Common Lisp Quick Reference

Contents

1 Numbers 3
1.1 Predicates . . . . 3
1.2 Numeric Functions . 3
1.3 Logic Functions . 5
1.4 Integer Functions . 6
1.5 Implementation-Dependent . . . . 6

2 Characters 7
3 Strings 8
4 Conses 8
4.1 Predicates . . . . 8
4.2 Lists . 9
4.3 Association Lists . 10
4.4 Trees . 10
4.5 Sets . . . . . . 11

5 Arrays 11
5.1 Predicates . . . . 11
5.2 Array Functions . 11
5.3 Vector Functions . 12

6 Sequences 12
6.1 Seq. Predicates . . . . 12
6.2 Seq. Functions . . . . 13

7 Hash Tables 15
8 Structures 16
9 Control Structure 16
9.1 Predicates . . . . 16
9.2 Variables . . . . . . 17
9.3 Functions . . . . . . 18
9.4 Macros . . . . . . 19
10 CLOS 25
10.1 Classes . . . . 25
10.2 Generic Functions . . . . . . 26
10.3 Method Combination Types . . . . . . 28

11 Conditions and Errors 28
12 Types and Classes 31
13 Input/Output 33
13.1 Predicates . . . . 33
13.2 Reader . . . . . . 33
13.3 Character Syntax . 35
13.4 Printer . . . . . . 36
13.5 Format . . . . . . 38
13.6 Streams . . . . . . 40
13.7 Paths and Files . . . . . . 42

14 Packages and Symbols 43
14.1 Predicates . . . . 43
14.2 Packages . . . . . . 44
14.3 Symbols . . . . . . 45
14.4 Std Packages . . . . . . 46
15 Compiler 46
15.1 Predicates . . . . 46
15.2 Compilation . . . . . . 46
15.3 REPL & Debug . . . . . . 47
15.4 Declarations . . . . . . 48
16 External Environment 49

Typographic Conventions

name; \_name; \_name; \_name; \_name; \_name; \_name; \_name
▶ Symbol defined in Common Lisp; esp. function, generic function, macro, special operator, variable, constant.

them ▶ Placeholder for actual code.
me ▶ Literal text.

[foo \_bar] ▶ Either one foo or nothing; defaults to bar.
foo+; \{foo\}+ ▶ Zero or more foos.
foo*; \{foo\}∗ ▶ One or more foos.
foos ▶ English plural denotes a list argument.

\{foo\}bar\_baz ▶ Either foo, or bar, or baz.

\{foo\}bar\_baz ▶ Anything from none to each of foo, bar, and baz.
foo ▶ Argument foo is not evaluated.
bar ▶ Argument bar is possibly modified.
foo\^\& ▶ foo* is evaluated as in \^\&prog\; see page 21.
foo; bar; baz ▶ Primary, secondary, and nth return value.
T; NIL ▶ t, or truth in general; and nil or ().
1 Numbers

1.1 Predicates

\[(f = \text{number}^+)\]
\[\implies T \text{ if all numbers, or none, respectively, are equal in value.}\]

\[(f > \text{number}^+)\]
\[(f >= \text{number}^+)\]
\[(f < \text{number}^+)\]
\[(f <= \text{number}^+)\]
\[\implies \text{Return } T \text{ if numbers are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.}\]

\[(f = \text{number}^+)\]
\[\implies T \text{ if } a < 0, a = 0, \text{ or } a > 0, \text{ respectively.}\]

\[(f \text{ evenp } \text{int})\]
\[\implies T \text{ if } \text{int} \text{ is even or odd, respectively.}\]

\[(f \text{ numberp } \text{foo})\]
\[(f \text{ realp } \text{foo})\]
\[(f \text{ rationalp } \text{foo})\]
\[(f \text{ floatp } \text{foo})\]
\[(f \text{ integerp } \text{foo})\]
\[(f \text{ complexp } \text{foo})\]
\[(f \text{ random-state-p } \text{foo})\]
\[\implies T \text{ if } \text{foo} \text{ is of indicated type.}\]

1.2 Numeric Functions

\[(f + \text{a} \text{0} \ast)\]
\[\implies \text{Return } \sum \text{a} \text{ or } \prod \text{a}, \text{ respectively.}\]

\[(f - \text{a} \text{b} \ast)\]
\[\implies \text{Return } \text{a} - \sum \text{b} \text{ or } \text{a}/\prod \text{b}, \text{ respectively. Without any } \text{b}s, \text{ return } -\text{a} \text{ or } 1/\text{a}, \text{ respectively.}\]

\[(f 1+ \text{a})\]
\[\implies \text{Return } \text{a} + 1 \text{ or } \text{a} - 1, \text{ respectively.}\]

\[(\{m \text{incf} m \text{decf}\} \tilde{\text{place}} [\delta_1])\]
\[\implies \text{Increment or decrement the value of } \text{place} \text{ by } \delta_1. \text{ Return new value.}\]

\[(f \text{exp} \text{p})\]
\[(f \text{expt b p})\]
\[\implies \text{Return } \text{e}^\text{p} \text{ or } \text{b}^\text{p}, \text{ respectively.}\]

\[(f \text{log a [e]})\]
\[\implies \text{Return } \log_e \text{a} \text{ or, without } e, \text{ ln } \text{a}.\]

\[(f \sqrt[n]{n})\]
\[(f \text{isqrt n})\]
\[\implies \sqrt[n]{n} \text{ in complex numbers/natural numbers.}\]

\[(f \text{lcm integer}^+)\]
\[(f \text{gcd integer}^+)\]
\[\implies \text{Least common multiple or greatest common denominator, respectively, of integers. (gcd) returns } 0.\]

\[\pi\]
\[\implies \text{long-float approximation of } \pi, \text{ Ludolph's number.}\]

\[(f \text{sin a})\]
\[(f \text{cos a})\]
\[(f \text{tan a})\]
\[\implies \text{sin } a, \text{ cos } a, \text{ or tan } a, \text{ respectively. (a in radians.)}\]

\[(f \text{asin a})\]
\[(f \text{acos a})\]
\[\implies \arcsin a \text{ or } \arccos a, \text{ respectively, in radians.}\]

\[(f \text{atan a [1]}\]
\[\implies \arctan \frac{a}{1} \text{ in radians.}\]
(\(\sinh a\)) \(\rightarrow\) \(\sinh a\), \(\cosh a\), or \(\tanh a\), respectively.

(\(\cosh a\))

(\(\tanh a\))

\(\rightarrow\) \(\sinh a\), \(\cosh a\), or \(\tanh a\), respectively.

(\(\text{cis} a\)) \(\rightarrow\) Return \(e^{ia} = \cos a + i \sin a\).

(\(\text{conjugate} a\)) \(\rightarrow\) Return complex conjugate of \(a\).

(\(\text{max} \text{ num}\)) \(\rightarrow\) Greatest or least, respectively, of \(\text{num}\).

(\(\text{round} \text{ num}\)) \(\rightarrow\) Return as integer or float, respectively, \(n/d\) rounded, or rounded towards \(-\infty\), \(+\infty\), or 0, respectively; and remainder.

(\(\text{mod} \text{ num}\)) \(\rightarrow\) Same as \(\text{floor}\) or \(\text{truncate}\), respectively, but return remainder only.

(\(\text{random-limit} \text{ state}\)) \(\rightarrow\) Return non-negative random number less than \(\text{limit}\), and of the same type.

(\(\text{random-state}\)) \(\rightarrow\) Copy of \(\text{random-state}\) object \(\text{state}\) or of the current random state; or a randomly initialized fresh \(\text{random}\) state.

(\(\text{complex} \text{ real}\)) \(\rightarrow\) Make a complex number.

(\(\text{phase} \text{ num}\)) \(\rightarrow\) Angle of \(\text{num}\)'s polar representation.

(\(\text{abs} \text{ n}\)) \(\rightarrow\) Return \(|n|\).

(\(\text{rationalize} \text{ real}\)) \(\rightarrow\) Convert \(\text{real}\) to \(\text{rational}\). Assume complete/limited accuracy for \(\text{real}\).

(\(\text{float} \text{ real}\)) \(\rightarrow\) Convert \(\text{real}\) into \(\text{float}\) with type of \(\text{prototype}\).
1.3 Logic Functions

Negative integers are used in two’s complement representation.

(\texttt{fboole operation int-a int-b})

\begin{itemize}
\item \texttt{c boole-1} \quad \triangleright \quad \text{\texttt{int-a}}.
\item \texttt{c boole-2} \quad \triangleright \quad \text{\texttt{int-b}}.
\item \texttt{c boole-c1} \quad \triangleright \quad \text{\texttt{\neg int-a}}.
\item \texttt{c boole-c2} \quad \triangleright \quad \text{\texttt{\neg int-b}}.
\item \texttt{c boole-set} \quad \triangleright \quad \text{All bits set.}
\item \texttt{c boole-clr} \quad \triangleright \quad \text{All bits zero.}
\item \texttt{c boole-eqv} \quad \triangleright \quad \text{\texttt{int-a} \equiv \texttt{int-b}}.
\item \texttt{c boole-and} \quad \triangleright \quad \text{\texttt{int-a} \wedge \texttt{int-b}}.
\item \texttt{c boole-andc1} \quad \triangleright \quad \text{\texttt{\neg int-a} \wedge \texttt{int-b}}.
\item \texttt{c boole-andc2} \quad \triangleright \quad \text{\texttt{int-a} \wedge \texttt{\neg int-b}}.
\item \texttt{c boole-nand} \quad \triangleright \quad \text{\texttt{\neg (int-a \wedge int-b)}}.
\item \texttt{c boole-ior} \quad \triangleright \quad \text{\texttt{int-a} \lor \texttt{int-b}}.
\item \texttt{c boole-orc1} \quad \triangleright \quad \text{\texttt{\neg int-a} \lor \texttt{int-b}}.
\item \texttt{c boole-orc2} \quad \triangleright \quad \text{\texttt{int-a} \lor \texttt{\neg int-b}}.
\item \texttt{c boole-xor} \quad \triangleright \quad \text{\texttt{\neg (int-a \equiv int-b)}}.
\item \texttt{c boole-nor} \quad \triangleright \quad \text{\texttt{\neg (int-a \lor int-b)}}.
\end{itemize}

(\texttt{f lognot integer})

\quad \triangleright \quad \text{\texttt{\neg integer}}.

(\texttt{f logeqv integer+})

\quad \triangleright \quad \text{Return value of exclusive-nored anded \texttt{integer}s, respectively. Without any \texttt{integer}, return \texttt{-1}.}

(\texttt{f logand integer+})

\quad \triangleright \quad \text{Return value of\texttt{exclusive-nored anded integer}s, respectively. Without any \texttt{integer}, return \texttt{0}.}

(\texttt{f logbitp i int})

\quad \triangleright \quad \text{T if zero-indexed \texttt{i}th bit of \texttt{int} is set.}

(\texttt{f logtest int-a int-b})

\quad \triangleright \quad \text{T if there is any bit set in \texttt{int-a} which is set in \texttt{int-b} as well.}

(\texttt{f logcount int})

\quad \triangleright \quad \text{Number of 1 bits in \texttt{int} \geq 0, number of 0 bits in \texttt{int} < 0.}
1.4 Integer Functions

(integer-length integer)
▷ Number of bits necessary to represent integer.

(ldb-test byte-spec integer)
▷ Return 1 if any bit specified by byte-spec in integer is set.

(ash integer count)
▷ Return copy of integer arithmetically shifted left by count
  adding zeros at the right, or, for count < 0, shifted right discarding bits.

(ldb byte-spec integer)
▷ Extract byte denoted by byte-spec from integer. setfable.

(deposit-field int-a byte-spec int-b)
▷ Return int-b with bits denoted by byte-spec replaced
  by corresponding bits of int-a, or by the low (byte-size
  byte-spec) bits of int-a, respectively.

(mask-field byte-spec integer)
▷ Return copy of integer with all bits unset but those denoted
  by byte-spec. setfable.

(byte size position)
▷ Byte specifier for a byte of size bits starting at a weight of
  2**position.

(byte-size byte-spec)
▷ Size or position, respectively, of byte-spec.

1.5 Implementation-Dependent

short-float
c single-float
double-float
c long-float
▷ Smallest possible number making a difference when added
  or subtracted, respectively.

least-negative
c least-negative-normalized
c least-positive
c least-positive-normalized
▷ Available numbers closest to −0 or +0, respectively.

most-negative
c most-positive
▷ Available numbers closest to −∞ or +∞, respectively.

(decode-float n)

(integer-decode-float n)
▷ Return significand, exponent, and sign of float n.

(scale-float n [i])
▷ With n’s radix b, return nb^i.

(float-radix n)

(float-digits n)

(float-precision n)
▷ Radix, number of digits in that radix, or precision in that
  radix, respectively, of float n.

(upgraded-complex-part-type foo [environment NIL])
▷ Type of most specialized complex number able to hold parts
  of type foo.
2 Characters

The standard-char type comprises a-z, A-Z, 0-9, Newline, Space, and !?*+-.\._<=>#%@&()
\{\}.

(characterp foo) ⇒ T if argument is of indicated type.

(standard-char-p char) ⇒ T if character is visible, alphabetic, or alphanumeric, respectively.

(graphic-char-p character) ⇒ T if character is visible, alphabetic, or alphanumeric, respectively.

(alpha-char-p character) ⇒ T if all characters, or none, respectively, are equal ignoring case.

(alphanumericp character) ⇒ T if character is visible, alphabetic, or alphanumeric, respectively.

(both-case-p character) ⇒ T if all characters, or none, respectively, are equal ignoring case.

(digit-char-p character [radix]) ⇒ Character representing digit i.


(char-code character) ⇒ Character named foo if any, or NIL.

(name-char foo) ⇒ Character with code.

(char-upper-case-p character) ⇒ T if character is uppercase, lowercase, or able to be in another case, respectively.

(char-downcase character) ⇒ T if all characters, or none, respectively, are equal ignoring case.

(char-upcase character) ⇒ T if all characters, or none, respectively, are equal ignoring case.

(char-downcase character) ⇒ T if all characters, or none, respectively, are equal ignoring case.

(char-equal character+) ⇒ T if all characters, or none, respectively, are equal.

(char-not-equal character+) ⇒ T if all characters, or none, respectively, are equal ignoring case.

(char-lessp character+) ⇒ T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-not-greaterp character+) ⇒ T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-greaterp character+) ⇒ T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-lessp character+) ⇒ T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-not-lessp character+) ⇒ T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(char-code-limit) ⇒ Upper bound of (char-code char); ≥ 96.

(character c) ⇒ Return #\c.
3 Strings

Strings can as well be manipulated by array and sequence functions; see pages 11 and 12.

\[(\text{stringp } \text{foo}) \quad \text{⇒ } \top \text{ if foo is of indicated type.}\]

\[(\text{simple-string-p } \text{foo}) \quad \text{⇒ } \top \text{ if foo is of indicated type.}\]

\[(\text{string=} \text{foo} \text{ bar}) \quad \text{⇒ } \top \text{ if subsequences of foo and bar are equal.}\]

\[(\text{string\{/=} \text{foo} \text{ bar}) \quad \text{⇒ } \top \text{ if foo is lexicographically not equal, greater, not less, less, or not greater, respectively, then return position of first mismatching character in foo. Otherwise return NIL.}\]

\[(\text{string-upcase } \text{foo}) \quad \text{⇒ } \text{all-uppercase string, respectively.}\]

\[(\text{string-downcase } \text{foo}) \quad \text{⇒ } \text{all-lowercase string, respectively.}\]

\[(\text{string-trim } \text{char-bag } \text{string}) \quad \text{⇒ } \text{string with all characters in sequence char-bag removed from both ends, from the beginning, or from the end, respectively.}\]

\[(\text{char } \text{string } \text{i}) \quad \text{⇒ } \text{zero-indexed i-th character of string ignoring/obeying, respectively, fill pointer. setfable.}\]

\[(\text{parse-integer } \text{string}) \quad \text{⇒ } \text{integer parsed from string and index of parse end.}\]

4 Conses

4.1 Predicates

\[(\text{consp } \text{foo}) \quad \text{⇒ } \top \text{ if foo is of indicated type.}\]

\[(\text{listp } \text{foo}) \quad \text{⇒ } \top \text{ if foo is of indicated type.}\]

\[(\text{endp } \text{list}) \quad \text{⇒ } \top \text{ if list/foo is NIL.}\]

\[(\text{null } \text{foo}) \quad \text{⇒ } \top \text{ if foo is NIL.}\]
Common Lisp Quick Reference

ATOM foo
\(\triangleright\) Return \(T\) if foo is not a cons.

TAILP foo list
\(\triangleright\) Return \(T\) if foo is a tail of list.

MEMBER foo list
\(\triangleright\) Return tail of list starting with its first element matching foo. Return NIL if there is no such element.

\(\triangleright\) Return tail of list starting with its first element satisfying test. Return NIL if there is no such element.

SUBSETP list-a list-b
\(\triangleright\) Return \(T\) if list-a is a subset of list-b.

LIST
\(\triangleright\) Return list of foo s.

\(\triangleright\) Return list of foo s with last foo becoming cdr of last cons. Return foo if only one foo given.

MAKE-LIST num [initial-element foo
\(\triangleright\) New list with num elements set to foo.

LENGTH list
\(\triangleright\) Length of list; NIL for circular list.

CAR list
\(\triangleright\) Car of list or NIL if list is NIL. setfable.

CDR list
\(\triangleright\) Cdr of list or NIL if list is NIL. setfable.

NTH list
\(\triangleright\) Return nth element of list if any, or NIL otherwise. setfable.

CXR list
\(\triangleright\) With X being one to four as and ds representing \(\text{car}\) and \(\text{cdrs}\), e.g. \((\text{cadr bar})\) is equivalent to \((\text{car (cadr bar)})\). setfable.

LAST list [num
\(\triangleright\) Return list of last num conses of list.

\(\triangleright\) Return list excluding last num conses.

RPLACA \(\text{cons object}\)
\(\triangleright\) Replace car, or cdr, respectively, of \text{cons} with object.

DIFF list foo
\(\triangleright\) If foo is a tail of list, return preceding part of list. Otherwise return list.

ADJOIN foo list
\(\triangleright\) Return list if foo is already member of list. If not, return \((\text{cons foo list})\).

MPOP place
\(\triangleright\) Set place to \((\text{cdr place})\), return \((\text{car place})\).
\(\texttt{push} \quad \texttt{foo} \quad \tilde{\texttt{place}} \) \quad \triangleright \quad \text{Set \ place \ to} \quad \texttt{(cons \ foo \ place)}.

\(\texttt{pushnew} \quad \texttt{foo} \quad \tilde{\texttt{place}} \) \quad \begin{align*}
\begin{cases}
\text{\texttt{test} \ function} \quad \triangleright \quad \texttt{new} \\
\text{\texttt{test-not} \ function} \quad \triangleright \quad \texttt{key} \quad \text{function}
\end{cases}
\end{align*}
\quad \triangleright \quad \text{Set} \ \texttt{place} \ \text{to} \quad \texttt{(adjoin \ foo \ place)}.

\(\texttt{append} \quad \texttt{[proper-list* \ foo]} \) \\
\(\texttt{nconc} \quad \texttt{[non-circular-list* \ foo]} \) \\
\quad \triangleright \quad \text{Return} \ \text{concatenated} \ \text{list} \ \text{or, \ with \ only \ one \ argument,} \ \texttt{foo}. \ \texttt{foo} \ \text{can \ be \ of \ any \ type}.

\(\texttt{revappend} \quad \texttt{list \ foo} \) \\
\(\texttt{nreconc} \quad \texttt{list \ foo} \) \\
\quad \triangleright \quad \text{Return} \ \text{concatenated} \ \text{list} \ \text{after \ reversing \ order \ in \ list}.

\(\{ \texttt{mapcar} \ \texttt{function \ list} \} \) \\
\(\{ \texttt{maplist} \ \texttt{function \ list} \} \) \\
\quad \triangleright \quad \text{Return} \ \text{list} \ \text{of} \ \text{return} \ \text{values} \ \text{of} \ \text{function} \ \text{successively \ invoked} \ \text{with} \ \text{corresponding} \ \text{arguments,} \ \text{either} \ \text{cars} \ \text{or} \ \text{cdrs, \ respectively,} \ \text{from} \ \text{each} \ \text{list}.

\(\{ \texttt{mapcan} \ \texttt{function} \ \tilde{\texttt{list}} \} \) \\
\(\{ \texttt{mapcon} \ \texttt{function} \ \tilde{\texttt{list}} \} \) \\
\quad \triangleright \quad \text{Return} \ \text{list} \ \text{of} \ \text{concatenated} \ \text{return} \ \text{values} \ \text{of} \ \text{function} \ \text{successively \ invoked} \ \text{with} \ \text{corresponding} \ \text{arguments,} \ \text{either} \ \text{cars} \ \text{or} \ \text{cdrs, \ respectively,} \ \text{from} \ \text{each} \ \text{list.} \ \text{function} \ \text{should \ return \ a} \ \text{list}.

\(\{ \texttt{mapc} \ \texttt{function} \ \tilde{\texttt{list}} \} \) \\
\(\{ \texttt{mapl} \ \texttt{function} \ \tilde{\texttt{list}} \} \) \\
\quad \triangleright \quad \text{Return} \ \text{first} \ \text{list} \ \text{after \ successively \ applying} \ \text{function} \ \text{to} \ \text{corresponding} \ \text{arguments,} \ \text{either} \ \text{cars} \ \text{or} \ \text{cdrs, \ respectively,} \ \text{from} \ \text{each} \ \text{list.} \ \text{function} \ \text{should \ have \ some} \ \text{side} \ \text{effects}.

\(\texttt{copy-list} \ \texttt{list} \) \\
\quad \triangleright \quad \text{Return} \ \text{copy} \ \text{of} \ \text{list} \ \text{with} \ \text{shared} \ \text{elements}.

4.3 Association Lists

\(\texttt{pairlis} \quad \texttt{keys} \ \texttt{values} \ \texttt{[alist]} \) \\
\quad \triangleright \quad \text{Prepend to} \ \texttt{alist} \ \text{an} \ \texttt{association} \ \texttt{list} \ \text{made \ from} \ \texttt{lists} \ \texttt{keys} \ \text{and} \ \texttt{values}.

\(\texttt{acons} \quad \texttt{key} \ \texttt{value} \ \texttt{alist} \) \\
\quad \triangleright \quad \text{Return} \ \texttt{alist} \ \text{with} \ \texttt{a} \ \texttt{(key} \ \text{,} \ \text{value)} \ \text{pair} \ \text{added}.

\(\{ \texttt{assoc} \ \texttt{foo} \ \texttt{alist} \} \) \\
\(\{ \texttt{rassoc} \ \texttt{foo} \ \texttt{alist} \} \) \\
\quad \begin{align*}
\begin{cases}
\text{\texttt{test} \ function} \quad \triangleright \quad \texttt{CA} \\
\text{\texttt{test-not} \ function} \quad \triangleright \quad \texttt{key} \quad \text{function}
\end{cases}
\end{align*}
\quad \triangleright \quad \text{First} \ \text{cons} \ \text{whose} \ \text{car,} \ \text{or} \ \text{cdr, \ respectively,} \ \text{satisfies} \ \text{test}.

\(\texttt{copy-alist} \ \texttt{alist} \) \\
\quad \triangleright \quad \text{Return} \ \text{copy} \ \text{of} \ \texttt{alist}.

4.4 Trees

\(\texttt{tree-equal} \quad \texttt{foo} \ \texttt{bar} \) \\
\quad \begin{align*}
\begin{cases}
\text{\texttt{test} \ function} \quad \triangleright \quad \texttt{CA} \\
\text{\texttt{test-not} \ function} \quad \triangleright \quad \texttt{key} \quad \text{function}
\end{cases}
\end{align*}
\quad \triangleright \quad \text{Return} \ \texttt{T} \ \text{if \ trees} \ \texttt{foo} \ \text{and} \ \texttt{bar} \ \text{have \ same \ shape} \ \text{and} \ \text{leaves} \ \text{satisfying} \ \text{test}.

\(\{ \texttt{subst} \ \texttt{new} \ \texttt{old} \ \texttt{tree} \} \) \\
\(\{ \texttt{nsubst} \ \texttt{new} \ \texttt{old} \ \texttt{tree} \} \) \\
\quad \begin{align*}
\begin{cases}
\text{\texttt{test} \ function} \quad \triangleright \quad \texttt{CA} \\
\text{\texttt{test-not} \ function} \quad \triangleright \quad \texttt{key} \quad \text{function}
\end{cases}
\end{align*}
\quad \triangleright \quad \text{Make} \ \text{copy} \ \text{of} \ \texttt{tree} \ \text{with} \ \text{each} \ \text{subtree \ or} \ \text{leaf} \ \text{matching} \ \texttt{old} \ \text{replaced \ by} \ \texttt{new}.

\(\{ \texttt{subst-if-not} \ \texttt{new} \ \texttt{test} \ \texttt{tree} \} \) \\
\(\{ \texttt{nsubst-if-not} \ \texttt{new} \ \texttt{test} \ \texttt{tree} \} \) \\
\quad \begin{align*}
\begin{cases}
\text{\texttt{test} \ function} \quad \triangleright \quad \texttt{CA} \\
\text{\texttt{test-not} \ function} \quad \triangleright \quad \texttt{key} \quad \text{function}
\end{cases}
\end{align*}
\quad \triangleright \quad \text{Make} \ \text{copy} \ \text{of} \ \texttt{tree} \ \text{with} \ \text{each} \ \text{subtree \ or} \ \text{leaf} \ \text{satisfying} \ \texttt{test} \ \text{replaced \ by} \ \texttt{new}.
4.5 Sets

\[ \text{\texttt{intersection \ a b}} \]  
\[ \text{\texttt{set-exclusive-or \ a b \ a \cup b \ or \ a \Delta b, \ respectively, \ of \ lists \ a \ and \ b.}} \]

5 Arrays

5.1 Predicates

\[ \text{\texttt{arrayp \ foo}} \]  
\[ \text{\texttt{vectorp \ foo}} \]  
\[ \text{\texttt{simple-vector-p \ foo}} \]  
\[ \text{\texttt{bit-vector-p \ foo}} \]  
\[ \text{\texttt{simple-bit-vector-p \ foo}} \]  
\[ \text{\texttt{adjustable-array-p \ array}} \]  
\[ \text{\texttt{array-has-fill-pointer-p \ array}} \]  
\[ \text{\texttt{array-in-bounds-p \ array \ [subscripts]}} \]  

5.2 Array Functions

\[ \text{\texttt{make-array \ dimension-sizes \ [adjustable \ boolean]}} \]  
\[ \text{\texttt{adjust-array \ array \ dimension-sizes \ \{element-type \ type \} \ \{fill-pointer \ num \ boolean\} \ \{initial-element \ obj\} \ \{initial-contents \ tree-or-array\} \ \{displaced-to \ array \ [displaced-index-offset] \} \} \]  
\[ \text{\texttt{aref \ array \ [subscripts]}} \]  
\[ \text{\texttt{row-major-aref \ array \ i}} \]  
\[ \text{\texttt{array-dimensions \ array}} \]  
\[ \text{\texttt{array-dimension \ array \ i}} \]  
\[ \text{\texttt{array-total-size \ array}} \]  
\[ \text{\texttt{array-rank \ array}} \]  
\[ \text{\texttt{array-displacement \ array}} \]  

\[ \text{\texttt{sublis \ association-list \ tree \ \{test \ function \ \#eql \} \ \{test-not \ function \} \ \{key \ function \}} \]  
\[ \text{\texttt{nsublis \ association-list \ tree}} \]  
\[ \text{\texttt{Make \ copy \ of \ tree \ with \ each \ subtree \ or \ leaf \ matching \ a \ key \ in \ association-list \ replaced \ by \ that \ key's \ value.}} \]

\[ \text{\texttt{copy-tree \ tree}} \]  
\[ \text{\texttt{Copy \ of \ tree \ with \ same \ shape \ and \ leaves.}} \]
Common Lisp Quick Reference

(bit bit-array [subscripts])

(bit not bit-array [result-bit-array])

(bit-eqv bit-array-a bit-array-b [result-bit-array])

(bit-not ~bit-array) [result-bit-array NIL]

(bit-eqv bit-array-a bit-array-b [result-bit-array])

(bit-and bit-array-a bit-array-b [result-bit-array])

(bit-andc1 bit-array-a bit-array-b [result-bit-array])

(bit-andc2 bit-array-a bit-array-b [result-bit-array])

(bit-nand bit-array-a bit-array-b [result-bit-array])

(bit-ior bit-array-a bit-array-b [result-bit-array])

(bit-iorc1 bit-array-a bit-array-b [result-bit-array])

(bit-iorc2 bit-array-a bit-array-b [result-bit-array])

(bit-xor bit-array-a bit-array-b [result-bit-array])

(bit-nor bit-array-a bit-array-b [result-bit-array])

(setf -able)

(array-rank-limit)

(array-dimension-limit)

(array-total-size-limit)

(f vector foo) [result-vector]

(svref vector i)

(vector-push foo ~vector)

(vector-push-extend foo vector [num])

(vector-pop ~vector)

(fill-pointer vector)

6 Sequences

6.1 Sequence Predicates

(every test sequence+)

(not-every test sequence+)

(some test sequence+)

(not-any test sequence+)

Return element of bit-array or of simple-bit-array.

Return result of bitwise negation of bit-array. If result-bit-array is T, put result in bit-array; if it is NIL, make a new array for result.

Return result of bitwise logical operations (cf. operations of /boole, page 5) on bit-array-a and bit-array-b. If result-bit-array is T, put result in bit-array-a; if it is NIL, make a new array for result.

Upper bound of array rank; ≥ 8.

Upper bound of an array dimension; ≥ 1024.

Upper bound of array size; ≥ 1024.

Vectors can as well be manipulated by sequence functions; see section 6.

Return fresh simple vector of foos.

Element i of simple vector.

Return NIL if vector’s fill pointer equals size of vector. Otherwise replace element of vector pointed to by fill pointer with foo; then increment fill pointer.

Replace element of vector pointed to by fill pointer with foo, then increment fill pointer. Extend vector’s size by ≥ num if necessary.

Return element of vector its fillpointer points to after decrementation.

Fill pointer of vector.
6.2 Sequence Functions

\( \text{\texttt{(\text{make-sequence} \ sequence-type-size \ initial-element \ foo)\)}} \)

\( \triangleright \) Make sequence of sequence-type with size elements.

\( \text{\texttt{(\text{concatenate} \ type \ sequence\)}} \)

\( \triangleright \) Return concatenated sequence of type.

\( \text{\texttt{(\text{merge} \ type \ sequence-a \ sequence-b \ test \ \text{\textit{key function}}} \)} \)

\( \triangleright \) Return interleaved sequence of type. Merged sequence will be sorted if both sequence-a and sequence-b are sorted.

\( \text{\texttt{(\text{fill} \ sequence \ foo \ \{\text{\texttt{start} \ start1} \text{\texttt{end} \ end1}\)}} \}

\( \triangleright \) Return sequence after setting elements between start and end to foo.

\( \text{\texttt{(\text{length} \ sequence)\)}} \)

\( \triangleright \) Return length of sequence (being value of fill pointer if applicable).

\( \text{\texttt{(\text{count} \ foo \ sequence \ \{\text{\texttt{start} \ start1} \text{\texttt{end} \ end1}\)}} \}

\( \triangleright \) Return number of elements in sequence which match foo.

\( \text{\texttt{(\text{count-if} \ test \ sequence \ \{\text{\texttt{start} \ start1} \text{\texttt{end} \ end1}\)}} \}

\( \triangleright \) Return number of elements in sequence which satisfy test.

\( \text{\texttt{(\text{elt} \ sequence \ index)\)}} \)

\( \triangleright \) Return element of sequence pointed to by zero-indexed index. settable.

\( \text{\texttt{(\text{subseq} \ sequence \ \{\text{\texttt{start} \ start1} \text{\texttt{end} \ end1}\)}} \)

\( \triangleright \) Return subsequence of sequence between start and end. settable.

\( \text{\texttt{(\text{sort} \ sequence \ test \ \text{\textit{key function}}} \)} \)

\( \triangleright \) Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.

\( \text{\texttt{(\text{reverse} \ sequence)\)}} \)

\( \triangleright \) Return sequence in reverse order.

\( \text{\texttt{(\text{reverse} \ sequence)\)}} \)

\( \triangleright \) Return sequence in reverse order.

\( \text{\texttt{(\text{find} \ position \ foo \ sequence \ \{\text{\texttt{start} \ start1} \text{\texttt{end} \ end1}\)}} \}

\( \triangleright \) Return first element in sequence which matches foo, or its position relative to the begin of sequence, respectively.
Return first element in sequence which satisfies test, or its position relative to the begin of sequence, respectively.

Search sequence-b for a subsequence matching sequence-a. Return position in sequence-b, or NIL.

Make copy of sequence without elements matching foo.

Make copy of sequence with all (or count) elements satisfying test removed.

Make copy of sequence without duplicates.

Make copy of sequence with all (or count) olds replaced by new.

Make copy of sequence with all (or count) elements satisfying test replaced by new.

Replace elements of sequence-a with elements of sequence-b.

Apply function successively to corresponding elements of the sequences. Return values as a sequence of type. If type is NIL, return NIL.


(map-into \textit{result-sequence} function sequence \dagger)
\begin{itemize}
  \item Store into \textit{result-sequence} successively values of \textit{function}
    applied to corresponding elements of the sequences.
\end{itemize}

\textit{(reduce} function sequence\textit{)}
\begin{itemize}
  \item Starting with the first two elements of \textit{sequence}, apply \textit{function}
    successively to its last return value together with the next element of \textit{sequence}.
    Return last value of \textit{function}.
\end{itemize}

\textit{(copy-seq} sequence\textit{)}
\begin{itemize}
  \item Copy of \textit{sequence} with shared elements.
\end{itemize}

7 Hash Tables

The Loop Facility provides additional hash table-related functionality; see \textit{loop}, page 22.

Key-value storage similar to hash tables can as well be achieved using association lists and property lists; see pages 10 and 17.

\textit{(hash-table-p} foo\textit{)}
\begin{itemize}
  \item Return \texttt{T} if \textit{foo} is of type \texttt{hash-table}.
\end{itemize}

\textit{(make-hash-table} \begin{itemize}
  \item \text{test} \{\texttt{eq}, \texttt{eql}, \texttt{equal}, \texttt{equalp}\}
  \item \text{size} \texttt{int}
  \item \text{rehash-size} \texttt{num}
  \item \text{rehash-threshold} \texttt{num}
\end{itemize}\textit{)}
\begin{itemize}
  \item Make a hash table.
\end{itemize}

\textit{(gethash} key hash-table [default\texttt{NIL}]\textit{)}
\begin{itemize}
  \item Return object with key if any or \texttt{default} otherwise; and \texttt{T} if found, \texttt{NIL} otherwise. \texttt{setf} table.
\end{itemize}

\textit{(hash-table-count} hash-table\textit{)}
\begin{itemize}
  \item Number of entries in \textit{hash-table}.
\end{itemize}

\textit{(remhash} key hash-table\textit{)}
\begin{itemize}
  \item Remove from \textit{hash-table} entry with key and return \texttt{T} if it existed. Return \texttt{NIL} otherwise.
\end{itemize}

\textit{(clrhash} \textit{hash-table}\textit{)}
\begin{itemize}
  \item Empty \textit{hash-table}.
\end{itemize}

\textit{(maphash} function hash-table\textit{)}
\begin{itemize}
  \item Iterate over \textit{hash-table} calling \textit{function} on key and value.
  \item Return \texttt{NIL}.
\end{itemize}

\textit{(with-hash-table-iterator} (foo hash-table) \textit{(declare} \texttt{decl}* \textit{form}*\textit{)}\textit{)}
\begin{itemize}
  \item Return values of \textit{forms}. In \textit{forms}, invocations of (foo) return: \texttt{T} if an entry is returned; its key; its value.
\end{itemize}

\textit{(hash-table-test} hash-table\textit{)}
\begin{itemize}
  \item Test function used in \textit{hash-table}.
\end{itemize}

\textit{(hash-table-size} hash-table\textit{)}
\textit{(hash-table-rehash-size} hash-table\textit{)}
\textit{(hash-table-rehash-threshold} hash-table\textit{)}
\begin{itemize}
  \item Current size, rehash-size, or rehash-threshold, respectively, as used in \textit{make-hash-table}.
\end{itemize}

\textit{(sxhash} foo\textit{)}
\begin{itemize}
  \item Hash code unique for any argument, \texttt{equal} \textit{foo}.
\end{itemize}
8 Structures

(defun defstruct (foo)
  (defstruct (foo
    (:conc-name (getf foo 'conc-name))
    (:constructor (foo (maker :ord-foo)))
    (:copier (foo :copy-foo)))
  (include struct
    (slot
      (:type list vector :type)
      (:print-object (print-object))
      (:print-function (print-function))
      (:predicate (predicate :name))
      (doc (slot-init (type slot-type :read-only bool)))))
  ; Define structure foo together with functions MAKE-foo, COPY-foo and foo-P; and set/able accessors foo-slot. Instances are of class foo or, if defstruct option :type is given, of the specified type. They can be created by (MAKE-foo \{(slot value)\}) or, if ord-\lambda (see page 18) is given, by \{(maker arg* \{(key value)\}\}). In the latter case, args and keys correspond to the positional and keyword parameters defined in ord-\lambda whose vars in turn correspond to slots.
  ;print-object/:print-function generate a :print-object method for an instance bar of foo calling \{(o-printer bar stream) or \{(f-printer bar stream print-level)\}, respectively. If :type without named is given, no foo-P is created.

(defun copy-structure structure)
  ; Return copy of structure with shared slot values.

9 Control Structure

9.1 Predicates

(defun eq foo bar)
  ; if foo and bar are identical.

(defun eql foo bar)
  ; if foo and bar are identical, or the same character, or numbers of the same type and value.

(defun equal foo bar)
  ; if foo and bar are eql, or are equivalent pathnames, or are conses with equal cars and cdrs, or are strings or bit-vectors with eql elements below their fill pointers.

(defun equalp foo bar)
  ; if foo and bar are identical; or are the same character ignoring case; or are numbers of the same value ignoring type; or are equivalent pathnames; or are conses or arrays of the same shape with equalp elements; or are structures of the same type with equalp elements; or are hash-tables of the same size with the same :test function, the same keys in terms of :test function, and equalp elements.

(defun not foo)
  ; if foo is NIL; NIL otherwise.

(defun boundp symbol)
  ; if symbol is a special variable.
\(\text{constantp } \text{foo} [\text{environment} \text{foo}]\)  \(\triangleright\) \(\text{T}\) if \(\text{foo}\) is a constant form.

\(\text{functionp } \text{foo}\)  \(\triangleright\) \(\text{T}\) if \(\text{foo}\) is of type function.

\(\text{fboundp } \{\text{setf } \text{foo}\}\)  \(\triangleright\) \(\text{T}\) if \(\text{foo}\) is a global function or macro.

9.2 Variables

\(\text{defconstant } \text{foo form} \{\text{doc}\}\)  \(\triangleright\) Assign value of \(\text{form}\) to global constant/dynamic variable \(\text{foo}\).

\(\text{defparameter } \text{foo} \{\text{form} \{\text{doc}\}\}\)  \(\triangleright\) Unless bound already, assign value of \(\text{form}\) to dynamic variable \(\text{foo}\).

\(\text{set} \{\text{place form}^*\}\)  \(\triangleright\) Set \(\text{places}\) to primary values of \(\text{forms}\). Return values of last form/\text{NIL}; work sequentially/in parallel, respectively.

\(\text{setq} \{\text{symbol form}^*\}\)  \(\triangleright\) Set symbols to primary values of \(\text{forms}\). Return value of last form/\text{NIL}; work sequentially/in parallel, respectively.

\(\text{makunbound } \text{foo}\)  \(\triangleright\) Delete special variable \(\text{foo}\) if any.

\(\text{progv} \{\text{symbols values form} \{\text{P}\}\}\)  \(\triangleright\) Evaluate \(\text{forms}\) with locally established dynamic bindings of \(\text{symbols}\) to \(\text{values}\) or \text{NIL}. Return values of \(\text{forms}\).
\[\text{multiple-value-bind (var\#)} \text{ values-form (declare decl*)}^* \text{ body-forms}^*\]

▷ Evaluate body-forms with vars lexically bound to the return values of values-form. Return values of body-forms.

\[\text{destructuring-bind destruct-\lambda bar (declare decl*)}^* \text{ form}\]

▷ Evaluate forms with variables from tree destruct-\lambda bound to corresponding elements of tree bar, and return their values. destruct-\lambda resembles macro-\lambda (section 9.4), but without any &environment clause.

### 9.3 Functions

Below, ordinary lambda list (ord-\lambda*) has the form

\[
\text{(var* \&optional [var \init]\{[supplied-p]\}) | \&rest var}
\]

\[
\text{[&key \{[var \init]\{[supplied-p]\}] \&allow-other-keys]]}
\]

\[
\text{[&aux \{[var \init]\{[supplied-p]\}]}})
\]

supplied-p is T if there is a corresponding argument. init forms can refer to any init and supplied-p to their left.

\[
\begin{align*}
\text{\{defun foo (ord-\lambda*)} &\text{ \{setf foo\} (new-value ord-\lambda*)} \text{ \{declare decl\}* \{doc\} } \text{form}\]
\end{align*}
\]

▷ Define a function named foo or (setf foo), or an anonymous function, respectively, which applies forms to ord-\lambda*. For mdefun, forms are enclosed in an implicit \block named foo.

\[
\begin{align*}
\text{\{flet [labels] \{\{foo (ord-\lambda*)} &\text{ \{setf foo\} (new-value ord-\lambda*)} \text{ \{declare decl\} \text{form}\}\} \text{ \{doc\} \text{local-forms\} \{declare decl\}* \text{form}\} } \text{form}\]
\end{align*}
\]

▷ Evaluate forms with locally defined functions foo. Globally defined functions of the same name are shadowed. Each foo is also the name of an implicit \block around its corresponding local-forms. Only for \labels, functions foo are visible inside local-forms. Return values of forms.

\[
\text{\{function foo \{lambda form\}}\}
\]

▷ Return lexically innermost function named foo or a lexical closure of the \lambda expression.

\[
\text{\{apply \{function \{setf function\} arg* args\} \text{arg* args}\}
\]

▷ Values of function called with args and the list elements of args. setfable if function is one of \aref, \bit, and \sbit.

\[
\text{\{funcall function arg\}^* \text{ Values of function called with args.\}
\]

\[
\text{\{multiple-value-call function form\}^* \text{ Call function with all the values of each form as its arguments. Return values returned by function.\}
\]

\[
\text{\{values-list list\} \text{Return elements of list.\}
\]

\[
\text{\{values foo\}^* \text{Return as multiple values the primary values of the foos. setfable.\}
\]

\[
\text{\{multiple-value-list form\} \text{List of the values of form.\}
\]

\[
\text{\{nth-value n form\} \text{Zero-indexed n-th return value of form.\}
\]

\[
\text{\{complement function\} \text{Return new function with same arguments and same side effects as function, but with complementary truth value.\}
\]

18
\(constantly\ foo\)  
\(\triangleright\) Function of any number of arguments returning \texttt{foo}.

\(identity\ foo\)  
\(\triangleright\) Return \texttt{foo}.

\(function-lambda-expression\ function\)  
\(\triangleright\) If available, return lambda expression of \texttt{function}, \texttt{NIL} if \texttt{function} was defined in an environment without bindings, and \texttt{name} of \texttt{function}.

\(definition\ \{\texttt{foo} \{\texttt{setf} \texttt{foo}\}\}\)  
\(\triangleright\) Definition of global function \texttt{foo}. \texttt{settable}.

\(fmakunbound\ foo\)  
\(\triangleright\) Remove global function or macro definition \texttt{foo}.

\(\text{call-arguments-limit}\)  
\(\text{lambda-parameters-limit}\)  
\(\triangleright\) Upper bound of the number of function arguments or lambda list parameters, respectively; \(\geq 50\).

\(\text{multiple-values-limit}\)  
\(\triangleright\) Upper bound of the number of values a multiple value can have; \(\geq 20\).

\section{9.4 Macros}

Below, macro lambda list \((\text{macro-}\lambda^{*})\) has the form of either

\[
([\text{&whole}\ \texttt{var}] [E] ([\text{macro-}\lambda^{*}]) [E])
\]

\[
([\text{&optional}\ \texttt{var} ([\{\text{macro-}\lambda^{*}\} [\text{init}\texttt{\{\texttt{supplied-p}\}]}])^{*}] [E])
\]

\[
([\text{&rest}\ \texttt{var} ([\text{macro-}\lambda^{*}]) [E])
\]

\[
([\text{&key} \{\texttt{var} ([\text{macro-}\lambda^{*}]) [\text{init}\texttt{\{\texttt{supplied-p}\}]}]\} [E])
\]

or

\[
([\text{&allow-other-keys}\] [\text{&aux} \{\texttt{var} [\text{init}]}\} [E])
\]

\[
([\text{&whole}\ \texttt{var}] [E] ([\text{macro-}\lambda^{*}]) [E] [\text{&optional}\ \{\texttt{var} [\text{init}][\texttt{\{\texttt{supplied-p}\}]}]\} [E] . \text{rest-var}).
\]

One toplevel \([E]\) may be replaced by \&\texttt{environment} \texttt{var}. \texttt{supplied-p} is \texttt{T} if there is a corresponding argument. \texttt{init} forms can refer to any \texttt{init} and \texttt{supplied-p} to their left.

\[
(\text{\texttt{define}\text{-}\texttt{compiler}\text{-}macro}\ \{\text{foo} \{\texttt{setf} \texttt{foo}\}\} ([\text{macro-}\lambda^{*}]) ([\text{declare} \texttt{decl}^{*}])^{*} \texttt{\{doc\ \texttt{form}\}^{*}})
\]

\(\triangleright\) Define macro \texttt{foo} which on evaluation as \((\texttt{foo} \texttt{tree})\) applies \texttt{expanded forms} to arguments from \texttt{tree}, which corresponds to \texttt{tree}-shaped \texttt{macro-\lambda}s. \texttt{forms} are enclosed in an implicit \texttt{\texttt{block}} named \texttt{foo}.

\[
(\text{\texttt{define}\text{-}\texttt{symbol}\text{-}macro\ \texttt{form}})
\]

\(\triangleright\) Define symbol macro \texttt{foo} which on evaluation evaluates \texttt{expanded form}.

\[
(\text{\texttt{macrolet}} (\{\text{foo} ([\text{macro-}\lambda^{*}]) ([\text{declare} \texttt{decl}^{*}])^{*} \texttt{\{doc\ \texttt{form}\}^{*}})
\]

\(\triangleright\) Evaluate \texttt{forms} with locally defined mutually invisible macros \texttt{foo} which are enclosed in implicit \texttt{\texttt{blocks}} of the same name.
(defmacro \(\text{foo}\) * \(\text{form}\*) | (\text{declare} \text{decl}*) * \(\text{form}\*)|)  
\text{Evaluate forms with locally defined symbol macros.}

(define-setf-expander \(\text{function}\) \(\text{arg-vars}\) \(\text{form}\*) | (\text{declare} \text{decl}*) * \(\text{form}\*)|)  
\text{Specify how to \text{setf} a place accessed by \text{function}. Short form: \(\text{function arg}\) \text{value-form} \(\Rightarrow\) \text{form}, \text{forms} must expand into code returning \text{arg-vars} and \text{set-form} where the elements of \text{macro lambda list} \text{macro-list-keywords} are \text{bound to corresponding arguments if any}. Long form: on invocation of \(\text{setf} \text{function arg}\) \text{value-form} \(\Rightarrow\) \text{form}, \text{forms} must expand into code that sets the place accessed where \text{self-} and \text{s-var} describe the arguments of \text{function} and the value(s) to be stored, respectively; and that returns the value(s) of \text{s-var}*. \text{forms} are enclosed in an implicit \text{block } named \text{function}.

(define-modify-macro \text{foo} \(\text{form}\*) \(\text{doc}\) \text{var} \(\text{var}\) \(\text{newval-vars}\) \(\text{set-form}\) \(\text{get-form}\) \(\text{arg-vars}\) \text{var} \text{&rest} \text{var} \text{&key} \text{var} \text{&allow-other-keys} \text{var} \text{&environment} \text{var})  
\text{Define macro \text{foo} able to modify a place. On invocation of \(\text{foo place arg}\) \text{value-form}, the value of \text{function} applied to \text{place} and \text{args} will be stored into \text{place} and returned.}

\text{lambda-list-keywords} \text{var} \text{&whole} \text{var} \text{&optional} \text{var} \text{&rest} \text{var} \text{&key} \text{var} \text{&allow-other-keys} \text{var} \text{&environment} \text{var} \text{&aux} \text{var}  
\text{List of macro lambda list keywords. These are at least:}
9.5 Control Flow

(if test then else)

➤ Return values of then if test returns T; return values of else otherwise.

(when test)

➤ Return the values of the first then whose test returns T; return NIL if all tests return NIL.

(unless test foo)

➤ Evaluate foos and return their values if test returns T or NIL, respectively. Return NIL otherwise.

(case test (key foo T)

➤ Return the values of the first foo whose key returns T; return values of bars if there is no matching key.

(multiple-value-prog1 form-r form*)

➤ Evaluate forms in order. Return values/primary value, respectively, of form-r.

(declare tag form)

➤ Evaluate s tagbody-like body with names lexically bound (in parallel or sequentially, respectively) to values. Return NIL or explicitly returned values. Implicitly, the whole form is a s block named NIL.

(unwind-protect protected cleanup)

➤ Evaluate protected and then, no matter how control leaves protected, cleansups. Return values of protected.

(block name form)

➤ Evaluate forms in a lexical environment, and return their values unless interrupted by ,return-from.

(return-from foo result)

➤ Have nearest enclosing block named foo/named NIL, respectively, return with values of result.

(tagbody tag form*)

➤ Evaluate forms in a lexical environment. tags (symbols or integers) have lexical scope and dynamic extent, and are targets for ,go. Return NIL.

(go tag)

➤ Within the innermost possible enclosing tagbody, jump to a tag ,eq tag.
9.6 Iteration

\[(\text{\textbf{do}} \text{\textbf{form}}) \] \> Evaluate \texttt{tagbody}-like body with \texttt{var} successively bound according to the values of the corresponding \texttt{start} and \texttt{step} forms. \texttt{vars} are bound in parallel/sequentially, respectively. Stop iteration when \texttt{stop} is \texttt{T}. Return values of \texttt{result}. Implicitly, the whole form is a \texttt{block} named \texttt{NIL}.

\[(\text{\texttt{loop}} \text{\texttt{clause}}) \] \> Loop Facility. For Loop Facility keywords see below and Figure 1.

\[\{\text{\texttt{for}} \text{\texttt{as}} \text{\texttt{var}} \text{\texttt{from}} \} \] \> Begin of iteration control clauses. Initialize and step (possibly trees of) local variables \texttt{var-s} sequentially and \texttt{var-p} in parallel. Destructuring type specifier \texttt{d-type} as with \texttt{with}.

\{(\text{\texttt{upfrom downfrom}}) \text{\texttt{start}} \] \> Start stepping with \texttt{start}

\{(\text{\texttt{upto downto below above}}) \text{\texttt{form}} \] \> Specify \texttt{form} as the end value for stepping.

\{(\text{\texttt{in on}}) \text{\texttt{list}} \] \> Bind \texttt{var} to successive elements/tails, respectively, of \texttt{list}.

\{\text{\texttt{by step function \#cdr}} \] \> Specify the (positive) decrement or increment or the function of one argument returning the next part of the list.
Common Lisp Quick Reference

### Loop Facility

**Overview**

```lisp
(loop [var (var*)] [d-type] \[foo\] \(and\) \[var*\] \[d-type] \[bar\]*

- **with**: `\{var\}`
- **named**: `\{var\}`
- **NIL**: `\NIL`

**Example**

```lisp
(loop [var (var*)] [d-type] \[foo\] \(and\) \[var*\] \[d-type] \[bar\]*

- **foras**: `\{var\}`
- **doings**: `\{form\}`

**Conditions**

- **ifwhenunless**: `\{form\}`
- **then**: `\{form\}`
- **else**: `\{form\}`
- **end**: `\{form\}`

**Tests**

- **return**: `\{form\}`
- **collecting**: `\{form\}`
- **appending**: `\{form\}`
- **nconc**: `\{form\}`

**Counts**

- **counting**: `\{form\}`
- **summing**: `\{form\}`
- **maximizing**: `\{form\}`
- **minimizing**: `\{form\}`

**Iterators**

- **repeat**: `\{form\}`
- **while**: `\{form\}`
- **until**: `\{form\}`
- **always**: `\{form\}`
- **never**: `\{form\}`

**Vectors**

- **across**: `\{form\}`

**Lists**

- **into**: `\{form\}`

**Hashes**

- **hash-key[s]**
- **hash-value[s]**

**Symbols**

- **symbol[s]**
- **present-symbol[s]**
- **external-symbol[s]**

**Packages**

- **package[s]**

**Figure 1**

Overview of the Loop Facility.
= foo [then bar]
  Bind var initially to foo and later to bar.

across vector
  Bind var to successive elements of vector.

being {the}each
  Iterate over a hash table or a package.

  {hash-key|hash-keys} {of|in} hash-table [using
  (hash-value value)]
  Bind var successively to the keys of hash-table; bind value
  to corresponding values.

  {hash-value|hash-values} {of|in} hash-table [using
  (hash-key key)]
  Bind var successively to the values of hash-table; bind key
  to corresponding keys.

  {symbol|symbols|present-symbol|present-symbols|external-symbol|external-symbols} {of|in}
  package[+package]
  Bind var successively to the accessible symbols,
  or the present symbols, or the external symbols
  respectively, of package.

{do|doing} form+
  Evaluate forms in every iteration.

{if|when|unless} test i-clause {and|and|j-clause}* [else k-clause
{and|and|l-clause}]* [end]
  If test returns T, T, or NIL, respectively, evaluate
  i-clause and j-clauses; otherwise, evaluate k-clause
  and l-clauses.

  it
  Inside i-clause or k-clause: value of test.

return {form|it}
  Return immediately, skipping any finally parts,
  with values of form or it.

collect|collecting} {form|it} [into list]
  Collect values of form or it into list. If no list is given,
  collect into an anonymous list which is returned after
  termination.

append|appending|nconc|nconcing} {form|it} [into list]
  Concatenate values of form or it, which should be lists,
  into list by the means of append or nconc, respectively.
  If no list is given, collect into an anonymous list
  which is returned after termination.

count|counting} {form|it} [into n] [type]
  Count the number of times the value of form or of it
  is T. If no n is given, count into an anonymous variable
  which is returned after termination.

sum|summing} {form|it} [into sum] [type]
  Calculate the sum of the primary values of form or of it.
  If no sum is given, sum into an anonymous variable
  which is returned after termination.

(maximize|maximizing|minimize|minimizing} {form|it} [into
  max-min] [type]
  Determine the maximum or minimum, respectively, of
  the primary values of form or of it. If no max-min
  is given, use an anonymous variable which is returned after
  termination.

{initially|finally} form+
  Evaluate forms before begin, or after end, respectively,
  of iterations.

repeat num
  Terminate mloop after num iterations; num is evaluated
  once.

while|until} test
  Continue iteration until test returns NIL or T, respec-
  tively.
```
{always|never} test
  → Terminate mloop returning NIL and skipping any finally parts as soon as test is NIL or T, respectively. Otherwise continue mloop with its default return value set to T.

thereis test
  → Terminate mloop when test is T and return value of test, skipping any finally parts. Otherwise continue mloop with its default return value set to NIL.

(m-loop-finish)
  → Terminate mloop immediately executing any finally clauses and returning any accumulated results.

10 CLOS

10.1 Classes
```

```
(defun defclass (foo (superclass"standard-object")
  (slot)
    (\{reader reader\}
      \{writer \{setf writer\}\}
      \{accessor accessor\}*)
  (\{slot-makunbound\}
    instance slot)
  (\{slot-exists-p\}
    \{instance slot\})
  (\{slot-boundp\}
    instance slot)
  (\{slot-value\}
    instance slot)
  (\{find-class\}
    symbol [error] [environment])
    → Return class named symbol. setfable.
  (\{make-instance\}
    class \{initarg \{initarg-name\}\} other-keyarg*)
    → Make new instance of class.
  (\{reinitialize-instance\}
    instance \{initarg value\} other-keyarg*)
    → Change local slots of instance according to initargs by means of shared-initialize.
  (\{slot-value\}
    foo slot)
    → Return value of slot in class foo. setfable.
  (\{slot-makunbound\}
    instance slot)
    → Make slot in instance unbound.
  (\{with-slots\}
    \{\{instance\}
      \{\{declare\} decl\}*
      \{\{form\} form\}*)
  (\{with-accessors\}
    \{\{instance\}
      \{\{form\} form\}*)
  (\{class-name\}
    class)
    → Get/set name of class.
  (\{class-of\}
    foo)
    → Class foo is a direct instance of.
```
(defmethod standard-generic-function (foo var) (declare (optimize method-selection-optimization +) (:documentation """) (:required-var+ """) (:method-class method-class-class-name) (:method-combination method-combination-class) (:method-args method-args-class))

> Define or modify generic function `foo`. Remove any methods previously defined by `defgeneric`. `gf-class` and the lambda parameters `required-var+` and `var+` must be compatible with existing methods. `defmethod-args` resemble those of `mdefmethod`. For `c-type` see section 10.3.
Before the body of a method is executed, a method selection process is performed. This process is driven by the call-next-method function, which is called on invocation of a method or from within a method. The call-next-method function is responsible for determining whether the current method is the primary method or whether it should call the next method.

The method selection process is driven by the call-next-method function, which is called on invocation of a method or from within a method. The call-next-method function is responsible for determining whether the current method is the primary method or whether it should call the next method.

The call-next-method function is called on invocation of a method or from within a method. The call-next-method function is responsible for determining whether the current method is the primary method or whether it should call the next method.

The call-next-method function is called on invocation of a method or from within a method. The call-next-method function is responsible for determining whether the current method is the primary method or whether it should call the next method.

The call-next-method function is called on invocation of a method or from within a method. The call-next-method function is responsible for determining whether the current method is the primary method or whether it should call the next method.

The call-next-method function is called on invocation of a method or from within a method. The call-next-method function is responsible for determining whether the current method is the primary method or whether it should call the next method.

The call-next-method function is called on invocation of a method or from within a method. The call-next-method function is responsible for determining whether the current method is the primary method or whether it should call the next method.

The call-next-method function is called on invocation of a method or from within a method. The call-next-method function is responsible for determining whether the current method is the primary method or whether it should call the next method.

The call-next-method function is called on invocation of a method or from within a method. The call-next-method function is responsible for determining whether the current method is the primary method or whether it should call the next method.

The call-next-method function is called on invocation of a method or from within a method. The call-next-method function is responsible for determining whether the current method is the primary method or whether it should call the next method.
10.3 Method Combination Types

standard

- Evaluate most specific * around method supplying the values of the generic function. From within this method, * call-next-method can call less specific * around methods if there are any. If not, or if there are no * around methods at all, call all before methods, most specific first, and the most specific primary method which supplies the values of the calling * call-next-method if any, or of the generic function; and which can call less specific primary methods via * call-next-method. After its return, call all after methods, least specific first.

and/or/append/list/concat/max/min+
- Simple built-in method-combination types; have the same usage as the c-types defined by the short form of * define-method-combination.

(*define-method-combination* c-type

  {::documentation string
   ::identity-with-one-argument boolean
   ::operator operator}

  * Short Form. Define new method-combination c-type. In a generic function using c-type, evaluate most specific * around method supplying the values of the generic function. From within this method, * call-next-method can call less specific * around methods if there are any. If not, or if there are no * around methods at all, return from the calling call-next-method or from the generic function, respectively, the values of (operator (primary-method gen-arg *)), gen-arg * being the arguments of the generic function. The primary-methods are ordered {::most-specific-first
   ::most-specific-last [most-specific-first]}

   * (specified as c-arg in * defgeneric). Using c-type as the qualifier in * defmethod makes the method primary.

(*define-method-combination* c-type (ord-λ*) ((group

  * (qualifier* #}))

  {::description control
   ::order [::most-specific-first
     ::most-specific-last most-specific-first]}

  {::arguments method-combination-λ*
     (generic-function symbol)
     (declare decl*)}

  body*)

* Long Form. Define new method-combination c-type. A call to a generic function using c-type will be equivalent to a call to the forms returned by body * with ord-λ* bound to c-arg * (cf. * defgeneric), with symbol bound to the generic function, with method-combination-λ* bound to the arguments of the generic function, and with groups bound to lists of methods. An applicable method becomes a member of the leftmost group whose predicate or qualifiers match. Methods can be called via * call-method. Lambda lists (ord-λ*) and (method-combination-λ*) according to ord-λ on page 18, the latter enhanced by an optional &whole argument.

(*call-method

  {::make-method form}

  {::next-method [::make-method form]}

  * From within an effective method form, call method with the arguments of the generic function and with information about its next-methods; return its values.

11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 32.
(define-condition foo (parent-type)  
{s1}
(slot (reader reader))
(slot (writer (self writer)))))

=! Return new instance of condition-type.

(make-condition condition-type (:\initarg :\initarg-name value))

= Define, as a subtype of parent-types, condition type foo. In a new condition, a slot’s value defaults to form unless set via :\initarg-name; it is readable via (reader i) or (accessor i), and writable via (writer value i) or (setf accessor i). With :\allocation :\class, slot is shared by all conditions of type foo. A condition is reported by string or by report-function of arguments condition and stream.

(\ignore-errors form)  
= Return values of forms or, in case of errors, NIL and the condition.

:invoke-debugger condition)
= Invoke debugger with condition.
(with-simple-restart (restart \{NIL\}) control arg*) form

▷ Return values of forms unless restart is called during their evaluation. In this case, describe restart using format control and args (see page 38) and return NIL and T.

(restart-case form (restart (ord-\lambda*))
  \{interactive \(\text{arg-function}\) \{report-function \{string\} \{test-function\} \}

(declare decl*) restart-form

▷ Return values of form or, if during evaluation of form one of the dynamically established restarts is called, the values of its restart-forms. A restart is visible under condition if (funcall \#"\text{test-function condition}\) returns T. If presented in the debugger, restarts are described by string or by \#"\text{report-function}\ (of a stream). A restart can be called by (invoke-restart restart arg*), where args match ord-\lambda*; or by (invoke-restart-interactively restart) where a list of the respective args is supplied by \#"\text{arg-function}\. See page 18 for ord-\lambda*.

(restart-bind (restart \{NIL\}) restart-function
  \{interactive-function \(\text{arg-function}\) \{report-function report-function \} \{test-function test-function\}

▷ Return values of forms evaluated with dynamically established restarts whose restart-functions should perform a non-local transfer of control. A restart is visible under condition if (test-function condition) returns T. If presented in the debugger, restarts are described by restart-function (of a stream). A restart can be called by (invoke-restart restart arg*), where args must be suitable for the corresponding restart-function, or by (invoke-restart-interactively restart) where a list of the respective args is supplied by arg-function.

(invoke-restart restart arg*)

(invoke-restart-interactively restart)

▷ Call function associated with restart with arguments given or prompted for, respectively. If restart function returns, return its values.

(find-restart \{compute-restarts name \} \{condition\}

▷ Return innermost restart name, or a list of all restarts, respectively, out of those either associated with condition or un-associated at all; or, without condition, out of all restarts. Return \NIL if search is unsuccessful.

(restart-name restart)

▷ Name of restart.

\{abort \{muffle-warning \{continue \{store-value \{use-value \} \} \} \} \{condition \NIL\}

▷ Transfer control to innermost applicable restart with same name (i.e. abort . . . continue . . .) out of those either associated with condition or un-associated at all; or, without condition, out of all restarts. If no restart is found, signal control-error for abort and muffle-warning, or return \NIL for the rest.

(with-condition-restarts condition restarts form)

▷ Evaluate forms with restarts dynamically associated with condition. Return values of forms.

\{arithmetic-error-operation condition\}

\{arithmetic-error-operands condition\}

▷ List of function or of its operands respectively, used in the operation which caused condition.
(cell-error-name condition)  \[\text{Name of cell which caused condition.}\]
(unbound-slot-instance condition)  \[\text{Instance with unbound slot which caused condition.}\]
(print-not-readable-object condition)  \[\text{The object not readably printable under condition.}\]
(package-error-package condition)  \[\text{Package, path, or stream, respectively, which caused the condition of indicated type.}\]
(file-error-pathname condition)  \[\text{File path which caused file-error condition.}\]
(stream-error-stream condition)  \[\text{Stream which caused stream-error condition.}\]
(type-error-datum condition)  \[\text{Object which caused condition of type type-error.}\]
(type-error-expected-type condition)  \[\text{Object which caused condition of type type-error, or its expected type, respectively.}\]
(simple-condition-format-control condition)  \[\text{Return format control or list of format arguments, respectively, of condition.}\]

\[\text{Break on signals}\]  \[\text{Condition type debugger is to be invoked on.}\]
\[\text{Debugger hook}\]  \[\text{Function of condition and function itself. Called before debugger.}\]

12 Types and Classes

For any class, there is always a corresponding type of the same name.

(typep foo type [environment])  \[\text{T if foo is of type.}\]
(subtypep type-a type-b [environment])  \[\text{Return T if type-a is a recognizable subtype of type-b, and NIL if the relationship could not be determined.}\]

(type form)  \[\text{Declare values of form to be of type.}\]
(coerce object type)  \[\text{Coerce object into type.}\]

(typecase foo (type a-form)* [ (otherwise) b-form NIL ])  \[\text{Return values of the first a-form whose type is foo of. Return values of b-form if no type matches.}\]

\{m\typecase\}  \[\text{Return values of the first form whose type is foo of. Signal non-correctable/correctable type-error if no type matches.}\]

(type-of foo)  \[\text{Type of foo.}\]
(check-type place type [string])  \[\text{Signal correctable type-error if place is not of type. Return NIL.}\]
(stream-element-type stream)  \[\text{Type of stream objects.}\]
(array-element-type array)  \[\text{Element type array can hold.}\]
(upgraded-array-element-type type [environment])  \[\text{Element type of most specialized array capable of holding elements of type.}\]
Figure 2: Precedence Order of System Classes ( ), Classes ( ), Types ( ), and Condition Types ( ). Every type is also a supertype of NIL, the empty type.
(deftype foo (declare (macroλ ∗) [doc form\p])

⊲ Define type foo which when referenced as (foo arg) (or as foo if macroλ doesn’t contain any required parameters) applies expanded forms to args returning the new type. For (macroλ ∗) see page 19 but with default value of ∗ instead of NIL. forms are enclosed in an implicit `block named foo.

(eql foo)

⊲ Specifier for a type comprising foo or foos.

(member foo\*)

⊲ Type specifier for all objects satisfying predicate.

(mod n)

⊲ Type specifier for all non-negative integers < n.

(not type)

⊲ Complement of type.

(and type\* type\*)

⊲ Type specifier for intersection of types.

(or type\* type\*)

⊲ Type specifier for union of types.

(values type\* [optional type\* [rest other-args]])

⊲ Type specifier for multiple values.

● ⊲ As a type argument (cf. Figure 2): no restriction.

13 Input/Output

13.1 Predicates

(\streamp foo)

⊲ T if foo is of indicated type.

(\pathnamep foo)

(\readablep foo)

(\input-stream-p stream)

(\output-stream-p stream)

(\interactive-stream-p stream)

(\open-stream-p stream)

⊲ Return T if stream is for input, for output, interactive, or open, respectively.

(\pathname-match-p path wildcard)

⊲ T if path matches wildcard.

(\wild-pathname-p path [\{\:host\:device\:directory\:name\:type\:version\:NIL\}])

⊲ Return T if indicated component in path is wildcard. (NIL indicates any component.)

13.2 Reader

(\y-or-n-p)

{\yes-or-no-p}

[\control arg\*)

⊲ Ask user a question and return T or NIL depending on their answer. See page 38, f\format, for control and args.

(with-standard-io-syntax form\p)

⊲ Evaluate forms with standard behaviour of reader and printer. Return values of forms.

(\read [\read-preserving-whitespace]

[\stream=standard-input [eof-err \m]

[eof-valu\ext [recursive \ext]]])

⊲ Read printed representation of object.

(\read-from-string string [eof-err \m] [eof-valu\ext

[\{\:start \start\:end \end\:preserve-whitespace \bool\ext \}]]])

⊲ Return object read from string and zero-indexed position of next character.
(read-delimited-list char [stream standard-input nil])
   ▷ Continue reading until encountering char. Return list of
   objects read. Signal error if no char is found in stream.

(read-char [stream standard-input nil nil]
   [eof-error nil]
   [eof-val nil]
   [recursive nil])
   ▷ Return next character from stream.

(read-char-no-hang [stream standard-input nil nil]
   [eof-error T nil]
   [eof-val nil]
   [recursive nil])
   ▷ Next character from stream or NIL if none is available.

(peek-char [mode nil]
   [stream standard-input nil nil]
   [eof-error T nil]
   [eof-val nil]
   [recursive nil])
   ▷ Next, or if mode is T, next non-whitespace
   character, or if mode is a character, next instance of it, from
   stream without removing it there.

(unread-char character [stream standard-input])
   ▷ Put last read character back into stream; return NIL.

(read-byte [stream standard-input nil]
   [eof-error T nil]
   [recursive nil])
   ▷ Read next byte from binary stream.

(read-line [stream standard-input nil nil]
   [eof-error T nil]
   [recursive nil])
   ▷ Return a line of text from stream and T if line has been
   ended by end of file.

(read-sequence sequence [stream standard-input nil nil]
   [start start]
   [end end]
   [recursive nil])
   ▷ Replace elements of sequence between start and end
   with elements from binary or character stream. Return
   index of sequence’s first unmodified element.

(set-macro-character char function
   [non-term-p nil]
   [rt nil]
   [readtable standard]
   [recursive nil])
   ▷ Make char a macro character associated with
   function of stream and char. Return T.

(get-macro-character char
   [rt standard]
   [readtable nil]
   [recursive nil])
   ▷ Reader macro function associated with char, and T if char
   is a non-terminating macro character.

(make-dispatch-macro-character char
   [non-term-p nil]
   [rt nil]
   [readtable standard]
   [recursive nil])
   ▷ Make char a dispatching macro character. Return T.
(set-dispatch-macro-character char sub-char function [rt (readtable)])
▷ Make function of stream, n, sub-char a dispatch function of char followed by n, followed by sub-char. Return T.

(get-dispatch-macro-character char sub-char [rt (readtable)])
▷ Dispatch function associated with char followed by sub-char.

13.3 Character Syntax

#| multi-line-comment* | #
; one-line-comment*
▷ Comments. There are stylistic conventions:
;; title ▷ Short title for a block of code.
;; intro ▷ Description before a block of code.
;; state ▷ State of program or of following code.
;; explanation ▷ Regarding line on which it appears.
; continuation ▷

(foo* . bar nil) ▷ List of foos with the terminating cdr bar.

" ▷ Begin and end of a string.
'foo ▷ (quote foo); foo unevaluated.
'(foo [bar] [[@baz] [...quux] [bing]]) ▷ Backquote. quote foo and bing; evaluate bar and splice the lists baz and quux into their elements. When nested, outermost commas inside the innermost backquote expression belong to this backquote.

#\c ▷ (r character "c"), the character c.
#B n; #O n; #X n; #r R n ▷ Integer of radix 2, 8, 10, 16, or r; 2 ≤ r ≤ 36.

n/d ▷ The ratio n/d.

{[m].n[|S[F]D|L|E]|x} Les ▷ m.n·10^x as short-float, single-float, double-float, long-float, or the type from *read-default-float-format*.

#C(a b) ▷ (r complex a b), the complex number a + bi.
'#foo ▷ (function foo); the function named foo.

#nA sequence ▷ n-dimensional array.

#(n)foo* ▷ Vector of some (or n) foos filled with last foo if necessary.

#(n)b* ▷ Bit vector of some (or n) bs filled with last b if necessary.

#S(type {slot value}) ▷ Structure of type.

#P string ▷ A pathname.

#:foo ▷ Uninterned symbol foo.

#:form ▷ Read-time value of form.

*read-eval* ▷ If NIL, a reader-error is signalled at #..

#integer= foo ▷ Give foo the label integer.

#integer# ▷ Object labelled integer.

#< ▷ Have the reader signal reader-error.
Common Lisp Quick Reference

#+feature when-feature
#– feature unless-feature

▷ Means when-feature if feature is T; means unless-feature if feature is NIL. feature is a symbol from \#*features*, or \{and

\| or\} feature*), or \{not feature\).

\#*features*
▷ List of symbols denoting implementation-dependent features.

\|c\|; \c
▷ Treat arbitrary character(s) c as alphabetic preserving case.

### 13.4 Printer

\|prin1\| \{print \|pprint \|princ\}

▷ Print foo to stream \|readably, \|readably between a newline and a space, \|readably after a newline, or human-readable without any extra characters, respectively. \|prin1, \|print and \|princ return foo.

\|prin1-to-string foo\|
\|princ-to-string foo\|

▷ Print foo to \|string \|readably or human-readably, respectively.

\|print-object object stream\|

▷ Print object to stream. Called by the Lisp printer.

\|print-unreadable-object (foo \|stream \{\|start \|start \|end \|end \} form\|

▷ Enclosed in \#< and \#>, print foo by means of forms to stream. Return NIL.

\|terpri \|stream\|

▷ Output a newline to stream. Return NIL.

\|fresh-line \|stream\|

▷ Output a newline to stream and return T unless stream is already at the start of a line.

\|write-char char \|stream\|

▷ Output char to stream.

\|write-string \|write-line \|sequence stream \|

▷ Write string to stream without/with a trailing newline.

\|write-byte byte \|stream\|

▷ Write byte to binary stream.

\|write-sequence sequence stream\|

▷ Write elements of sequence to binary or character stream.
Print foo to stream and return foo, or print foo into string, respectively, after dynamically setting printer variables corresponding to keyword parameters (*print-bar* becoming :bar).

```
(defun pprint-linear (stream foo
  (declare (stream stream))
  (pprint-newline stream))
```

Print a conditional newline if stream is a pretty printing stream. Return NIL.

```
(defun *print-array*
  (declare (stream stream))
  (print-array stream))
```

If T, print arrays readably.

```
(defun *print-base*
  (declare (stream stream))
  (print-base stream 2))
```

Radix for printing rationals, from 2 to 36.
Common Lisp Quick Reference

*print-case
▷ Print symbol names all uppercase (*upcase*), all lowercase (*downcase*), capitalized (*capitalize*).

*print-circle
▷ If T, avoid indefinite recursion while printing circular structure.

*print-escape
▷ If NIL, do not print escape characters and package prefixes.

*print-gensym
▷ If T, print #: before uninterned symbols.

*print-length
*print-level
*print-lines
▷ If integer, restrict printing of objects to that number of elements per level/to that depth/to that number of lines.

*print-miser-width
▷ If integer and greater than the width available for printing a substructure, switch to the more compact miser style.

*print-pretty
▷ If T, print prettily.

*print-radix
▷ If T, print rationals with a radix indicator.

*print-readably
▷ If T, print readable or signal error print-not-readable.

*print-right-margin
▷ Right margin width in ems while pretty-printing.

(set-pprint-dispatch type function [priority] [table *print-pprint-dispatch*])
▷ Install entry comprising function of arguments stream and object to print; and priority as type into table. If function is NIL, remove type from table. Return NIL.

(pprint-dispatch foo [table *print-pprint-dispatch*])
▷ Return highest priority function associated with type of foo and T if there was a matching type specifier in table.

(copy-pprint-dispatch [table *print-pprint-dispatch*])
▷ Return copy of table or, if table is NIL, initial value of *print-pprint-dispatch*.

*print-pprint-dispatch
▷ Current pretty print dispatch table.

13.5 Format

(formatter control)
▷ Return function of stream and arg* applying *format to stream, control, and arg* returning NIL or any excess args.

(format [T]
          NIL out-string out-stream control arg*)
▷ Output string control which may contain * directives possibly taking some args. Alternatively, control can be a function returned by formatter which is then applied to out-stream and arg*. Output to out-string, out-stream or, if first argument is T, to *standard-output*. Return NIL. If first argument is NIL, return formatted output.

- [min-col] [col-inc] [min-pad] [pad-char]
  ○ Aesthetic/Standard. Print argument of any type for consumption by humans/by the reader, respectively. With :, print NIL as () rather than nil; with @, add pad-char on the left rather than on the right.

- [radix] [width] [pad-char] [comma-char] [comma-interval]
  ○ Radix. (With one or more prefix arguments.) Print argument as number; with :, group digits comma-interval each; with @, always prepend a sign.
\{ -R \} -R | -0R | -0R \}
\> Roman. Take argument as number and print it as Eng-
lish cardinal number, as English ordinal number, as Ro-
numeral, or as old Roman numeral, respectively.

\{ -width \} [\{ *pad-char \} | \{ *comma-char \}]
\> Decimal/Binary/Octal/Hexadecimal. Print integer ar-
gument as number. With \$, group digits comma-interval
each; with \$, always prepend a sign.

\{ -width \} [\{ dec-digits \} | \{ shift \} | \{ *overflow-char \}]
\> Fixed-Format Floating-Point. With \$, always prepend
a sign.

\{ -width \} [\{ dec-digits \} | \{ exp-digits \} | \{ scale-factor \}]
\> Exponential/General Floating-Point. Print argument as
floating-point number with dec-digits after decimal point
and exp-digits in the signed exponent. With \-G, choose
either \-E or \-F. With \$, always prepend a sign.

\{ -width \} [\{ int-digits \} | \{ width \} | \{ *pad-char \}]
\> Monetary Floating-Point. Print argument as fixed-

\{ -text \} [\{ -text \} | \$(text \-text) | \$](text \-text)
\> Case-Conversion. Convert text to lowercase, convert
first letter of each word to uppercase, capitalize first word
and convert the rest to lowercase, or convert to uppercase,
respectively.

\{ -P \} -P | -0P | -0P \}
\> Plural. If argument eql 1 print nothing, otherwise print
\$; do the same for the previous argument; if argument eql
1 print \$, otherwise print \$; do the same for the previous
argument, respectively.

\{ -n \} % \> Newline. Print n newlines.

\{ -n \} & \> Fresh-Line. Print n \n newlines if output stream is at
the beginning of a line, or n newlines otherwise.

\{ -\} -0 \} -0 \}
\> Conditional Newline. Print a newline like
\pprint-newline with argument :linear, :fill, :miser,
or :mandatory, respectively.

\{ -\} -0 \} -0 \}
\> Ignored Newline. Ignore newline, or whitespace follow-

\{ -n \} | \> Page. Print n page separators.

\{ -n \} \> Tilde. Print n tildes.

\{ [min-co\} | \{ min-pec \} | [\{ *pad-char \}]
\> Justification. Justify text produced by texts in a field
of at least min-co columns. With \$, right justify; with \$,
left justify. If this would leave less than spare characters
on the current line, output nl-text first.

\{ *[prefix] \} \{ per-line-prefix -[\} \}
\> Logical Block. Act like \pprint-logical-block using body
as format control string on the elements of the list argument
or, with \$, on the remaining arguments, which are
extracted by \pprint-pop. With \$, prefix and suffix default
to ( and ). When closed by -0\$, spaces in body
are replaced with conditional newlines.
13.6 Streams

\[
- \{ [n] \texttt{i} \} \text{ - } [m] \texttt{d}
\]

\textbf{Indent.} Set indentation to \( n \) relative to leftmost/to current position.

\[
\{ [d] \texttt{i} \} \text{ - } [d] \texttt{d} \text{ T}
\]

\textbf{Tabulate.} Move cursor forward to column number \( c + ki \), \( k \geq 0 \) being as small as possible. With \( ; \), calculate column numbers relative to the immediately enclosing section. With \( \texttt{0} \), move to column number \( c_0 + c + ki \) where \( c_0 \) is the current position.

\[
- \{ [n] \} \text{ - } [m] \texttt{d}
\]

\textbf{Go-To.} Jump \( m \) arguments forward, or backward, or to argument \( n \).

\[
- \{ \texttt{limit} \} \text{ [ ] } \texttt{O} \} \text{ { text } - }
\]

\textbf{Iteration.} Use \( \texttt{text} \) repeatedly, up to \( \texttt{limit} \), as control string for the elements of the list argument or (with \( \texttt{O} \)) for the remaining arguments. With \( ; \) or \( \texttt{0} \), list elements or remaining arguments should be lists of which a new one is used at each iteration step.

\[
- \{ [x] \texttt{y} \texttt{z} \}
\]

\textbf{Escape Upward.} Leave immediately \( \langle - \rangle \), \( \langle - : \rangle \), \( \langle - @ \rangle \), or the entire \( \langle \texttt{format} \rangle \) operation. With one to three prefixes, act only if \( x = 0 \), \( x \neq y \), or \( y \leq y \), respectively.

\[
- \{ [i] \texttt{o} \} \text{ [ (text - ) \} text \texttt{z} \texttt{default}\ - ]
\]

\textbf{Conditional Expression.} Use the zero-indexed argument (or \( \texttt{ith} \) if given) \( \texttt{text} \) as a \( \texttt{format} \) control subclause. With \( ; \), use the first text if the argument value is \( \texttt{NIL} \), or the second \( \texttt{text} \) if it is \( T \). With \( \texttt{0} \), do nothing for an argument value of \( \texttt{NIL} \). Use the only \( \texttt{text} \) and leave the argument to be read again if it is \( T \).

\[
- \{ [t] \texttt{O} \}
\]

\textbf{Recursive Processing.} Process two arguments as control string and argument list, or take one argument as control string and use then the rest of the original arguments.

\[
- \{ \texttt{prefix} \} \texttt{y} \texttt{z} \texttt{i} \texttt{O} \} \text{ [package \texttt{z} \texttt{t} function/ ]
\]

\textbf{Call Function.} Call all-uppercase \( \texttt{package:function} \) with the arguments stream, \( \texttt{format}-\texttt{argument} \), colon-p, at-sign-p and \( \texttt{prefix} \) for printing \( \texttt{format}-\texttt{argument} \).

\[
- \{ [i] \texttt{O} \} \texttt{W}
\]

\textbf{Write.} Print argument of any type obeying every printer control variable. With \( ; \), pretty-print. With \( \texttt{O} \), print without limits on length or depth.

\[
\{ V \# \}
\]

\textbf{In place of the comma-separated prefix parameters:} use next argument or number of remaining unprocessed arguments, respectively.

13.6 Streams

- \texttt{input}
- \texttt{output}
- \texttt{io}
- \texttt{probe}
- \texttt{external-format}
- \texttt{file-stream}
- \texttt{to path}

- \texttt{direction}
- \texttt{type}
- \texttt{new-version}
- \texttt{error}
- \texttt{rename}
- \texttt{append}
- \texttt{supersede}
- \texttt{create}
- \texttt{error}
- \texttt{NIL}
- \texttt{function/}
- \texttt{package/

\textbf{Open path}
(\texttt{make-concatenated-stream input-stream\textsuperscript{*}})
\hspace*{1em}\triangleright \text{Return stream of indicated type.}

(\texttt{make-broadcast-stream output-stream\textsuperscript{*}})
\hspace*{1em}\triangleright \text{Return list of streams broadcast-stream still has to read from/broadcast-stream is broadcasting to.}

(\texttt{make-two-way-stream input-stream-part output-stream-part})
\hspace*{1em}\triangleright \text{Return source stream or sink stream of two-way-stream, respectively.}

(\texttt{make-echo-stream from-input-stream to-output-stream})
\hspace*{1em}\triangleright \text{Return symbol of synonym-stream.}

(\texttt{make-synonym-stream variable-bound-to-stream})
\hspace*{1em}\triangleright \text{Return stream of indicated type.}

(\texttt{make-string-input-stream string \texttt{[start end]}})
\hspace*{1em}\triangleright \text{Return a string-stream supplying the characters from string.}

(\texttt{make-string-output-stream \texttt{[:element-type type character]}})
\hspace*{1em}\triangleright \text{Return a string-stream accepting characters (available via \texttt{get-output-stream-string}).}

(\texttt{concatenated-stream-streams concatenated-stream})
\hspace*{1em}\triangleright \text{Return source stream or sink stream of concatenated-stream still has to read from/concatenated-stream is broadcasting to.}

(\texttt{two-way-stream-input-stream two-way-stream})
\hspace*{1em}\triangleright \text{Return source stream or sink stream of two-way-stream, respectively.}

(\texttt{two-way-stream-output-stream two-way-stream})
\hspace*{1em}\triangleright \text{Return source stream or sink stream of two-way-stream, respectively.}

(\texttt{echo-stream-input-stream echo-stream})
\hspace*{1em}\triangleright \text{Return source stream or sink stream of two-way-stream/echo-stream, respectively.}

(\texttt{echo-stream-output-stream echo-stream})
\hspace*{1em}\triangleright \text{Return source stream or sink stream of two-way-stream/echo-stream, respectively.}

(\texttt{listen [\texttt{stream standard-input}]})
\hspace*{1em}\triangleright \text{T if there is a character in input stream.}

(\texttt{clear-input \texttt{[\texttt{stream standard-input}]}})
\hspace*{1em}\triangleright \text{Clear input from stream, return NIL.}

(\texttt{clear-output \texttt{[\texttt{stream standard-output}]}})
\hspace*{1em}\triangleright \text{End output to stream and return NIL immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.}

(\texttt{close \texttt{[\texttt{stream abort boolean}]}})
\hspace*{1em}\triangleright \text{Close stream. Return T if stream had been open. If :abort is T, delete associated file.}

(\texttt{with-open-file \texttt{(stream path open-arg\textsuperscript{*}) (declare decl\textsuperscript{*}) form\textsuperscript{*}}})
\hspace*{1em}\triangleright \text{Use \texttt{\texttt{open}} with open-args to temporarily create stream to path; return values of forms.}

(\texttt{with-open-stream \texttt{(foo stream\textsuperscript{*}) (declare decl\textsuperscript{*}) form\textsuperscript{*}})
\hspace*{1em}\triangleright \text{Evaluate forms with foo locally bound to stream. Return values of forms.}

(\texttt{with-input-from-string \texttt{(foo string \texttt{[:index index :start startindex :end endindex]}) (declare decl\textsuperscript{*}) form\textsuperscript{*}})
\hspace*{1em}\triangleright \text{Evaluate forms with foo locally bound to input string-stream from string. Return values of forms; store next reading position into index.}
13.7 Pathnames and Files

(pathname-host
pathname-device
(pathname-name
(pathname-type

(pathname-version path-or-stream)

(pathname) path-or-stream [case :local :common :local]

(pathname-host
(printname-device
(pathname-name
(pathname-type

(with-output-to-string (foo [string [element-type type [unspecified]]] (declare decl*+* form+))

(stream-external-format stream)

*terminal-io*  ➤ Bidirectional stream to user terminal.
*standard-input*  ➤ Standard input stream.
*standard-output*  ➤ Standard output stream.
*error-output*  ➤ Error output stream.
*debug-io*  ➤ Bidirectional streams for debugging and user interaction.
*query-io*  ➤ Bidirectional streams for debugging and user interaction.

Construct a logical pathname if there is a logical pathname for host, otherwise construct a physical pathname. For case :local, leave case of components unchanged. For case :common, leave mixed-case components unchanged; convert all-uppercase components into local customary case; do the opposite with all-lowercase components.

Return pathname component.

Return pathname converted from string, pathname, or stream foo; and position where parsing stopped.

Return pathname made by filling in components missing in path-or-stream from default-path-or-stream.

Pathname to use if one is needed and none supplied.

User’s home directory.
(enough-namestring path-or-stream
 [root-path/default pathname-defaults])
▷ Return minimal path string that sufficiently describes the path
 of path-or-stream relative to root-path.

(namestring path-or-stream)

(file-namestring path-or-stream)

(directory-namestring path-or-stream)

(host-namestring path-or-stream)
▷ Return string representing full pathname; name, type,
 and version; directory name; or host name, respectively, of
 path-or-stream.

(translate-pathname path-or-stream wildcard-path-a wildcard-path-b)
▷ Translate the path of path-or-stream from wildcard-path-a
 into wildcard-path-b. Return new path.

(pathname path-or-stream)
▷ Pathname of path-or-stream.

(logical-pathname logical-path-or-stream)
▷ Logical pathname of logical-path-or-stream. Logical
 pathnames are represented as all-uppercase
 "([host]:;[;]{dir}*{name}*{type}*{version}.*\n [.version]*\n [newest[NEWEST]]).*

(logical-pathname-translations logical-host)
▷ List of (from-wildcard to-wildcard) translations for
 logical-host. setfable.

(load-logical-pathname-translations logical-host)
▷ Load logical-host’s translations. Return NIL if already
 loaded; return T if successful.

(translate-logical-pathname path-or-stream)
▷ Physical pathname corresponding to (possibly logical)
 pathname of path-or-stream.

(probe-file file)

(truename file)
▷ Canonical name of file. If file does not exist, return
 NIL/signal file-error, respectively.

(file-write-date file)
▷ Time at which file was last written.

(file-author file)
▷ Return name of file owner.

(file-length stream)
▷ Return length of stream.

(rename-file foo bar)
▷ Rename file foo to bar. Unspecified components of path bar
 default to those of foo. Return new pathname, old physical
 file name, and new physical file name.

(delete-file file)
▷ Delete file. Return T.

(directory path)
▷ List of pathnames matching path.

(ensure-directories-exist path [verbose bool])
▷ Create parts of path if necessary. Second return value is T
 if something has been created.

14 Packages and Symbols

The Loop Facility provides additional means of symbol handling; see
loop, page 22.

14.1 Predicates

(symbolp foo)

(packagep foo)
▷ T if foo is of indicated type.

(keywordp foo)
14.2 Packages

`:bar` keyword: `bar`  ▶ Keyword, evaluates to `:bar`.

`package::symbol` ▶ Exported symbol of package.

`package::symbol` ▶ Possibly unexported symbol of package.

```
((nicknames nick*)
 (documentation string)
 (intern interned-symbol*)
 (use used-package*)
 )
```

`defpackage foo` ▶ Create or modify package `foo` with internal-symbols, symbols from used-packages, imported-symbols, and shd-symbols. Add shd-symbols to `foo`’s shadowing list.

```
(make-package foo
 ( nickname ( nick* env )
 )
 ▶ Create package `foo`.
```


```
(in-package foo) ▶ Make package `foo` current.
```

```
(use-package other-packages |package env|
 (un-use-package)
 ▶ Make exported symbols of other-packages available in package, or remove them from package, respectively. Return `T`.
```

```
(package-use-list package)
 (package-used-by-list package)
 ▶ List of other packages used by/using package.
```

```
(delete-package package)
 ▶ Delete package. Return `T` if successful.
```

```
*package* [common-lisp use]
 ▶ The current package.
```

```
(list-all-packages)
 ▶ List of registered packages.
```

```
(package-name package)
 ▶ Name of package.
```

```
(package-nicknames package)
 ▶ Nicknames of package.
```

```
(find-package name)
 ▶ Package with name (case-sensitive).
```

```
(find-all-symbols foo)
 ▶ List of symbols `foo` from all registered packages.
```

```
(intern |find-symbol| foo |package env|
 ▶ Intern or find, respectively, symbol `foo` in package. Second return value is one of `internal`, `external`, `inherited` (or `NIL` if `intern` has created a fresh symbol).
```

```
(unintern symbol |package env|
 ▶ Remove symbol from package, return `T` on success.
```

```
(import |shadowing-import| symbols |package env|
 ▶ Make symbols internal to package. Return `T`. In case of a name conflict signal correctable `package-error` or shadow the old symbol, respectively.
```

```
(shadow symbols |package env|
 ▶ Make symbols of package shadow any otherwise accessible, equally named symbols from other packages. Return `T`.
```

---

Common Lisp Quick Reference
Common Lisp Quick Reference

(package-shadowing-symbols package)
▷ List of symbols of package that shadow any otherwise accessible, equally named symbols from other packages.

(export symbols [package])
▷ Make symbols external to package. Return T.

(unexport symbols [package])
▷ Revert symbols to internal status. Return T.

(do-symbols (var)
  (declare decl)
  (form)
  ▷ Evaluate tagbody-like body with var successively bound to every symbol from package, to every external symbol from package, or to every symbol from all registered packages, respectively. Return values of result. Implicitly, the whole form is a block named NIL.

(with-package-iterator (foo packages)
  (declare decl)
  (form)
  ▷ Return values of form s. In form s, successive invocations of (foo) return: T if a symbol is returned; a symbol from packages; accessibility (internal, external, or inherited); and the package the symbol belongs to.

(require module [paths])
▷ If not in *modules*, try paths to load module from. Signal error if unsuccessful. Deprecated.

(provide module)
▷ If not already there, add module to *modules*. Deprecated.

*modules*  ▷ List of names of loaded modules.

14.3 Symbols

A symbol has the attributes name, home package, property list, and optionally value (of global constant or variable name) and function (function, macro, or special operator name).

(make-symbol name)
▷ Make fresh, uninterned symbol name.

gensym
▷ Return fresh, uninterned symbol #:sn with n from *gensym-counter*. Increment *gensym-counter*.

gentemp [prefix] [package]
▷ Intern fresh symbol in package. Deprecated.

copy-symbol [symbol]
▷ Return uninterned copy of symbol. If props is T, give copy the same value, function and property list.

(symbol-name symbol)

(symbol-package symbol)

(symbol-plist symbol)

(symbol-value symbol)

(symbol-function symbol)
▷ Name, package, property list, value, or function, respectively, of symbol. setfable.

documentation

(setf documentation) new-doc
▷ Get/set documentation string of foo of given type.
\( \top \) 
\( \) Truth; the supertype of every type including \( t \); the superclass of every class except \( t \); \( \ast \)terminal-io\.

\( \bot \) 
\( \) Falsity; the empty list; the empty type, subtype of every type; \( \ast \)standard-input\; \( \ast \)standard-output\; the global environment.

### 14.4 Standard Packages

**common-lisp**

\( \) Exports the defined names of Common Lisp except for those in the **keyword** package.

**common-lisp-user**

\( \) Current package after startup; uses package **common-lisp**.

**keyword**

\( \) Contains symbols which are defined to be of type **keyword**.

### 15 Compiler

#### 15.1 Predicates

\( (\text{special-operator-p} \ foo) \) 
\( \top \) if \( foo \) is a special operator.

\( (\text{compiled-function-p} \ foo) \) 
\( \top \) if \( foo \) is of type **compiled-function**.

#### 15.2 Compilation

\( (\text{compile} \ \{ \text{NIL definition} \ (\text{name} (\text{setf name}) \ [\text{definition}] \}) \) 
\( \) Return compiled function or replace \text{name}'s function definition with the compiled function. Return \( \top \) in case of \text{warnings} or \text{errors}, and \( \top \) in case of \text{warnings} or \text{errors} excluding style-warning\.

\( (\text{compile-file} \ \{ \text{file} \ (\text{:output-file out-path} (\text{:verbose bool}) (\text{:print bool}) (\text{:external-format file-format}) \}) \) 
\( \) Write compiled contents of \text{file} to \text{out-path}. Return true output path or \text{NIL} \( \top \) in case of \text{warnings} or \text{errors}, \( \top \) in case of \text{warnings} or \text{errors} excluding style-warning\.

\( (\text{compile-file-pathname} \ \{ \text{file} \ (\text{:output-file path}) \}) \) 
\( \) Pathname **compile-file** writes to if invoked with the same arguments.

\( (\text{load} \ \{ \text{path} \ (\text{:verbose bool}) (\text{:print bool}) (\text{:if-does-not-exist bool}) (\text{:external-format file-format}) \}) \) 
\( \) Load source file or compiled file into Lisp environment. Return \( \top \) if successful.

\( \ast \text{compile-file} \) \( \) Input file used by **compile-file**/by **load**.

\( \ast \text{load} \) 
\( \) Defaults used by **compile-file**/by **load**.
(eval-when \(\{\text{compile-toplevel}\} \text{load-toplevel}\) \{execute\}) form\textsuperscript{P}

- Return values of forms if eval-when is in the top-level of a file being compiled, in the top-level of a compiled file being loaded, or anywhere, respectively. Return NIL if forms are not evaluated. (compile, load and eval deprecated.)

(\text{locally (declare decl\textsuperscript{*})} form\textsuperscript{P})

- Evaluate forms in a lexical environment with declarations decl in effect. Return values of forms.

(with-compilation-unit \(\{\text{override bool}\}\) form\textsuperscript{P})

- Return values of forms. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of forms.

(load-time-value form \(\{\text{read-only NIL}\}\))

- Evaluate form at compile time and treat its value as literal at run time.

(quote foo)

- Return unevaluated foo.

(make-load-form foo \{environment\})

- Its methods are to return a creation form which on evaluation at load time returns an object equivalent to foo, and an optional initialization form which on evaluation performs some initialization of the object.

(make-load-form-saving-slots foo \{\text{slot-names slots \text{NIL \text{NIL \text{NIL}}}}\} \{environment \text{environment}\})

- Return a creation form and an initialization form which on evaluation construct an object equivalent to foo with slots initialized with the corresponding values from foo.

(macro-function symbol \{environment\})

- Return specified macro function, or compiler macro function, respectively, if any. Return NIL otherwise. setfable.

(evaluate arg)

- Return values of value of arg evaluated in global environment.

15.3 REPL and Debugging

+|++|++++|
+*|**|****|
+// |/// |

- Last, penultimate, or antepenultimate form evaluated in the REPL, or their respective primary value, or a list of their respective values.

v-

- Form currently being evaluated by the REPL.

(apropos string \{package\})

- Print interned symbols containing string.

(apropos-list string \{package\})

- List of interned symbols containing string.

(dribble path)

- Save a record of interactive session to file at path. Without path, close that file.

(ed \{file-or-function\})

- Invoke editor if possible.

\{macroexpand-1\} \text{form} \{environment\})

- Return macro expansion, once or entirely, respectively, of form and T if form was a macro form. Return form and NIL otherwise.
\texttt{\*macroexpand-hook*}  
\> Function of arguments expansion function, macro form, and environment called by \texttt{\*macroexpand-1} to generate macro expansions.

\texttt{(mtrace \{function \{setf function\}\} \*)}  
\> Cause functions to be traced. With no arguments, return list of traced functions.

\texttt{(muntrace \{function \{setf function\}\} \*)}  
\> Stop functions, or each currently traced function, from being traced.

\texttt{\*trace-output*}  
\> Output stream \texttt{mtrace} and \texttt{mtime} send their output to.

\texttt{(mstep form)}  
\> Step through evaluation of \texttt{form}. Return values of \texttt{form}.

\texttt{(break \{control arg\} \* \*)}  
\> Jump directly into debugger; return \texttt{NIL}. See page 38, \texttt{fformat}, for control and arg.s.

\texttt{(mtime form)}  
\> Evaluate \texttt{form}s and print timing information to \texttt{\*trace-output*}. Return values of \texttt{form}.

\texttt{(\*inspect foo)}  
\> Interactively give information about \texttt{foo}.

\texttt{(\*describe foo \{stream \{\*standard-output*\}\})}  
\> Send information about \texttt{foo} to \texttt{stream}.

\texttt{(\*describe-object foo \{stream\})}  
\> Send information about \texttt{foo} to \texttt{stream}. Called by \texttt{\*describe}.  

\texttt{(\*disassemble function)}  
\> Send disassembled representation of \texttt{function} to \texttt{\*standard-output*}. Return \texttt{NIL}.

\texttt{(\*room \{\{NIL\} \{default T\}\} \{\*room\})}  
\> Print information about internal storage management to \texttt{\*standard-output*}.

### 15.4 Declarations

\texttt{(\*proclaim decl)}  
\> Globally make declaration(s) \texttt{decl}. \texttt{decl} can be: \texttt{declaration}, \texttt{type}, \texttt{ftype}, \texttt{inline}, \texttt{notinline}, \texttt{optimize}, or \texttt{special}. See below.

\texttt{(declare decl\* \* \*)}  
\> Inside certain forms, locally make declarations \texttt{decl\*}. \texttt{decl} can be: \texttt{dynamic-extent}, \texttt{type}, \texttt{ftype}, \texttt{ignore}, \texttt{ignore}, \texttt{inline}, \texttt{notinline}, \texttt{optimize}, or \texttt{special}. See below.

\texttt{(declaration foo\* \*)}  
\> Make \texttt{foos} names of declarations.

\texttt{(\{\*dynamic-extent variable\* \{function function\}\} \*)}  
\> Declare lifetime of \texttt{variables} and/or \texttt{functions} to end when control leaves enclosing block.

\texttt{(\{\*type variable\*\}) \* \*)  
\> Declare \texttt{variables} or \texttt{functions} to be of \texttt{type}.

\texttt{\{\*ignore \} \{\*var \{\*function function\}\} \*)}  
\> Suppress warnings about \texttt{used/unused bindings}.

\texttt{(\{\*inline function\*\}) \* \*)}  
\> Tell compiler to integrate/not to integrate, respectively, called \texttt{functions} into the calling routine.
(optimize (compilation-speed n) (debug n) (safety n) (space n) (speed n))

▷ Tell compiler how to optimize. n = 0 means unimportant, n = 1 is neutral, n = 3 means important.

(special var*) ▷ Declare vars to be dynamic.

16 External Environment

(/get-internal-real-time)
(/get-internal-run-time)
  ▷ Current time, or computing time, respectively, in clock ticks.

/internal-time-units-per-second
  ▷ Number of clock ticks per second.

(/encode-universal-time sec min hour date month year [zone curr])
(/get-universal-time)
  ▷ Seconds from 1900-01-01, 00:00, ignoring leap seconds.

(/decode-universal-time universal-time [time-zone current])
(/get-decoded-time)
  ▷ Return second, minute, hour, date, month, year, day, daylight-p, and zone.

(/short-site-name)
(/long-site-name)
  ▷ String representing physical location of computer.

(/lisp-implementation)
(/software)
(/machine)
  ▷ Name or version of implementation, operating system, or hardware, respectively.

(/machine-instance)
  ▷ Computer name.