

Common Lisp

©1997-2016 Mitch Richling <https://www.mitchr.me>

[mitchr.me](https://www.mitchr.me)

Last Updated 2016-07-09

Control: Code Blocks

- (prog[1|2|n] *form1* ... *formn*) ;; Evaluate forms left to right.
- ; Returns *form1*, *form2*, or *formn*
- (block *symp form1*...) ;; A progn + name & emergency exit (return-from)
- (return-from *symp [value]*) -- Break out of a block *symp* & return *value*
- (return [*value*]) ;; == (return-from nil [*value*]) (think: 'do' blocks)
- (tagbody ...) ;; Atoms within a tagbody are LABELS that one may
;; use 'go', as in (go 'foo), to jump to. Many loop
;; constructs implicitly use tagbody so go may be used.
- (error *symp*) ;; Print message & Break to top level .
- (eval *form*) ;; Evaluates *form* as a lisp expression
- Short-cut, left-right, logical functions: and, or, not

Control: Conditionals

- (if *test body-true [body-false]*)
- (when *test form-true*)
- (unless *test form-false*)
- (cond (*test1 body1*) ... ; The first *body* with a true *test*
(*test2 body2*)...) ; clause is evaluated.

Control: Iteration (do)

- (dotimes (*symp n [form-ret]*) *body*) ;; Do *body* *n* times.
== (loop for *symp* from 0 to (- *n* 1) finally return *form-ret* do *body*)
- (dolist (*symp list [form-ret]*) *body*)
== (loop for *symp* in *list* finally return *form-ret* do *body*)
- (do ((*symp1 value1 [form-incr]*)... (test [*value-exit*]) *body*)
;; Provides block and tagbody. do* to assigns/updates vars in order

Control: Iteration (loop)

- (loop *form1*...) ;; If no KWs in *form1*..., then loop forever
- (loop [named *symp*] {with/initially/finally/for}... *body*...)
- KW Subs: • upfrom/downfrom ==> from
• downto/upto ==> to
• hash-key ==> hash-keys • hash-value ==> hash-values
• the ==> each
- KW Control Clauses
• for *symp* upfrom *value1* [{upto | below} *value2*] [by *value3*]
• for *symp* downfrom *value1* [{downto | above} *value2*] [by *value3*]
• for *symp* in *list* [by *func*] ;; Over list elements
• for *symp* on *list* [by *func*] ;; Over list TAILS
• for *symp* = *value1* [then *value2*]...
• for *symp* across *vector* or *string*
• for *symp* being the hash-keys of *hash* [using (hash-values *value*)]
• for *symp* being the hash-values of *hash* [using (hash-keys *value*)]
• initially *form*... ;; Evaluate as prologue
• finally *form*... ;; Evaluate as part of epilogue
• return *form* ;; return value. Skip epilogue
• { if | when | unless } *form1* [else *form2*] [end] ;; conditional
• { collect[ing] | append[ing] | nconc[ing] | count[ing] |
sum[ing] | maximize[ing] | minimize[ing] } *form* [into *symp*]
• repeat *n* ;; Iteration stops after *n* loops
• while *bool* ;; Iteration stops when *bool* is nil
• until *bool* ;; Iteration stops when *bool* is not nil
• (loop-finish) ;; Causes a jump to the loop epilogue
• (return-from [*symp [value]*]) ;; Return from loop
• (return) ;; Return from loop
• Destructured binding examples
• (loop for (a b) in '(x 1) (y 2)) collect (list a) ==>> (X Y)
• (loop for (x . y) in '((1 . 2) (3 . 4)) collect y) ==>> (2 4)

Pair Construction & Access

- Type Predicate: consp
- (cons *form1 form2*) ;; Can use '(*form1 . form2*) too
- (car *pair*) ;; left part. Settable!
- (cdr *pair*) ;; right part. Settable!
- (rplaca *pair form*) ;; Set (car *pair*) to *form*. Destructive
- (rplacd *pair form*) ;; Set (cdr *pair*) to *form*. Destructive

Lists

- Type Predicate: listp, null (T if nil, else nil)
- (list *form1 form2 form3* ...)
- (make-list *n* &K:le) ;; Create list (initialize using KW)
- nil <=> () ;; Empty list.
- (append *list* ...)
- (nconc *list1 list2*) ;; Destructively add *list2* to *list1*
- (nreconc *list1 list2*) ;; Same as (nconc (nreverse x) y)
- (cons *form list*) ;; Prepend *list* with *form*
- Named elements: first, second, third, fourth, fifth, ..., tenth
- (car *list*) ;; first element
- (cdr *list*) ;; all but first element
- Compositions of car & cdr have names, (cadr *list*)<=>(car (cad *list*)).
- Forms exist up to four compositions (Perl regex: m/^c[ad]{2,4}r\$/).
- (rest *list*) ;; all but first element
- (last *list* [*n*]) ;; Last cons (or *n* to last)
- (nth 5 *list*) ;; get nth element (zero indexed)
- (nthcdr *n list*) ;; get nth cdr (nth element on) (zero indexed)
- (mapcar *func list1*...) ;; Apply *func* the lists in parallel
- (mapc *func list1*) ;; Like mapcar, but returns *list*
- (copy-list *list*) ;; Create a copy of *list*
- (copy-tree *list*) ;; Recursively copy *list* and its sublists
- (subst *value1 value2 list* &K:TTnK) ;; Recessive version of substitute
- (sublis *list list* &K:TTnK) ;; Recursively replace all keys with values
- (tree-equal *list1 list2* &K:TTn)
- (list-length *list*) ;; Length of list. Works with circular lists.
- (butlast *list* [*n*]) ;; List except last *n* elements.
- (member *value list* &K:TTnK) ;; Returns from first match on to end
- (adjoin *form list* &K:TTnK) ;; Add to *list* unless *form* is in *list*
- (subsetp *list1 list2* &K:TTnK) ;; T if every ele of *list1* in *list2*
- Set Operators: union, intersection, set-difference, set-exclusive-or
- Alternate, DESTRUCTIVE, forms:
 - nsubst • nbutlast • mintersection • nset-exclusive-or
 - nsublis • nunion • nset-difference
- Alternate -if, -if-not forms:
 - (nsubst-if *pred list* &K:K) • (nsubst-if-not *pred list* &K:K)
 - (subst-if *pred list* &K:K) • (subst-if-not *pred list* &K:K)
 - (member-if *pred list* &K:K) • (member-if-not *pred list* &K:K)

Types

- atomp, symbolp
- (typep *form symp*) ;; t if *form* os of type *symp*
- (type-of *form*) ;; Return the type of *form*

Numbers

- Type Tree (p means a type predicate exists)
numberp +- realp +- floatp ----- short-float
| +- rationalp +- ratio +- single-float
| +- integerp +- bignum +- double-float
+ complex +- fixnum -- bit +- long-float
- Number Classes: evenp, oddp, zerop, plusp, minusp
- Conversion: float, truncate, floor, ceiling, rationalize, rational, (complex *value1* [*value2*])
- Parts: numerator, denominator (always positive), realpart, imagpart
- Comparison & Arithmetic: =, >, <, <=, >=, *, /, +, -
- Special Syntax: • Rational: *value1/value2*
 - Float: m.Xn (m, n integers).
 - X=s for short-float • X=f for single-float
 - X=d for double-float • X=l for long-float
- (random *value*) ;; Random number less than *value* and of the same numeric type

Notation

- #0 octal rat #0777/2
- #C complex #C(1 2)
- #B binary rat #B101/11 <=> 5/3
- #nA array #2A((1 2) (3 4))
- #S structure #S(pnt x 10 y 23)
- #n(Simple Vec #4n(1 2 3 4)
- #nR Base n Rat #3R1021
- #' Function #'+
- #(simple vec #(1 2 3)
- ## bit vec ##*101001
- #\ char #\a
- #X hex rational #Xf00
- ##* simple bit-vec ##6*101001
- #|...#1 Comment

Traditional Mathematical Functions

- sqrt • sin • gcd • asin • sinh • asinh • round • conjugate
- mod • tan • lcm • atan • tanh • atanh • realpart • ceiling
- min • cos • abs • acos • cosh • acosh • imagpart
- max • exp • log • isqrt • expt • floor • signum

Equality

- equal objects logically the same • eq same address
- equalp Liberal equal (ignores case...) • = works with numbers
- eql equal for same numeric type, else eq

Bit Vectors (0's & 1's)

- Make a bit-vector: (make-array *n* :element-type 'bit :initial-element 0)
- Type Predicate: bit-vector-p, simple-bit-vector-p
- (bit *bit-vector n*) ;; like aref, just for bit-vectors
- (sbit *bit-vector n*) ;; like svref, just for bit-vectors
- Bit operations: bit-eqv, bit-xor, bit-nand, bit-and, bit-not, bit-nor

Path & File Names

- Type Predicate: pathnamep
- (make-pathname ...) ;; Create a pathname object. KW parms:
 - :directory • :name • :host • :device • :type • :version
- Path to string: file-namestring, directory-namestring, namestring
- Component access: pathname-directory, pathname-name *path*

File System

- (delete-file *path*) ;; Delete the file given by *path*
- (directory *path*) ;; list of files in *path*
- (ensure-directories-exist *path*) ;; Create every directory on *path*
- (file-write-date *path*) ;; last modify time for file
- (probe-file *path*) ;; nil if file dose not exist
- (rename-file *path1 path2*) ;; rename *path1* to *path2*
- (truename *path*) ;; real name of file at *path*

Streams

- Type/State Predicates: stream-p, input-stream-p, interactive-stream-p, open-stream-p, output-stream-p
- (open path) ; Returns a Stream. Useful KW args:
 - :direction [:input | :output | :io]
 - :if-exists [:error | :overwrite | :append | :supersede]
 - :element-type ['base-character | 'character | 'unsigned-byte]
- (file-length stream)
- (file-position stream [n]) ; queries or sets file pointer
- (finish-output [stream])
- (clear-input [stream]) ; throw away any waiting input
- (close stream)

I/O

- (with-open-file (symb stream [open-args]) body)
- (with-open-file (symb string) body) ; Not portable, but handy
- (with-open-stream (symb stream) body)
- (read [stream] [bool-err-on-EOF] [value-ret-on-EOF]) ; read LISP
- (with-output-to-string (symb [string]) body)
 - ; printed string is returned if string not given
- (read-line [stream] [bool-err-on-EOF] [value-ret-on-EOF])
- (read-from-string string [bool-err-on-EOF] [value-ret-on-EOF])
- (read-char [stream] [bool-err-on-EOF] [value-ret-on-EOF])
- (read-byte [stream] [bool-err-on-EOF] [value-ret-on-EOF]) ; return int
- (write-byte n [stream])
- (peek-char [bool] [stream] [bool-err-on-EOF] [value-ret-on-EOF])
- (fresh-line [stream]) ; write newline if not at start of line
- (terpri) ; Move to newline
- (print form [stream]) ; LISP like
- (prinl form [stream]) ; No NL
- (princ form [stream]) ; Human like
- Print to strings: princ-to-string, prin-to-string
- (dribble [string]) ; print session to file. Stop if no argument.
- (load string) ; load named file and evaluate lisp

Format

- (format *dst string-fmt form1...*)
 - ; value-dst may be T (for STDOUT), NIL (for a string), or a stream
 - ~r,~R Base r int • ~wA Like princ (@ right justifies)
 - ~d Decimal int • ~wS Like princ (@ right justifies)
 - ~B Binary int • ~wL Like write (@ right justifies)
 - ~O Octal int • ~cC Character
 - ~x Hex int • ~n% n newlines
 - ~w,d,sF Float • ~n& n smart newlines
 - ~w,d,e,sE Exp Float • ~nT Move to Col n
 - ~w,d,e,sG do F or E
 - d=digits before dec, s=digits after dec, e=exp digits, w=width
 - R,D,B,O,X Mods: '~@' prints + signs & '~:' prints commas

Arrays

- Type Predicate: array
- (make-array '(dim1...) &key :l e :adjustable :initial-contents)
- (adjust-array array new-dim &key ...)
- (aref array int1...) ; Array element access. Zero-indexed. Settable.
- (array-dimension array n) ; Length of n-th dim. Zero-indexed
- (array-dimensions array) ; List of ints representing dimensions.
- (array-element-type array)
- (array-rank array) ; Returns the number of dimensions
- (array-total-size array) ; Returns number of locations in array.

Vectors

NOTE: VECTORS ARE 1D ARRAYS -- SO ALL ARRAY FUNCTIONS WORK.

- Type Predicates: vectorp, simple-vector-p
- (vector form1...) ; Create new vector from form1...
- (svref vector n) ; Just like aref, but faster for SIMPLE VECTORS
- (setf (aref vector n) form) ; Can setf an aref like this

Characters

- Type Predicate: characterp
- (character n) or (character char)
- (char-code char) ; Return numeric code for character
- (char-name char) ; Return string for char
- (code-char n) ; Return char for code
- Character Transformation: char-upcase, char-downcase
- Binary Predicates: char<, char>, char<=, char>=, char>=, char/=, char-not-greaterp, char-equal, char-lessp, char-not-lessp, char-greaterp, char-not-equal
- Class Predicates: digit-char-p, alpha-char-p, graphic-char-p, lower-case-b, upper-case-p, alphanumericp, standard-char-p

Strings

NOTE: STRINGS ARE VECTORS OF CHARACTERS.

- "I am a string" ; Syntax for a string literal
- Type Predicate: stringp, simple-string-p
- (string form) ; Convert symbols/characters/strings to strings
- (char string n) ; same as (aref string n)
- (schar string n) ; same as svref (simple strings)
- (substring string value1 value2) ; Same as subseq
- (make-string size &key :l e :element-type) ; Same as make-array
- (string-width string) ; same as length
- (string-concat string1 string2...) ; specialized as concatenate
- String Transformations: string-capitalized, string-downcase, string-left-trim, string-right-trim, string-trim, string-upcase
- PREFIX 'N' TO TRANSFORMATIONS TO GET A DESTRUCTIVE VERSION
- CASE TRANSFORMATIONS TAKE KEYWORD PARAMETERS: &K:SE
- Binary Predicates: string-lessp, string/=, string-not-equal, string<, string-not-greaterp, string<=, string-not-lessp, string=, string>, string-equal, string>=, string-greaterp
- ALL BINARY PREDICATES TAKE KEYWORD PARAMETERS: &K:S1E1S2E2

Structures

- (destructure symb symb1...)
 - Define a structure named symb with members symbN
 - This will create several functions/macros including:
 - make-symb • symb-p
 - copy-symb • symb-symbN for all N
 - Instance: #S(symb value1...)

Associative Lists

- (assoc form-key list &K:TTnK) ; find pair with given key
- (rassoc form-value list &K:TTnK) ; find pair with given value
- (acons form-key value-form list) ; Add pair to list
- (copy-alist list) ; Make a copy of list.
- (pairlis list-keys list-values) ; Build a-list from parts.
- Alternate -if, -if-not forms:
 - (assoc-if pred list &K:K) • (assoc-if-not pred list &K:K)
 - (rassoc-if pred list &K:K) • (rassoc-if-not pred list &K:K)
- Examples
 - (assoc "a" '((("a" . 1) ("b" . 2))) :test #'string=) ==> ("a" . 1)
 - (assoc :a '((:a . 1) (:b . 2))) ==> (:A . 1)

Hash Tables

- Type Predicate: hash-table-p
- (clrhash hash)
- (hash-table-count hash) ; Number of entries
- (hash-table-size hash) ; Size of hash table
- (maphash func hash) ; Apply func to each entry in hash
- (make-hash-table [:size n] [:text func]) ; Create hash table
- (gethash symb hash) ; Returns object or nil. Settable.
- (rmhash symb hash) ; Remove symb from hash
- (with-hash-table-iterator (symb hash) body...)

Integer Bit & Byte Manipulation

- (byte value-size value-position) ; Create a bytespec
- Byte Spec component access: byte-size, byte-position
- (ldb bytespec n) ; Extract part of integer and shift
- (ldb-test bytespec n) ; Are any of the bits 1
- (mask-field bytespec n) ; Extract part and leave it in place
- (dpb bytespec1 bytespec2 n)
- (deposit-field bytespec1 bytespec2 n) ; bytespec1 to bytespec2
- (logcount int1) ; Returns the number of '1' bits in int1
- Logical, bitwise, operations on integers
 - logxor • lognand • lognor • logior (inclusive or)
 - logand • logandc2 • logorc2 • logeqv (exclusive nor)
 - logandc1 • logorc1 • lognot
 - logtest ; t if (and int1 int2) not zero
 - (logbitp int1 int2) t if bit int1 of int2 is 1
 - (ash int1 int2) ; Shift int1 left int2 bits (int2<0 is OK)
 - (boole op int1 int2) ; Any of the 16 boolean, binary ops
 - Op must be one of (all names prefixed with "boole"):
 - a 0 0 1 1 a 0 0 1 1 a 0 0 1 1 a 0 0 1 1
 - b 0 1 0 1 b 0 1 0 1 b 0 1 0 1 b 0 1 0 1
 - clr 0 0 0 0 -xor 0 1 1 0 -c1 1 1 0 0 -andc1 0 1 0 0
 - set 1 1 1 1 -eqv 1 0 0 1 -c2 1 0 1 0 -andc2 0 0 1 0
 - 1 0 0 1 1 -nand 1 1 1 0 -and 0 0 0 1 -orc1 1 1 0 1
 - 2 0 1 0 1 -nor 1 0 0 0 -ior 0 1 1 1 -orc2 1 0 1 1

Variables

- (let ((symb1 value1)...) body...) ; Declare local variables
- (let* ((symb1 value1)...) body...) ; Declare local variables (in order)
- (defparameter symb value [string]) ; Declare global variable
- (defvar symb [value [string]]) ; Declare global variable
- (defconstant symb value [string]) ; Declare global constant
- (defun name list-lambda [string-doc] body...) ; Declare global function
 - ; Add (interactive) before body... for EMACS interactive function
- (defun (setf name) list-lambda body...) ; Define setf behavior for name
 - ; arg-val is the new value given to setf.
- (defsetf
- (setf symb value) ; Set variables (speical, global, local, ...)
- (incf symb [symb1]) ; Same as (setf symb (+ symb symb1))
- (decf symb [symb1]) ; Same as (setf symb (- symb symb1))
- (push value symb) ; Same as (setf symb (cons value symb))
- (pushnew value symb &K:TTnK) ; push only if value no in symb already
- (pop symb) ; Returns (car symb) & sets symb to (cdr symb)
- (boundp symb) ; t if symb is bound to a non-function
- (fboundp symb) ; t if symb is bound to a function

Functions

- Type Predicates: compiled-function-p, functionp
- (function symb) ; Returns the function bound to symb
- (lambda (list-lambda) body...) ; Define function
 - The list-lambda is of the form:
 - symb ... ; Arg List
 - [&optional symb1 [value1] ...] ; Optional args
 - [&rest symb] ; Rest of args
 - [&key symb1 [value1] ...] ; Key-value args
- (funcall name arg1...) ; like apply, but last arg need not be list
- (apply name arg1 ...list) ; Apply function with arguments in list; list: append(arg1... list). Much like funcall
 - ; See maplist to apply a function to each element of a list
 - ; See reduce to apply function recursively to list
- (values [nArg1...]) ; Return zero or more values
- (values-list list) ; Like values, but returns list elements
- (multiple-value-list body) ; Evaluates body and returns a LIST of returns from body
- (multiple-value-bind (symb1...) body body1...) ; Eval body, bind returns, eval rest
- (multiple-value-setq (symb1) body) ; Eval body, and set variables.
- (compile symb) ; Compile a function

Sequences

NOTE: SEQUENCES INCLUDE LISTS & VECTORS (AND THUS STRINGS TOO)

- (make-sequence aType size &K:Ie)
- (concatenate aType seq1...) ;; Concatenates given sequences
- (count form seq &K:FeTtnSEK) ;; Count elements in seq matching form
- (copy-seq seq)
- (elt seq n) ;; Return the n element of seq
- (fill seq value &K:SE) ;; Fill seq with value
- (find value seq &K:FeTtnSEK) ;; Returns value if found
- (length seq)
- (map aType func seq) ;; Like mapc but for sequences
- (map-into seq func seq1) ;; destructive map. Result into seq
- (mismatch seq1 seq2 &K:KFeTtnKS1S2E1E2) ;; Return position of first mismatch
- (position value seq &K:FeTtnSEK) ;; Returns zero based index of value in seq, else nil.
- (reduce func seq &K:FeSEIv) ;; recursively apply binary function func
;; to elements of seq returning one atomic value.
- (remove value seq &K:FeTtnSECK) ;; Remove all occurrences of value from seq
- (reverse seq)
- (merge aType seq1 seq2 pred &K:K) ;; Destructively merge with sorting predicate pred
- (sort seq pred &K:K) ;; WARNING: DESTRUCTIVE!! (pred - binary comparison)
- (subseq seq value-start [value-end])
- (substitute value1 value2 seq &K:FeTtnSECK) ;; Replace value1 for value2 in seq
- (every func seq1...) ;; Apply func like mapcar, return T if func was never nil
- (notany func seq1...) ;; Similar to every, but different :)
- (notevery func seq1...) ;; Similar to every, but different :)
- (some func seq1...) ;; Similar to every, but different :)
- (search seq1 seq2 &K:FeTtnKS1S2E1E2) ;; Find seq1 in seq2. Return index.
- (remove-duplicates seq &K:FeTtnSEK) ;; Remove duplicate objects from seq
- Alternate, -if and -if-not forms:
 - (count-if pred seq &K:FeSEK) • (count-if-not pred seq &K:FeSEK)
 - (find-if pred seq &K:FeSEK) • (find-if-not pred seq &K:FeSEK)
 - (position-if pred seq &K:FeSEK) • (position-if-not pred seq &K:FeSEK)
 - (remove-if pred seq &K:FeSECK) • (remove-if-not pred seq &K:FeSECK)
 - (delete-if pred seq &K:FeSECK) • (delete-if-not pred seq &K:FeSECK)
 - (substitute-if value1 pred seq) • (substitute-if-not value1 pred seq &K:FeSECK)
- Alternate, DESTRUCTIVE, forms:
 - nreverse • nsubstitute-if • delete-duplicates (see: remove-duplicates)
 - nsubstitute • nsubstitute-if-not • delete (see: remove)

Keyword Argument Key

- :key :K Function used before :test • :from-end :Fe Work in reverse
- :test :T Test to use for comparison • :start :S Where to start working
- :end :E Where to stop working • :test-not :Tn Test to use for comparison
- :end1 :E1 Where to stop working Arg1 • :start1 :S1 Where to start working Arg1
- :end2 :E2 Where to stop working Arg2 • :start2 :S2 Where to start working Arg2
- :count :C How many times/elements
- :initial-element :Ie Initializeing element for various make-* functions
- :initial-value :Iv Initializeing value for an accumulator

IN THE LISTINGS, &K: INDICATES THAT THE &KEY ARGUMENT LIST IS COMPLETELY ABBREVIATED. FOR EXAMPLE:

```
(foo &K:TtnK) <=> (foo &key :test :test-not :key)
```

A KEY ARGUMENT THAT IS IN UPPER CASE AND CONSISTS OF JUSTOPOSED ABBREVIATIONS FROM ABOVE, SHOULD BE ASSUMED TO BE ABBREVIATIONS. FOR EXAMPLE:

```
(foo &key :bar :TR) <=> (foo &key :bar :test :key)
```