

Return true if number `x` is not equal to `y`, nil otherwise.

See also: `>`, `>=`, `<`, `<=`.

2.15.7 < : procedure/2

Usage: `(< x y)=> bool`

Return true if `x` is smaller than `y`.

See also: `<=`, `>=`, `>`.

2.15.8 <= : procedure/2

Usage: `(<= x y)=> bool`

Return true if `x` is smaller than or equal to `y`, nil otherwise.

See also: `>`, `<`, `>=`, `/=`.

2.15.9 = : procedure/2

Usage: `(= x y)=> bool`

Return true if number `x` equals number `y`, nil otherwise.

See also: `eql?`, `equal?`.

2.15.10 > : procedure/2

Usage: `(> x y)=> bool`

Return true if `x` is larger than `y`, nil otherwise.

See also: `<`, `>=`, `<=`, `/=`.

2.15.11 >= : procedure/2

Usage: `(>= x y)=> bool`

Return true if `x` is larger than or equal to `y`, nil otherwise.

See also: `>`, `<`, `<=`, `/=`.

2.15.12 abs : procedure/1

Usage: `(abs x)=> num`

Returns the absolute value of number `x`.

See also: `*`, `-`, `+`, `/`.

2.15.13 add1 : procedure/1

Usage: `(add1 n)=> num`

Add 1 to number `n`.

See also: `sub1`, `+`, `-`.

2.15.14 div : procedure/2

Usage: `(div n k)=> int`

Integer division of `n` by `k`.

See also: `truncate`, `/`, **`int`**.

2.15.15 `even?` : procedure/1

Usage: `(even? n)=> bool`

Returns true if the integer `n` is even, nil if it is not even.

See also: `odd?`.

2.15.16 `float` : procedure/1

Usage: `(float n)=> float`

Convert `n` to a floating point value.

See also: **`int`**.

2.15.17 `int` : procedure/1

Usage: `(int n)=> int`

Return `n` as an integer, rounding down to the nearest integer if necessary.

See also: **`float`**.

Warning: If the number is very large this may result in returning the maximum supported integer number rather than the number as integer.

2.15.18 `max` : procedure/1 or more

Usage: `(max x1 x2 ...)=> num`

Return the maximum of the given numbers.

See also: `min`, `minmax`.

2.15.19 `min` : procedure/1 or more

Usage: `(min x1 x2 ...)=> num`

Return the minimum of the given numbers.

See also: `max`, `minmax`.

2.15.20 `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`.

2.15.21 `mod` : procedure/2

Usage: `(mod x y)=> num`

Compute `x` modulo `y`.

See also: `%`, `/`.

2.15.22 `odd?` : procedure/1

Usage: `(odd? n)=> bool`

Returns true if the integer `n` is odd, nil otherwise.

See also: `even?`.

2.15.23 `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`.

2.15.24 `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`.

2.15.25 `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`.

2.15.26 `sub1` : procedure/1

Usage: `(sub1 n)=> num`

Subtract 1 from `n`.

See also: `add1`, `+`, `-`.

2.15.27 `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`.

2.16 Semver Semantic Versioning

The `semver` package provides functions to deal with the validation and parsing of semantic versioning strings.

2.16.1 `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`.

2.16.2 `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`.

2.16.3 `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`.

2.16.4 `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`.

2.16.5 `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`.

2.16.6 `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`.

2.16.7 `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`.

2.16.8 `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`.

2.17 Sequence Functions

Sequences are either strings, lists, or arrays. Sequences 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.

2.17.1 `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`.

2.17.2 `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`.

2.17.3 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`.

2.17.4 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`.

2.17.5 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`.

2.17.6 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`.

2.17.7 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`.

2.17.8 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`.

2.17.9 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`.

2.17.10 9th : procedure/1 or more

Usage: `(9th seq [default])=> any`

Get the nineth 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`.

2.17.11 `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?`.

2.17.12 `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?`.

2.17.13 `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`.

2.17.14 `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?`.

2.17.15 `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`.

2.17.16 `len` : procedure/1

Usage: `(len seq)=> int`

Return the length of `seq`. Works for lists, strings, arrays, and dicts.

See also: `seq?`.

2.17.17 `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`.

2.17.18 `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`.

2.17.19 `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`.

2.17.20 `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`.

2.17.21 `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`.

2.17.22 `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`.

2.17.23 `seq?` : procedure/1

Usage: `(seq? seq)=> bool`

Return true if `seq` is a sequence, nil otherwise.

See also: `list`, `array`, `string`, `slice`, `nth`.

2.17.24 `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?`.

2.17.25 `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?`.

2.18 Sound Support

Only a few functions are provided for sound support.

2.18.1 `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: `play-sound`, `load-sound`.

2.18.2 `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: `play-sound`, `play-music`.

2.19 String Manipulation

These functions all manipulate strings in one way or another.

2.19.1 `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`.

2.19.2 `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`.

2.19.3 `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`.

2.19.4 `spaces` : procedure/1

Usage: `(spaces n)=> str`

Create a string consisting of `n` spaces.

See also: `strbuild`, `strleft`, `strright`.

2.19.5 `str+` : procedure/0 or more

Usage: `(str+ [s] ...)=> str`

Append all strings given to the function.

See also: `str?`.

2.19.6 `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`.

2.19.7 `str-empty?` : procedure/1

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

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

See also: `strlen`.

2.19.8 `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?`.

2.19.9 `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?`.

2.19.10 `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`.

2.19.11 `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 macthing char is found, nil is returned.

See also: `strsplit`, `chars`, `inchars`.

2.19.12 `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`.

2.19.13 `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`.

2.19.14 `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`.

2.19.15 `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`.

2.19.16 `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`.

2.19.17 `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`.

2.19.18 `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`.

2.19.19 `str-reverse` : procedure/1

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

Reverse string `s`.

See also: `reverse`, `array-reverse`, `list-reverse`.

2.19.20 `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`.

2.19.21 `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`.

2.19.22 `strbuild` : procedure/2

Usage: `(strbuild s n)=> str`

Build a string by repeating string `s`` n` times.

See also: `str+`.

2.19.23 `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`.

2.19.24 `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`.

2.19.25 `strcnt` : procedure/2

Usage: `(strcnt s del)=> int`

Returnt the number of non-overlapping substrings `del` in `s`.

See also: `strsplit`, `str-index`.

2.19.26 `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`.

2.19.27 `strlen` : procedure/1

Usage: `(strlen s)=> int`

Return the length of `s`.

See also: `len`, `seq?`, `str?`.

2.19.28 `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`.

2.19.29 `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`.

2.19.30 `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`.

2.19.31 `stropen` : procedure/1

Usage: `(stropen s)=> streamport`

Open the string `s` as input stream.

See also: `open`, `close`.

2.19.32 `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`.

2.19.33 `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?`.

2.20 System Functions

These functions concern the inner workings of the Lisp interpreter. Your warranty might be void if you abuse them!

2.20.1 *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*`.

2.20.2 `*error-printer*` : procedure/1

Usage: `(*error-printer* err)`

The global printer procedure which takes an error and prints it.

See also: `error`.

2.20.3 *last-error* : sym

Usage: `*last-error* => str`

Contains the last error that has occurred.

See also: `*error-printer*`, `*error-handler*`.

Warning: This may only be used for debugging! Do *not* use this for error handling, it will surely fail!

2.20.4 *reflect* : symbol

Usage: `*reflect* => li`

The list of feature identifiers as symbols that this Lisp implementation supports.

See also: `feature?`, `on-feature`.

2.20.5 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`.

2.20.6 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`.

Warning: Internal use only.

2.20.7 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`.

2.20.8 `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`.

2.20.9 `bound?` : macro/1

Usage: `(bound? sym)=> bool`

Return true if a value is bound to the symbol `sym`, nil otherwise.

See also: `bind`, `setq`.

2.20.10 `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?`.

2.20.11 `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 colllection 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`.

Warning: There should rarely be a use for this. Try to use less memory-consuming data structures instead.

2.20.12 `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*`.

2.20.13 `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`.

2.20.14 `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*`.

2.20.15 `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?`.

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!

2.20.16 `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?`.

2.20.17 `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?`.

2.20.18 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`.

2.20.19 `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`.

2.20.20 `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`**.

2.20.21 `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`.

2.20.22 `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: `n/a`.

2.20.23 `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`.

2.20.24 `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`.

2.20.25 `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`.

2.20.26 `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`.

2.20.27 `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`.

2.20.28 `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`.

2.20.29 `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`.

2.20.30 `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`.

2.20.31 `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`.

2.20.32 `functional-arity` : procedure/1

Usage: `(functional-arity proc)=> int`

Return the arity of a functional `proc`.

See also: `functional?`, `functional-has-rest?`.

2.20.33 `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`.

2.20.34 `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?`.

2.20.35 `gensym` : procedure/0

Usage: `(gensym)=> sym`

Return a new symbol guaranteed to be unique during runtime.

See also: `nonce`.

2.20.36 hook : procedure/1

Usage: `(hook symbol)`

Lookup the internal hook number from a symbolic name.

See also: `*hooks*`, `add-hook`, `remove-hook`, `remove-hooks`.

2.20.37 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`.

2.20.38 intern : procedure/1

Usage: `(intern s)=> sym`

Create a new interned symbol based on string `s`.

See also: `gensym`, `str->sym`, `make-symbol`.

2.20.39 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`.

Warning: This function currently only returns the binding but this behavior might change in future.

2.20.40 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?`.

Warning: What counts as an intrinsic or not may change from version to version. This is for internal use only.

2.20.41 `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?`.

2.20.42 `make-symbol` : procedure/1

Usage: `(make-symbol s)=> sym`

Create a new symbol based on string `s`.

See also: `str->sym`.

2.20.43 `memstats` : procedure/0

Usage: `(memstats)=> dict`

Return a dict with detailed memory statistics for the system.

See also: `collect-garbage`.

2.20.44 `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`.

2.20.45 `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*`.

2.20.46 `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`.

2.20.47 `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`.

2.20.48 `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`.

2.20.49 `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`**.

2.20.50 `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?`.

2.20.51 `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`.

2.20.52 `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`, `when-permission?`, `dict-protect`, `dict-`**`protected?`**, `dict-unprotect`.

2.20.53 `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`.

2.20.54 `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`.

2.20.55 `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`.

2.20.56 `read-eval-reply` : procedure/0

Usage: `(read-eval-reply)`

Start a new read-eval-reply loop.

See also: `end-input`, `sys`.

Warning: Internal use only. This function might not do what you expect it to do.

2.20.57 `remove-hook` : procedure/2

Usage: `(remove-hook hook id)=> bool`

Remove the symbolic or numberic `hook` with `id` and return true if the hook was removed, nil otherwise.

See also: `add-hook`, `remove-hooks`, `replace-hook`.

2.20.58 `remove-hook-internal` : procedure/2

Usage: `(remove-hook-internal hook id)`

Remove the hook with ID `id` from numeric `hook`.

See also: `remove-hook`.

Warning: Internal use only.

2.20.59 `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`.

2.20.60 `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`.

2.20.61 `run-hook` : procedure/1

Usage: `(run-hook hook)`

Manually run the hook, executing all procedures for the hook.

See also: `add-hook`, `remove-hook`.

2.20.62 `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`.

Warning: Internal use only.

2.20.63 `run-selftest` : procedure/1 or more

Usage: `(run-selftest [silent?])=> any`

Run a diagnostic self-test of the Z3S5 Machine. If `silent?` is true, then the self-test returns a list containing a boolean for success, the number of tests performed, the number of successes, the number of errors, and the number of failures. If `silent?` is not provided or nil, then the test progress and results are displayed. An error indicates a problem with the testing, whereas a failure means that an expected value was not returned.

See also: `expect`, `testing`.

2.20.64 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`.

2.20.65 `sleep` : procedure/1

Usage: `(sleep ms)`

Halt the current task execution for `ms` milliseconds.

See also: `sleep-ns`, `time`, `now`, `now-ns`.

2.20.66 `sleep-ns` : procedure/1

Usage: `(sleep-ns n`

Halt the current task execution for `n` nanoseconds.

See also: `sleep`, `time`, `now`, `now-ns`.

2.20.67 `sys-key?` : procedure/1

Usage: `(sys-key? key)=> bool`

Return true if the given sys key `key` exists, nil otherwise.

See also: `sys`, `setsys`.

2.20.68 `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`.

2.20.69 `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`.

2.20.70 `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`.

2.20.71 `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`.

2.20.72 `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`.

2.20.73 `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`.

2.20.74 `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?`.

2.20.75 `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`.

2.20.76 `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`.

2.20.77 `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`.

2.21 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.

2.21.1 `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`.

2.21.2 `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`.

2.21.3 `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`.

2.21.4 `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`.

2.21.5 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`.

2.21.6 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`.

2.21.7 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`.

2.21.8 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`.

2.21.9 hour+ : procedure/2

Usage: `(hour+ dateli n)=> dateli`

Adds `n` hours to the given date `dateli` in datelist format and returns the new datelist.

See also: `sec+`, `minute+`, `day+`, `week+`, `month+`, `year+`, `now`.

2.21.10 `minute+` : procedure/2

Usage: `(minute+ dateli n)=> dateli`

Adds `n` minutes to the given date `dateli` in datelist format and returns the new datelist.

See also: `sec+`, `hour+`, `day+`, `week+`, `month+`, `year+`, `now`.

2.21.11 `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`.

2.21.12 `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`.

2.21.13 `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`.

2.21.14 `now-ns` : procedure/0

Usage: `(now-ns)=> int`

Return the current time in Unix nanoseconds.

See also: `now`, `time`.

2.21.15 `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`.

2.21.16 `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`.

2.21.17 `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`.

2.21.18 `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`.

2.21.19 `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`.

2.22 User Interface

This section lists miscellaneous user interface commands such as color for terminals.

2.22.1 *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`.

2.22.2 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.

See also: `set-color`, `the-color`, `with-colors`.

2.22.3 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`.

2.22.4 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`.

2.22.5 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`.

2.22.6 `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`.

2.22.7 `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`.

2.22.8 `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`, `the-color`, `with-colors`.

2.22.9 `synout` : procedure/1

Usage: `(synout arg)`

Like out, but enforcing a new input line afterwards. This needs to be used when outputing concurrently in a future or task.

See also: `out`, `outy`, `synouty`.

Warning: Concurrent display output can lead to unexpected visual results and ought to be avoided.

2.22.10 `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`.

2.22.11 `the-color-names` : procedure/0

Usage: `(the-color-names)=> li`

Return the list of color names in *colors*.

See also: `*colors*`, `the-color`.

3 Complete Reference

3.1 `%` : procedure/2

Usage: `(% x y)=> num`

Compute the remainder of dividing number `x` by `y`.

See also: `mod`, `/`.

3.2 `*` : procedure/0 or more

Usage: `(* [args] ...)=> num`

Multiply all `args`. Special cases: *() is 1 and (* x) is x.

See also: `+`, `-`, `/`.

3.3 *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`.

3.4 *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*`.

3.5 `*error-printer*` : procedure/1

Usage: `(*error-printer* err)`

The global printer procedure which takes an error and prints it.

See also: `error`.

3.6 *help* : dict

Usage: `*help*`

Dict containing all help information for symbols.

See also: `help`, `defhelp`, `apropos`.

3.7 *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`.

3.8 *last-error* : sym

Usage: `*last-error* => str`

Contains the last error that has occurred.

See also: `*error-printer*`, `*error-handler*`.

Warning: This may only be used for debugging! Do *not* use this for error handling, it will surely fail!

3.9 *reflect* : symbol

Usage: `*reflect* => li`

The list of feature identifiers as symbols that this Lisp implementation supports.

See also: `feature?`, `on-feature`.

3.10 + : procedure/0 or more

Usage: `(+ [args] ...)=> num`

Sum up all `args`. Special cases: (+) is 0 and (+ x) is x.

See also: `-`, `*`, `/`.

3.11 − : procedure/1 or more

Usage: `(- x [y1] [y2] ...)=> num`

Subtract `y1`, `y2`, ..., from `x`. Special case: (- x) is -x.

See also: `+`, `*`, `/`.

3.12 / : 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: `+`, `*`, `-`.

3.13 /= : procedure/2

Usage: `(/= x y)=> bool`

Return true if number `x` is not equal to `y`, nil otherwise.

See also: `>`, `>=`, `<`, `<=`.

3.14 `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`.

3.15 `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`.

3.16 `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`.

3.17 `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`.

3.18 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`.

3.19 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`.

3.20 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`.

3.21 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`.

3.22 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`.

3.23 9th : procedure/1 or more

Usage: `(9th seq [default])=> any`

Get the nineth 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`.

3.24 < : procedure/2

Usage: `(< x y)=> bool`

Return true if `x` is smaller than `y`.

See also: `<=`, `>=`, `>`.

3.25 <= : procedure/2

Usage: `(<= x y)=> bool`

Return true if `x` is smaller than or equal to `y`, nil otherwise.

See also: `>`, `<`, `>=`, `/=`.

3.26 = : procedure/2

Usage: `(= x y)=> bool`

Return true if number `x` equals number `y`, nil otherwise.

See also: `eql?`, `equal?`.

3.27 > : procedure/2

Usage: `(> x y)=> bool`

Return true if `x` is larger than `y`, nil otherwise.

See also: `<`, `>=`, `<=`, `/=`.

3.28 >= : procedure/2

Usage: `(>= x y)=> bool`

Return true if `x` is larger than or equal to `y`, nil otherwise.

See also: `>`, `<`, `<=`, `/=`.

3.29 abs : procedure/1

Usage: `(abs x)=> num`

Returns the absolute value of number `x`.

See also: `*`, `-`, `+`, `/`.

3.30 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`.

3.31 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`.

Warning: Internal use only.

3.32 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`.

3.33 add1 : procedure/1

Usage: `(add1 n)=> num`

Add 1 to number `n`.

See also: `sub1`, `+`, `-`.

3.34 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`.

3.35 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`.

3.36 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`.

3.37 append : procedure/1 or more

Usage: `(append li1 li2 ...)=> li`

Concatenate the lists given as arguments.

See also: `cons`.

3.38 apply : procedure/2

Usage: `(apply proc arg)=> any`

Apply function `proc` to argument list `arg`.

See also: `functional?`.

3.39 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*`.

3.40 array : procedure/0 or more

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

Create an array containing the arguments given to it.

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

3.41 array->list : procedure/1

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

Convert array `arr` into a list.

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

3.42 `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`.

3.43 `array-copy` : procedure/1

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

Return a copy of `arr`.

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

3.44 `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?`.

3.45 `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?`.

3.46 `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`.

3.47 `array-len` : procedure/1

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

Return the length of array `arr`.

See also: `len`.

3.48 `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?`.

3.49 `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?`.

3.50 `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?`.

3.51 `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`.

3.52 `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`.

3.53 `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.

See also: `array-ref`, `array-len`.

3.54 `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`.

3.55 `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?`.

3.56 `array?` : procedure/1

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

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

See also: `seq?`, `array`.

3.57 `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`.

3.58 `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?`.

3.59 `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?`.

3.60 `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?`.

3.61 `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?`.

3.62 `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`.

3.63 `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: `play-sound`, `load-sound`.

3.64 `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`.

3.65 `bitand` : procedure/2

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

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

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

3.66 `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`.

3.67 `bitor` : procedure/2

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

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

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

3.68 `bitshl` : procedure/2

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

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

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

3.69 `bitshr` : procedure/2

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

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

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

3.70 `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`.

3.71 `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?`.

3.72 `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`.

3.73 `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`.

3.74 `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?`.

3.75 `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`.

3.76 `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`.

3.77 `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?`.

3.78 `bound?` : macro/1

Usage: `(bound? sym)=> bool`

Return true if a value is bound to the symbol `sym`, nil otherwise.

See also: `bind`, `setq`.

3.79 `build-array` : procedure/2

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

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

See also: `array`, `array?`.

3.80 `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`.

3.81 `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`.

3.82 `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`.

3.83 `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`.

3.84 `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`.

3.85 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`.

3.86 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`.

3.87 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?`.

3.88 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`.

3.89 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!`.

3.90 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`.

3.91 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`.

3.92 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`.

3.93 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`, `cdddr`, `nth`, `1st`, `2nd`, `3rd`.

3.94 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`, `nth`, `1st`, `2nd`, `3rd`.

3.95 `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`.

3.96 `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!`.

3.97 `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?`.

3.98 `char->str` : procedure/1

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

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

See also: `str->char`.

3.99 `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`.

3.100 `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`.

3.101 `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!`.

3.102 `close` : procedure/1

Usage: `(close p)`

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

See also: `open`, `stropen`.

3.103 `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?`.

3.104 `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 colllection 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`.

Warning: There should rarely be a use for this. Try to use less memory-consuming data structures instead.

3.105 `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.

See also: `set-color`, `the-color`, `with-colors`.

3.106 `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?`.

3.107 `cons?` : procedure/1

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

return true if `x` is not an atom, nil otherwise.

See also: `atom?`.

3.108 `count-partitions` : procedure/2

Usage: `(count-partitions m k)=> int`

Return the number of partitions for divding `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`.

3.109 `cpunum` : procedure/0

Usage: `(cpunum)`

Return the number of cpu cores of this machine.

See also: `sys`.

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

3.110 `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`.

3.111 `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*`.

3.112 `cwait` : procedure/3

Usage: `(cwait sym value timeout)`

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

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

3.113 `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`.

3.114 `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`.

3.115 `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`.

3.116 `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`.

3.117 `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`.

3.118 `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`.

3.119 `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`.

3.120 `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`.

3.121 `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`.

3.122 `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*`.

3.123 `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`.

3.124 `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`.

3.125 `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`.

3.126 `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`.

3.127 `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`.

3.128 `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`.

3.129 `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`.

3.130 `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`.

3.131 `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`.

3.132 `dict-copy` : procedure/1

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

Return a copy of dict `d`.

See also: `dict`, `dict?`.

3.133 `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`.

3.134 `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`.

3.135 `dict-map` : procedure/2

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

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

See also: `dict-map!`, `map`.

3.136 `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`.

3.137 `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`.

3.138 `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?`.

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!

3.139 `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?`.

3.140 `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?`.

3.141 `dict?` : procedure/1

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

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

See also: `dict`.

3.142 `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`.

3.143 `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`.

3.144 `div` : procedure/2

Usage: `(div n k)=> int`

Integer division of `n` by `k`.

See also: `truncate`, `/`, **`int`**.

3.145 `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`.

3.146 `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`**.

3.147 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`.

3.148 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`.

3.149 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 prodedure 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`.

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

3.150 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`.

3.151 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`.

3.152 `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?`.

3.153 `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?`.

Warning: Deprecated.

3.154 `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?`.

3.155 `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`.

3.156 `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`.

3.157 `even?` : procedure/1

Usage: `(even? n)=> bool`

Returns true if the integer `n` is even, nil if it is not even.

See also: `odd?`.

3.158 `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?`.

3.159 `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: `n/a`.

3.160 `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`.

3.161 `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`.

3.162 `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`.

3.163 `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`.

3.164 `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`.

3.165 `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`.

3.166 `fdelete` : procedure/1

Usage: `(fdelete path)`

Removes the file or directory at `path`.

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

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

3.167 `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`.

3.168 `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`.

3.169 `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`.

3.170 `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`.

3.171 `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`.

3.172 `fl.abs` : procedure/1

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

Return the absolute value of `x`.

See also: **`float`**, `*`.

3.173 `fl.acos` : procedure/1

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

Return the arc cosine of `x`.

See also: `fl.cos`.

3.174 `fl.asin` : procedure/1

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

Return the arc sine of `x`.

See also: `fl.acos`.

3.175 `fl.asinh` : procedure/1

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

Return the inverse hyperbolic sine of `x`.

See also: `fl.cosh`.

3.176 `fl.atan` : procedure/1

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

Return the arctangent of `x` in radians.

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

3.177 `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`.

3.178 `fl.atanh` : procedure/1

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

Return the inverse hyperbolic tangent of `x`.

See also: `fl.atan`.

3.179 `fl.cbrt` : procedure/1

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

Return the cube root of `x`.

See also: `fl.sqrt`.

3.180 `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`.

3.181 `fl.cos` : procedure/1

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

Return the cosine of `x`.

See also: `fl.sin`.

3.182 `fl.cosh` : procedure/1

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

Return the hyperbolic cosine of `x`.

See also: `fl.cos`.

3.183 `fl.dim` : procedure/2

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

Return the maximum of x, y or 0.

See also: `max`.

3.184 `fl.erf` : procedure/1

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

Return the result of the error function of `x`.

See also: `fl.erfc`, `fl.dim`.

3.185 `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`.

3.186 `fl.erfcinv` : procedure/1

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

Return the inverse of (fl.erfc `x`).

See also: `fl.erfc`.

3.187 `fl.erfinv` : procedure/1

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

Return the inverse of (fl.erf `x`).

See also: `fl.erf`.

3.188 `fl.exp` : procedure/1

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

Return e^`x`, the base-e exponential of `x`.

See also: `fl.exp`.

3.189 `fl.exp2` : procedure/2

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

Return 2^`x`, the base-2 exponential of `x`.

See also: `fl.exp`.

3.190 `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`.

3.191 `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`**.

3.192 `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: `*`, `+`.

3.193 `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` == `frac` × 2^`exp` where the absolute value of `frac` is in the interval [0.5, 1).

See also: `fl.exp`.

3.194 `fl.gamma` : procedure/1

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

Compute the Gamma function of `x`.

See also: `fl.lgamma`.

3.195 `fl.hypot` : procedure/2

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

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

See also: `fl.sqrt`.

3.196 `fl.ilogb` : procedure/1

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

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

See also: `fl.exp2`.

3.197 `fl.inf` : procedure/1

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

Return positive 64 bit floating point infinity +INF if `x` >= 0 and negative 64 bit floating point finfinity -INF if `x` < 0.

See also: `fl.is-nan?`.

3.198 `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`.

3.199 `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`.

3.200 `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`.

3.201 `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`.

3.202 `fl.ldexp` : procedure/2

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

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

See also: `fl.frexp`.

3.203 `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`.

3.204 `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`.

3.205 `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`.

3.206 `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`.

3.207 `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`.

3.208 `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`.

3.209 `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`.

3.210 `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`.

3.211 `fl.mod` : procedure/2

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

Return the floating point remainder of `x` / `y`.

See also: `fl.remainder`.

3.212 `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`.

3.213 `fl.nan` : procedure/1

Usage: `(fl.nan)=> fl`

Return the IEEE 754 not-a-number value.

See also: `fl.is-nan?`, `fl.inf`.

3.214 `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`.

3.215 `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`.

3.216 `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`.

3.217 `fl.remainder` : procedure/2

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

Return the IEEE 754 floating-point remainder of `x` / `y`.

See also: `fl.mod`.

3.218 `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`**.

3.219 `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`**.

3.220 `fl.signbit` : procedure/1

Usage: `(fl.signbit x)=> bool`

Return true if `x` is negative, nil otherwise.

See also: `fl.abs`.

3.221 `fl.sin` : procedure/1

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

Return the sine of `x`.

See also: `fl.cos`.

3.222 `fl.sinh` : procedure/1

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

Return the hyperbolic sine of `x`.

See also: `fl.sin`.

3.223 `fl.sqrt` : procedure/1

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

Return the square root of `x`.

See also: `fl.pow`.

3.224 `fl.tan` : procedure/1

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

Return the tangent of `x` in radian.

See also: `fl.tanh`, `fl.sin`, `fl.cos`.

3.225 `fl.tanh` : procedure/1

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

Return the hyperbolic tangent of `x`.

See also: `fl.tan`, `flsinh`, `fl.cosh`.

3.226 `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`.

3.227 `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`.

3.228 `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`.

3.229 `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`.

3.230 `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`.

3.231 `float` : procedure/1

Usage: `(float n)=> float`

Convert `n` to a floating point value.

See also: `int`.

3.232 `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`.

3.233 `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?`.

3.234 `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`.

3.235 `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`.

3.236 `functional-arity` : procedure/1

Usage: `(functional-arity proc)=> int`

Return the arity of a functional `proc`.

See also: `functional?`, `functional-has-rest?`.

3.237 `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`.

3.238 `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?`.

3.239 `gensym` : procedure/0

Usage: `(gensym)=> sym`

Return a new symbol guaranteed to be unique during runtime.

See also: `nonce`.

3.240 `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`.

3.241 `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`.

3.242 `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 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`.

3.243 `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`.

3.244 `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!`.

3.245 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`.

3.246 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`.

3.247 help : macro/1

Usage: `(help sym)`

Display help information about `sym` (unquoted).

See also: `defhelp`, `help-entry`, `*help*`, `apropos`.

3.248 help->manual-entry : nil

Usage: `(help->manual-entry key [level])=> 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.

See also: `help`.

3.249 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`.

3.250 `help-entry` : procedure/1

Usage: `(help-entry sym)=> list`

Get usage and help information for `sym`.

See also: `defhelp`, `help`, `apropos`, `*help*`.

3.251 `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`.

3.252 `help-topics` : procedure/0

Usage: `(help-topics)=> li`

Obtain a list of help topics for commands.

See also: `help`, `help-topic`, `apropos`.

3.253 `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`.

3.254 `hook` : procedure/1

Usage: `(hook symbol)`

Lookup the internal hook number from a symbolic name.

See also: `*hooks*`, `add-hook`, `remove-hook`, `remove-hooks`.

3.255 `hour+` : procedure/2

Usage: `(hour+ dateli n)=> dateli`

Adds `n` hours to the given date `dateli` in datelist format and returns the new datelist.

See also: `sec+`, `minute+`, `day+`, `week+`, `month+`, `year+`, `now`.

3.256 `identity` : procedure/1

Usage: `(identity x)`

Return `x`.

See also: `apply`, `equal?`.

3.257 `if` : macro/3

Usage: `(if cond expr1 expr2)=> any`

Evaluate `expr1` if `cond` is true, otherwise evaluate `expr2`.

See also: `cond`, `when`, `unless`.

3.258 `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`.

3.259 `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`.

3.260 `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?`.

3.261 `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`.

3.262 `int` : procedure/1

Usage: `(int n)=> int`

Return `n` as an integer, rounding down to the nearest integer if necessary.

See also: **`float`**.

Warning: If the number is very large this may result in returning the maximum supported integer number rather than the number as integer.

3.263 `intern` : procedure/1

Usage: `(intern s)=> sym`

Create a new interned symbol based on string `s`.

See also: `gensym`, `str->sym`, `make-symbol`.

3.264 `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`.

Warning: This function currently only returns the binding but this behavior might change in future.

3.265 `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?`.

Warning: What counts as an intrinsic or not may change from version to version. This is for internal use only.

3.266 `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`.

3.267 `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`.

3.268 `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`.

3.269 `len` : procedure/1

Usage: `(len seq)=> int`

Return the length of `seq`. Works for lists, strings, arrays, and dicts.

See also: `seq?`.

3.270 `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`.

3.271 `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`.

3.272 `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`.

3.273 `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`.

3.274 `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`.

3.275 `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`.

3.276 `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`.

3.277 `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`.

3.278 `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`.

3.279 `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`.

3.280 `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`.

3.281 `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`.

3.282 `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`.

3.283 `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`.

3.284 `list` : procedure/0 or more

Usage: `(list [args] ...)=> li`

Create a list from all `args`. The arguments must be quoted.

See also: `cons`.

3.285 `list->array` : procedure/1

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

Convert the list `li` to an array.

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

3.286 `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`.

3.287 `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`.

3.288 `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?`.

3.289 `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?`.

3.290 `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`.

3.291 `list-last` : procedure/1

Usage: `(list-last li)=> any`

Return the last element of `li`.

See also: `reverse`, `nreverse`, `car`, `1st`, `last`.

3.292 `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`.

3.293 `list-reverse` : procedure/1

Usage: `(list-reverse li)=> li`

Create a reversed copy of `li`.

See also: `reverse`, `array-reverse`, `str-reverse`.

3.294 `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`.

3.295 `list?` : procedure/1

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

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

See also: `cons?`, `atom?`, **`null`**`?`.

3.296 `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?`.

3.297 `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?`.

3.298 `make-mutex` : procedure/1

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

Create a new mutex.

See also: `mutex-lock`, `mutex-unlock`, `mutex-rlock`, `mutex-runlock`.

3.299 `make-queue` : procedure/0

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

Make a synchronized queue.

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

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

3.300 `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?`.

3.301 `make-stack` : procedure/0

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

Make a synchronized stack.

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

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

3.302 `make-symbol` : procedure/1

Usage: `(make-symbol s)=> sym`

Create a new symbol based on string `s`.

See also: `str->sym`.

3.303 `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`.

3.304 `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`.

3.305 `mapcar` : procedure/2

Usage: `(mapcar li proc)=> li`

Return the list obtained from applying `proc` to each elements in `li`.

See also: `map`, `foreach`.

3.306 `max` : procedure/1 or more

Usage: `(max x1 x2 ...)=> num`

Return the maximum of the given numbers.

See also: `min`, `minmax`.

3.307 `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?`.

3.308 `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?`.

3.309 `memstats` : procedure/0

Usage: `(memstats)=> dict`

Return a dict with detailed memory statistics for the system.

See also: `collect-garbage`.

3.310 `min` : procedure/1 or more

Usage: `(min x1 x2 ...)=> num`

Return the minimum of the given numbers.

See also: `max`, `minmax`.

3.311 `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`.

3.312 `minute+` : procedure/2

Usage: `(minute+ dateli n)=> dateli`

Adds `n` minutes to the given date `dateli` in datelist format and returns the new datelist.

See also: `sec+`, `hour+`, `day+`, `week+`, `month+`, `year+`, `now`.

3.313 `mod` : procedure/2

Usage: `(mod x y)=> num`

Compute `x` modulo `y`.

See also: `%`, `/`.

3.314 `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`.

3.315 `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`.

3.316 `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`.

3.317 `mutex-runlock` : procedure/1

Usage: `(mutex-runlock m)`

Unlock the mutex `m` from reading.

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

3.318 `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`.

3.319 `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`.

3.320 `nl` : procedure/0

Usage: `(nl)`

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

See also: `out`, `outy`, `output-at`.

3.321 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`.

3.322 not : procedure/1

Usage: `(not x)=> bool`

Return true if `x` is nil, nil otherwise.

See also: `and`, `or`.

3.323 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`.

3.324 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`.

3.325 now-ns : procedure/0

Usage: `(now-ns)=> int`

Return the current time in Unix nanoseconds.

See also: `now`, `time`.

3.326 `nreverse` : procedure/1

Usage: `(nreverse li)=> li`

Destructively reverse `li`.

See also: `reverse`.

3.327 `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`.

3.328 `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`.

3.329 `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`.

3.330 `null?` : procedure/1

Usage: `(null? li)=> bool`

Return true if `li` is nil, nil otherwise.

See also: `not`, `list?`, `cons?`.

3.331 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?`.

3.332 odd? : procedure/1

Usage: `(odd? n)=> bool`

Returns true if the integer `n` is odd, nil otherwise.

See also: `even?`.

3.333 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*`.

3.334 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`.

3.335 `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`.

3.336 `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`.

3.337 `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`.

3.338 `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`.

3.339 `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`.

3.340 `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`.

3.341 `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`.

3.342 `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`.

3.343 `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`.

3.344 `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`.

3.345 `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`.

3.346 `prin1` : procedure/1

Usage: `(prin1 s)`

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

See also: `princ`, `terpri`, `out`, `outy`.

3.347 `princ` : procedure/1

Usage: `(princ s)`

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

See also: `prin1`, `terpri`, `out`, `outy`.

3.348 `print` : procedure/1

Usage: `(print x)`

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

See also: `prin1`, `princ`.

3.349 `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?`.

3.350 `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`.

3.351 `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`, `when-permission?`, `dict-protect`, `dict-`**`protected?`**, `dict-unprotect`.

3.352 `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`.

3.353 `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`.

3.354 `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`.

3.355 `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`.

3.356 `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`.

3.357 `pushstacked` : procedure/3

Usage: `(pushstacked dict key datum)`

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

See also: `getstacked`, `popstacked`.

3.358 `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`.

3.359 `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`.

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!

3.360 `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`.

3.361 `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`.

3.362 `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`.

3.363 `read` : procedure/1

Usage: `(read p)=> any`

Read an expression from input port `p`.

See also: `input`, `write`.

3.364 `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`.

3.365 `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`.

3.366 `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`.

3.367 `remove-hook` : procedure/2

Usage: `(remove-hook hook id)=> bool`

Remove the symbolic or numberic `hook` with `id` and return true if the hook was removed, nil otherwise.

See also: `add-hook`, `remove-hooks`, `replace-hook`.

3.368 `remove-hook-internal` : procedure/2

Usage: `(remove-hook-internal hook id)`

Remove the hook with ID `id` from numeric `hook`.

See also: `remove-hook`.

Warning: Internal use only.

3.369 `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`.

3.370 `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`.

3.371 `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`.

3.372 `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`.

3.373 `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`.

3.374 `rplaca` : procedure/2

Usage: `(rplaca li a)=> li`

Destructively mutate `li` such that its car is `a`, return the list afterwards.

See also: `rplacd`.

3.375 `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`.

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

3.376 `run-hook` : procedure/1

Usage: `(run-hook hook)`

Manually run the hook, executing all procedures for the hook.

See also: `add-hook`, `remove-hook`.

3.377 `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`.

Warning: Internal use only.

3.378 `run-selftest` : procedure/1 or more

Usage: `(run-selftest [silent?])=> any`

Run a diagnostic self-test of the Z3S5 Machine. If `silent?` is true, then the self-test returns a list containing a boolean for success, the number of tests performed, the number of successes, the number of errors, and the number of failures. If `silent?` is not provided or nil, then the test progress and results are displayed. An error indicates a problem with the testing, whereas a failure means that an expected value was not returned.

See also: `expect`, `testing`.

3.379 `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`.

3.380 `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`.

3.381 `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`.

3.382 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`.

3.383 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`.

3.384 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`.

3.385 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`.

3.386 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`.

3.387 `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`.

3.388 `seq?` : procedure/1

Usage: `(seq? seq)=> bool`

Return true if `seq` is a sequence, nil otherwise.

See also: `list`, `array`, `string`, `slice`, `nth`.

3.389 `set` : procedure/3

Usage: `(set d key value)`

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

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

3.390 `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`.

3.391 `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`.

3.392 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, the-color, with-colors.

3.393 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?.

3.394 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?.

3.395 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?.

3.396 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?.

3.397 `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?`.

3.398 `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`.

3.399 `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?`.

3.400 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`.

3.401 `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?`.

3.402 `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?`.

3.403 `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: `play-sound`, `play-music`.

3.404 `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?`.

3.405 `setcar` : procedure/1

Usage: `(setcar li elem)=> li`

Mutate `li` such that its car is `elem`. Same as rplaca.

See also: `rplaca`, `rplacd`, `setcdr`.

3.406 `setcdr` : procedure/1

Usage: `(setcdr li1 li2)=> li`

Mutate `li1` such that its cdr is `li2`. Same as rplacd.

See also: `rplacd`, `rplaca`, `setcar`.

3.407 `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`.

3.408 `sleep` : procedure/1

Usage: `(sleep ms)`

Halt the current task execution for `ms` milliseconds.

See also: `sleep-ns`, `time`, `now`, `now-ns`.

3.409 `sleep-ns` : procedure/1

Usage: `(sleep-ns n`

Halt the current task execution for `n` nanoseconds.

See also: `sleep`, `time`, `now`, `now-ns`.

3.410 `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?`.

3.411 `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 otheriwse.

See also: `array-sort`.

3.412 sort-symbols : nil

Usage: `(sort-symbols li)=> list`

Sort the list of symbols `li` alphabetically.

See also: `out`, `dp`, `du`, `dump`.

3.413 spaces : procedure/1

Usage: `(spaces n)=> str`

Create a string consisting of `n` spaces.

See also: `strbuild`, `strleft`, `strright`.

3.414 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`.

3.415 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`.

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!

3.416 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`.

3.417 `str+` : procedure/0 or more

Usage: `(str+ [s] ...)=> str`

Append all strings given to the function.

See also: `str?`.

3.418 `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`.

3.419 `str->blob` : procedure/1

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

Convert string `s` into a blob.

See also: `blob->str`.

3.420 `str->char` : procedure/1

Usage: `(str->char s)`

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

See also: `char->str`.

3.421 `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`.

3.422 `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`.

3.423 `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 expession parser works.

See also: `str->expr`, `expr->str`, `openstr`, `internalize`, `externalize`.

3.424 `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`.

3.425 `str->sym` : procedure/1

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

Convert a string into a symbol.

See also: `sym->str`, `intern`, `make-symbol`.

3.426 `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`.

3.427 `str-empty?` : procedure/1

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

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

See also: `strlen`.

3.428 `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?`.

3.429 `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?`.

3.430 `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`.

3.431 `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 macthing char is found, nil is returned.

See also: `strsplit`, `chars`, `inchars`.

3.432 `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`.

3.433 `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`.

3.434 `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`.

3.435 `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`.

3.436 `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`.

3.437 `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`.

3.438 `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`.

3.439 `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`.

3.440 `str-reverse` : procedure/1

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

Reverse string `s`.

See also: `reverse`, `array-reverse`, `list-reverse`.

3.441 `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`.

3.442 `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`.

3.443 `str?` : procedure/1

Usage: `(str? s)=> bool`

Return true if `s` is a string, nil otherwise.

See also: `num?`, `atom?`, `sym?`, `closure?`, `intrinsic?`, `macro?`.

3.444 `strbuild` : procedure/2

Usage: `(strbuild s n)=> str`

Build a string by repeating string `s`` `` n` times.

See also: `str+`.

3.445 `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`.

3.446 `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`.

3.447 `strcnt` : procedure/2

Usage: `(strcnt s del)=> int`

Returnt the number of non-overlapping substrings `del` in `s`.

See also: `strsplit`, `str-index`.

3.448 `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`.

3.449 `strlen` : procedure/1

Usage: `(strlen s)=> int`

Return the length of `s`.

See also: `len`, `seq?`, `str?`.

3.450 `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`.

3.451 `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`.

3.452 `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`.

3.453 `stropen` : procedure/1

Usage: `(stropen s)=> streamport`

Open the string `s` as input stream.

See also: `open`, `close`.

3.454 `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`.

3.455 `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?`.

3.456 `sub1` : procedure/1

Usage: `(sub1 n)=> num`

Subtract 1 from `n`.

See also: `add1`, `+`, `-`.

3.457 `sym->str` : procedure/1

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

Convert a symbol into a string.

See also: `str->sym`, `intern`, `make-symbol`.

3.458 `sym?` : procedure/1

Usage: `(sym? sym)=> bool`

Return true if `sym` is a symbol, nil otherwise.

See also: `str?`, `atom?`.

3.459 `synout` : procedure/1

Usage: `(synout arg)`

Like out, but enforcing a new input line afterwards. This needs to be used when outputing concurrently in a future or task.

See also: `out`, `outy`, `synouty`.

Warning: Concurrent display output can lead to unexpected visual results and ought to be avoided.

3.460 `synouty` : procedure/1

Usage: `(synouty li)`

Like outy, but enforcing a new input line afterwards. This needs to be used when outputing concurrently in a future or task.

See also: `synout`, `out`, `outy`.

Warning: Concurrent display output can lead to unexpected visual results and ought to be avoided.

3.461 `sys-key?` : procedure/1

Usage: `(sys-key? key)=> bool`

Return true if the given sys key `key` exists, nil otherwise.

See also: `sys`, `setsys`.

3.462 `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`.

3.463 `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`.

3.464 `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?`.

3.465 `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`.

3.466 `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`.

3.467 `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`.

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

3.468 `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`.

3.469 `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-`.

3.470 `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`.

3.471 `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`.

3.472 `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`.

3.473 `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`.

3.474 `terpri` : procedure/0

Usage: `(terpri)`

Advance the host OS terminal to the next line.

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

3.475 `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`.

3.476 `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`.

3.477 `the-color-names` : procedure/0

Usage: `(the-color-names)=> li`

Return the list of color names in *colors*.

See also: `*colors*`, `the-color`.

3.478 `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`.

3.479 `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`.

3.480 `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`.

3.481 `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`.

3.482 `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`.

3.483 `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: `gfx.reset`.

3.484 `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`**?.

3.485 `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`*.

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

3.486 `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`*.

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

3.487 `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*`.

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

3.488 `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*`.

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

3.489 `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`.

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

3.490 `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`.

3.491 `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`.

3.492 `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`.

3.493 `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`.

3.494 `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?`.

3.495 `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`.

3.496 `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`.

3.497 `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`.

3.498 `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`.

3.499 `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`.

Warning: Make sure to never lock the same mutex twice from the same task, otherwise a deadlock will occur!

3.500 `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`.

3.501 `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`.

3.502 `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`.

3.503 `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`.

3.504 `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`.

3.505 `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`.