Quick Reference

cl

Common

c

lisp

Bert Burgemeister
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; _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.

[f00|b00] ▷ Either one f00 or nothing; defaults to b00.
f00*; {f00}\* ▷ Zero or more f00s.
f00*; {f00}+ ▷ One or more f00s.
foos ▷ English plural denotes a list argument.

{f00|b00|b00} ▷ Either f00, or bar, or baz.

f00 ; b00 ; b00 ▷ Anything from none to each of f00, bar, and baz.
f00 ▷ Argument f00 is not evaluated.
bar ▷ Argument bar is possibly modified.
f00^b ▷ f00* is evaluated as in ^progn; see page 21.
f00; bar; baz ▷ Primary, secondary, and nth return value.
T; NIL ▷ t, or truth in general; and nil or ().
1 Numbers

1.1 Predicates

\( (\text{=}\ \text{number}^+) \)  \( \triangleright \) \( \top \) if all \text{numbers}, or none, respectively, are equal in value.

\( (\text{>\ number}^+) \)  \( (\text{>=\ number}^+) \)  \( (\text{<\ number}^+) \)  \( (\text{<=\ number}^+) \)  \( \triangleright \) Return \( \top \) if \text{numbers} are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

\( (\text{minusp} \ a) \)  \( (\text{zerop} \ a) \)  \( (\text{plusp} \ a) \)  \( (\text{evenp} \ \text{int}) \)  \( (\text{oddp} \ \text{int}) \)  \( \triangleright \top \) if \( a < 0 \), \( a = 0 \), or \( a > 0 \), respectively.

\( (\text{numberp} \ \text{foo}) \)  \( (\text{realp} \ \text{foo}) \)  \( (\text{rationalp} \ \text{foo}) \)  \( (\text{floatp} \ \text{foo}) \)  \( (\text{rationalp} \ \text{foo}) \)  \( (\text{integerp} \ \text{foo}) \)  \( (\text{complexp} \ \text{foo}) \)  \( (\text{random-state-p}\ \text{foo}) \)  \( \triangleright \top \) if \( \text{foo} \) is of indicated type.

1.2 Numeric Functions

\( (\text{+} a 0) \)  \( (\text{\*} a 1) \)  \( \triangleright \sum a \) or \( \prod a \), respectively.

\( (\text{-} a b) \)  \( (\text{/} a b) \)  \( \triangleright \) Return \( a - \sum b \) or \( a / \prod b \), respectively. Without any \( b \)s, return \( -a \) or \( 1/a \), respectively.

\( (1\text{+} a) \)  \( (1\text{-} a) \)  \( \triangleright \) Return \( a + 1 \) or \( a - 1 \), respectively.

\( \{\text{incf} \ \text{place} | \text{delta}\} \)  \( \triangleright \) Increment or decrement the value of \( \text{place} \) by \( \text{delta} \). Return new value.

\( (\text{exp} \ p) \)  \( (\text{expt} \ b \ p) \)  \( \triangleright \) Return \( e^p \) or \( b^p \), respectively.

\( (\log \ a [\text{base}]) \)  \( \triangleright \) Return \( \log_{\text{base}} a \) or, without \( \text{base} \), \( \ln a \).

\( (\sqrt{\text{int}} n) \)  \( (\text{isqrt} \ n) \)  \( \sqrt{n} \) in complex numbers/natural numbers.

\( (\text{lcm} \ \text{integer}* 1) \)  \( (\text{gcd} \ \text{integer}* 1) \)  \( \triangleright \) Least common multiple or greatest common denominator, respectively, of \text{integers}. \( (\text{gcd}) \) returns 0.

\( \pi \)  \( \triangleright \) long-float approximation of \( \pi \), Ludolph’s number.

\( (\sin \ a) \)  \( (\cos \ a) \)  \( (\tan \ a) \)  \( \text{\sin} a, \text{cos} a, \text{tan} a \), respectively. \( a \) in radians.

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

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

\(\text{(f cosh } a)\) \(\triangleright\) \(\sinh a\), \(\cosh a\), or \(\tanh a\), respectively.

\(\text{(f tanh } a)\) \(\triangleright\) \(\sinh a\), \(\cosh a\), or \(\tanh a\), respectively.

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

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

\(\text{(f max num}^+\text{)}\) \(\triangleright\) Greatest or least, respectively, of \(\text{num}\).

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

\(\text{(f mod n d)}\) \(\triangleright\) Same as \(\text{f floor}\) or \(\text{f truncate}\), respectively, but return remainder only.

\(\text{(f random limit }\text{[state]}^\ast\text{random-state)}\) \(\triangleright\) Return non-negative random number less than \(\text{limit}\), and of the same type.

\(\text{(f make-random-state }\text{[state]}^\ast\text{random-state)}\) \(\triangleright\) Copy of \(\text{random-state}\) object \(\text{state}\) or of the current random state; or a randomly initialized fresh random state.

\(\text{v \ast random-state}\) \(\triangleright\) Current random state.

\(\text{(f float-sign num-}a\text{[num-}b\text{]}\ast\text{)}\) \(\triangleright\) \(\text{num-}b\) with \(\text{num-}a\)’s sign.

\(\text{(f signum }n)\) \(\triangleright\) Number of magnitude 1 representing sign or phase of \(n\).

\(\text{(f numerator rational)}\) \(\triangleright\) Numerator or denominator, respectively, of \(\text{rational}\)’s canonical form.

\(\text{(f realpart number)}\) \(\triangleright\) Real part or imaginary part, respectively, of \(\text{number}\).

\(\text{(f imagpart number)}\) \(\triangleright\) Make a complex number.

\(\text{(f phase num)}\) \(\triangleright\) Angle of \(\text{num}\)’s polar representation.

\(\text{(f abs }n)\) \(\triangleright\) Return \(|n|\).

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

\(\text{(f float real }\text{[prototype]}\ast\text{)}\) \(\triangleright\) Convert \(\text{real}\) into \(\text{float}\) with type of \(\text{prototype}\).
1.3 Logic Functions

Negative integers are used in two’s complement representation.

(defun boole (operation int-a int-b)
  "Return value of bitwise logical operation. operations are"
  (if operation int-a int-b)
  (bmeta-1) ⇒ int-a.
  (bmeta-2) ⇒ int-b.
  (bmeta-c1) ⇒ ¬int-a.
  (bmeta-c2) ⇒ ¬int-b.
  (bmeta-set) ⇒ All bits set.
  (bmeta-cfr) ⇒ All bits zero.
  (bmeta-eqv) ⇒ int-a ≡ int-b.
  (bmeta-and) ⇒ int-a ∧ int-b.
  (bmeta-andc1) ⇒ ¬int-a ∧ int-b.
  (bmeta-andc2) ⇒ int-a ∧ ¬int-b.
  (bmeta-nand) ⇒ ¬(int-a ∧ int-b).
  (bmeta-ior) ⇒ int-a ∨ int-b.
  (bmeta-orc1) ⇒ ¬int-a ∨ int-b.
  (bmeta-orc2) ⇒ int-a ∨ ¬int-b.
  (bmeta-xor) ⇒ ¬(int-a ≡ int-b).
  (bmeta-nor) ⇒ ¬(int-a ∨ int-b).

(defun lognot (integer)
  "Return value of exclusive-nored anded integer, respectively. Without any integer, return -1."
  (if integer
    ¬integer.
    (lognot integer)

(defun logeqv (integer)
  "Return value of exclusive-nored anded integer, respectively. Without any integer, return -1."
  (if integer
    ¬integer.
    (logeqv integer)

(defun logand (integer)
  "Return value of exclusive-nored anded integer, respectively. Without any integer, return 0."
  (if integer
    ¬integer.
    (logand integer)

(defun logandc1 (int-a int-b)
  ⇒ ¬int-a ∧ int-b.

(defun logandc2 (int-a int-b)
  ⇒ int-a ∧ ¬int-b.

(defun lognand (int-a int-b)
  ⇒ ¬(int-a ∧ int-b).

(defun logxor (integer)
  "Return value of exclusive-ored or ored integer, respectively. Without any integer, return 0."
  (if integer
    ¬integer.
    (logxor integer)

(defun logior (integer)
  "Return value of exclusive-ored or ored integer, respectively. Without any integer, return 0."
  (if integer
    ¬integer.
    (logior integer)

(defun logbitp (i int)
  ⇒ T if zero-indexed i-th bit of int is set.

(defun logtest (int-a int-b)
  "Return T if there is any bit set in int-a which is set in int-b as well."
  (if integer
    ¬integer.
    (logtest int-a int-b)

(defun logcount (int)
  "Number of 1 bits in int ≥ 0, number of 0 bits in int < 0."
  (if integer
    ¬integer.
    (logcount int)
1.4 Integer Functions

- (f integer-length integer) ▷ Number of bits necessary to represent integer.
- (f ldb-test byte-spec integer) ▷ Return T if any bit specified by byte-spec in integer is set.
- (f 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.
- (f ldb byte-spec integer) ▷ Extract byte denoted by byte-spec from integer. setfable.
- (f 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 (f byte-size byte-spec) bits of int-a, respectively.
- (f mask-field byte-spec integer) ▷ Return copy of integer with all bits unset but those denoted by byte-spec. setfable.
- (f byte size position) ▷ Byte specifier for a byte of size bits starting at a weight of position.
- (f byte-size byte-spec) ▷ Size or position, respectively, of byte-spec.

1.5 Implementation-Dependent

- short-float single-float double-float long-float
  ▷ Smallest possible number making a difference when added or subtracted, respectively.
- least-negative least-negative-normalized least-positive least-positive-normalized
  ▷ Available numbers closest to −0 or +0, respectively.
- most-negative most-positive
  ▷ Available numbers closest to −∞ or +∞, respectively.
- (f decode-float n) ▷ Return significand, exponent, and sign of float n.
- (f integer-decode-float n) ▷ 
- (f scale-float n [i]) ▷ With n’s radix b, return nb^i.
- (f float-radix n) ▷ 
- (f float-digits n) ▷ 
- (f float-precision n) ▷ Radix, number of digits in that radix, or precision in that radix, respectively, of float n.
- (f upgraded-complex-part-type foo [environment))] ▷ 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)`
- `(graphic-char-p character)`
- `(alpha-char-p character)`
- `(alphanumericp character)`  ▶ T if character is visible, alphabetic, or alphanumeric, respectively.
- `(upper-case-p character)`
- `(lower-case-p character)`
- `(both-case-p character)`  ▶ Return T if character is uppercase, lowercase, or able to be in another case, respectively.
- `(digit-char-p character [radix])`  ▶ Return its weight if character is a digit, or NIL otherwise.
- `(char= character)`
- `(char/= character)`  ▶ Return T if all characters, or none, respectively, are equal.
- `(char-equal character)`
- `(char-not-equal character)`  ▶ Return T if all characters, or none, respectively, are equal ignoring case.
- `(char> character)`
- `(char>= character)`
- `(char< character)`
- `(char<= character)`  ▶ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.
- `(char-greaterp character)`
- `(char-not-lessp character)`
- `(char-lessp character)`
- `(char-not-greaterp character)`  ▶ Return T if characters are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.
- `(char-upcase character)`
- `(char-downcase character)`  ▶ Return corresponding uppercase/lowercase character, respectively.
- `(digit-char i [radix])`  ▶ Character representing digit i.
- `(char-name char)`  ▶ char’s name if any, or NIL.
- `(name-char foo)`  ▶ Character named foo if any, or NIL.
- `(char-int character)`
- `(code-char code)`  ▶ Character with code.
- `(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.

\( (\text{stringp } \text{foo}) \)  \( \triangleright \) T if \( \text{foo} \) is of indicated type.

\( (\text{simple-string-p } \text{foo}) \)  \( \triangleright \) T if \( \text{foo} \) is of indicated type.

\( (\text{string=} \text{string-equal}) \)  \( \triangleright \) T if subsequences of \( \text{foo} \) and \( \text{bar} \) are equal.

\( (\text{string} \{\text{/=} \} \text{not-equal}) \)  \( \triangleright \) If \( \text{foo} \) is lexicographically not equal, greater, not less, less, or not greater, respectively, then return position of first mismatching character in \( \text{foo} \). Otherwise return NIL. Obey/ignore, respectively, case.

\( (\text{make-string} \text{size} \{\text{ButtonModule} \}) \)  \( \triangleright \) Return string of length \( \text{size} \).

\( (\text{string} \text{x}) \)  \( \triangleright \) Convert \( \text{x} \) (symbol, string, or character) into a string, a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

\( (\text{string-capitalize} \text{string-upcase} \text{string-downcase}) \)  \( \triangleright \) Convert string into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

\( (\text{string-trim} \text{string-left-trim} \text{string-right-trim}) \)  \( \triangleright \) Return string with all characters in sequence \( \text{char-bag} \) removed from both ends, from the beginning, or from the end, respectively.

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

\( (\text{parse-integer} \text{string} \text{radix} \text{junk-allowed} \text{bool}) \)  \( \triangleright \) Return integer parsed from \( \text{string} \) and index of parse end.

4 Conses

4.1 Predicates

\( (\text{consp} \text{foo}) \)  \( \triangleright \) Return T if \( \text{foo} \) is of indicated type.

\( (\text{listp} \text{foo}) \)  \( \triangleright \) Return T if \( \text{foo} \) is of indicated type.

\( (\text{endp} \text{list}) \)  \( \triangleright \) Return T if \( \text{list} \)/\( \text{foo} \) is NIL.
(atom foo) ▷ Return T if foo is not a cons.
(tailp foo list) ▷ Return T if foo is a tail of list.
(member foo list)
  ▷ Return tail of list starting with its first element matching foo. Return NIL if there is no such element.
(member-if test foo list)
  ▷ Return tail of list starting with its first element satisfying test. Return NIL if there is no such element.
(member-if-not test foo list)
  ▷ Return tail of list starting with its first element not matching foo. Return NIL if there is no such element.
(subsetp list-a list-b)
  ▷ Return T if list-a is a subset of list-b.

4.2 Lists
(cons foo bar) ▷ Return new cons (foo . bar).
(list foo) ▷ Return list of foos.
(lists foo+)
  ▷ Return list of foos with last foo becoming cdr of last cons. Return foo if only one foo given.
(make-list num [initial-element foo NIL])
  ▷ New list with num elements set to foo.
(list-length list) ▷ Length of list; NIL for circular list.
(car list) ▷ Car of list or NIL if list is NIL. setfable.
(cdr list) ▷ Cdr of list or NIL if list is NIL. setfable.
(rest list) ▷ Return tail of list after calling cdr n times.
(nthcdr n list)
  ▷ Zero-indexed nth element of list if any, or NIL otherwise. setfable.
(nth n list)
  ▷ Return nth element of list if any, or NIL otherwise. setfable.
(cadr list) ▷ With X being one to four a and ds representing cars and cdr, e.g. (cadr bar) is equivalent to (car (cdr bar)). setfable.
(last list [num])
  ▷ Return list of last num conses of list.
(butlast list [num])
  ▷ Return list excluding last num conses.
(rplaca cons object)
  ▷ Replace car, or cdr, respectively, of cons with object.
(rplacd cons object)
  ▷ Return car, or cdr, respectively, of cons with object.
(adjoin foo list)
  ▷ Return list if foo is already member of list. If not, return (cons foo list).
(pop place)
  ▷ Set place to (cdr place), return (car place).
(\push \textit{foo} \textit{place}) \Rightarrow \text{Set place to } (\textit{cons} \textit{foo} \textit{place}).

(\pushnew \textit{foo} \textit{place})
\Rightarrow \text{Set place to } (\textit{adjoin} \textit{foo} \textit{place}).

(\begin{array}{l}
\text{(append \textit{proper-list*} \textit{foo})} \\
\text{(\textit{nconc} \textit{non-circular-list*} \textit{foo})}
\end{array})
\Rightarrow \text{Return concatenated list or, with only one argument, \textit{foo}. \textit{foo} can be of any type.}

\begin{array}{l}
\text{(revappend \textit{list} \textit{foo})} \\
\text{(\textit{nreconc} \textit{list} \textit{foo})}
\end{array}
\Rightarrow \text{Return concatenated list after reversing order in list.}

\begin{array}{l}
\text{(mapcar \textit{function list})} \\
\text{(maplist \textit{function list})}
\end{array}
\Rightarrow \text{Return list of return values of \textit{function} successively invoked with corresponding arguments, either cars orcdrs, respectively, from each \textit{list}.}\textit{function} should have some side effects.

\begin{array}{l}
\text{(mapcan \textit{function} \textit{list})} \\
\text{(mapcon \textit{function} \textit{list})}
\end{array}
\Rightarrow \text{Return list of concatenated return values of \textit{function} successively invoked with corresponding arguments, either cars or cdrs, respectively, from each \textit{list}. \textit{function} should return a list.}

\begin{array}{l}
\text{(mapc \textit{function list})} \\
\text{(mapl \textit{function list})}
\end{array}
\Rightarrow \text{Return first list after successively applying \textit{function} to corresponding arguments, either cars orcdrs, respectively, from each \textit{list}. \textit{function} should have some side effects.}

\text{(copy-list \textit{list})} \Rightarrow \text{Return copy of \textit{list} with shared elements.}

\subsection*{4.3 Association Lists}
\begin{array}{l}
\text{(pairlis \textit{keys} \textit{values} \textit{alist})} \\
\text{(acons \textit{key} \textit{value} \textit{alist})}
\end{array}
\Rightarrow \text{Return \textit{alist} with a (\textit{key} , \textit{value}) pair added.}

\begin{array}{l}
\text{(assoc \textit{foo} \textit{alist})} \\
\text{(assoc-if \textit{test} \textit{foo} \textit{alist})}
\end{array}
\Rightarrow \text{First cons whose car, or cdr, respectively, satisfies \textit{test}.}

\begin{array}{l}
\text{(assoc-not \textit{foo} \textit{alist})} \\
\text{(assoc-if-not \textit{test} \textit{foo} \textit{alist})}
\end{array}
\Rightarrow \text{First cons whose car, or cdr, respectively, satisfies \textit{test}.}

\begin{array}{l}
\text{(copy-alist \textit{alist})}
\end{array}
\Rightarrow \text{Return copy of \textit{alist}.}

\subsection*{4.4 Trees}
\begin{array}{l}
\text{(tree-equal \textit{foo} \textit{bar})} \\
\text{\textit{test} function}
\end{array}
\Rightarrow \text{Return T if trees \textit{foo} and \textit{bar} have same shape and leaves satisfying \textit{test}.}

\begin{array}{l}
\text{(subst \textit{new} \textit{old} \textit{tree})} \\
\text{(nsubst \textit{new} \textit{old} \textit{tree})}
\end{array}
\Rightarrow \text{Make copy of \textit{tree} with each subtree or leaf matching \textit{old} replaced by \textit{new}.}

\begin{array}{l}
\text{(subst-if-not \textit{test} \textit{old} \textit{tree})} \\
\text{(nsubst-if-not \textit{test} \textit{old} \textit{tree})}
\end{array}
\Rightarrow \text{Make copy of \textit{tree} with each subtree or leaf satisfying \textit{test} replaced by \textit{new}.}
4.5 Sets

\[ \text{intersection} \]
\[ \text{set-difference} \]
\[ \text{union} \]
\[ \text{nintersection} \]
\[ \text{nset-difference} \]
\[ \text{nunion} \]
\[ \text{nset-exclusive-or} \]

\[ \text{test} \text{ function} \]
\[ \text{test-not} \text{ function} \]
\[ \text{key} \text{ function} \]

\[ \text{Return } a \cap b, a \setminus b, a \cup b, \text{ or } a \triangle b, \text{ respectively, of lists } a \text{ and } b. \]

5 Arrays

5.1 Predicates

\[ \text{arrayp} \ text{foo} \]
\[ \text{vectorp} \ text{foo} \]
\[ \text{simple-vector-p} \ text{foo} \]
\[ \text{bit-vector-p} \ text{foo} \]
\[ \text{simple-bit-vector-p} \ text{foo} \]

\[ \text{T} \text{ if } \text{foo} \text{ is of indicated type.} \]

\[ \text{adjustable-array-p} \ \text{array} \]
\[ \text{array-has-fill-pointer-p} \ \text{array} \]

\[ \text{T} \text{ if } \text{array} \text{ is adjustable/has a fill pointer, respectively.} \]

\[ \text{array-in-bounds-p} \ \text{array} \ [\text{subscripts}] \]

\[ \text{T} \text{ if } \text{subscripts} \text{ are in } \text{array}'s \text{ bounds.} \]

5.2 Array Functions

\[ \text{make-array} \ \text{dimension-sizes} \ [\text{adjustable} \ \text{bool}] \]
\[ \text{adjust-array} \ \text{array} \ \text{dimension-sizes} \]
\[ \text{aref} \ \text{array} \ [\text{subscripts}] \]
\[ \text{row-major-aref} \ \text{array} \ i \]
\[ \text{array-row-major-index} \ \text{array} \ [\text{subscripts}] \]
\[ \text{array-dimensions} \ \text{array} \]
\[ \text{array-dimension} \ \text{array} \ i \]
\[ \text{array-total-size} \ \text{array} \]
\[ \text{array-rank} \ \text{array} \]
\[ \text{array-displacement} \ \text{array} \]

\[ \text{Return fresh, or readjust, respectively, vector or array.} \]
\[ \text{Return array element pointed to by subscripts. setfable.} \]
\[ \text{Return } i\text{th element of array in row-major order. setfable.} \]
\[ \text{Index in row-major order of the element denoted by subscripts.} \]
\[ \text{List containing the lengths of array's dimensions.} \]
\[ \text{Length of } i\text{th dimension of array.} \]
\[ \text{Number of elements in array.} \]
\[ \text{Number of dimensions of array.} \]
\[ \text{Target array and offset.} \]
\( \text{bit} \) \text{bit-array [subscripts]} \\
\( \text{sbit} \) \text{simple-bit-array [subscripts]} \\
\( \text{bit-not} \) \text{bit-array [result-bit-array\text{T}\text{L}]} \\
\( \text{bit-eqv} \) \text{bit-and} \text{bit-andc1} \text{bit-andc2} \\
\( \text{bit-nand} \) \text{bit-ior} \text{bit-orc1} \text{bit-orc2} \\
\( \text{bit-xor} \) \text{bit-nor} \\
\begin{matrix}
\text{array-rank-limit} & \geq 8. \\
\text{array-dimension-limit} & \geq 1024. \\
\text{array-total-size-limit} & \geq 1024. 
\end{matrix}

5.3 Vector Functions

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

\( \text{vector} \) \text{foo} \\
\( \text{svref vector i} \) \\
\( \text{vector-push foo vector} \) \\
\( \text{vector-push-extend foo vector [num]} \) \\
\( \text{vector-pop vector} \) \\
\( \text{fill-pointer vector} \)

6 Sequences

6.1 Sequence Predicates

\( \{\text{every} \text{notevery}\} \) \text{test sequence\text{+}} \\
\( \{\text{some} \text{notany}\} \) \text{test sequence\text{+}}
6.2 Sequence Functions

(/make-sequence sequence-type [initial-element])
  ⊲ Make sequence of sequence-type with size elements.

(/concatenate type sequence)
  ⊲ Return concatenated sequence of type.

(/merge type sequence-a sequence-b test [key function])
  ⊲ Return interleaved sequence of type. Merged sequence will be
  sorted if both sequence-a and sequence-b are sorted.

(/fill sequence foo {start end})
  ⊲ Return sequence after setting elements between start and end to foo.

(/length sequence)
  ⊲ Return length of sequence (being value of fill pointer if
  applicable).

(/count foo sequence)
  ⊲ Return number of elements in sequence which match foo.

(/count-if test sequence)
  ⊲ Return number of elements in sequence which satisfy test.

(/elt sequence index)
  ⊲ Return element of sequence pointed to by zero-indexed
  index. settable.

(/subseq sequence start [end])
  ⊲ Return subsequence of sequence between start and end.
  settable.

(/sort sequence test [key function])
  ⊲ Return sequence sorted. Order of elements considered
  equal is not guaranteed/retained, respectively.

(/reverse sequence)
  ⊲ Return sequence in reverse order.

(/reverse sequence)
  ⊲ Return sequence in reverse order.

(/find position) foo sequence
  ⊲ Return first element in sequence which matches foo, or its
  position relative to the begin of sequence, respectively.
\[
\begin{align*}
\text{(find-if \text{test sequence})} & \quad \triangleright \text{Return first element in sequence which satisfies test, or its position relative to the begin of sequence, respectively.} \\
\text{(search sequence-a sequence-b)} & \quad \triangleright \text{Search sequence-b for a subsequence matching sequence-a. Return position in sequence-b, or \text{NIL}.} \\
\text{(remove foo sequence)} & \quad \text{\{ } \text{test function \text{start \text{end \text{key function}}} \} \text{\}} \\
\text{(delete foo sequence)} & \quad \text{\{ } \text{test function \text{start \text{end \text{key function}}} \} \text{\}} \\
\text{\{ } \text{count count} \text{\}} \\
\text{(remove-duplicates sequence)} & \quad \triangleright \text{Make copy of sequence without duplicates.} \\
\text{(delete-duplicates sequence)} & \quad \triangleright \text{Make copy of sequence with all (or count) elements satisfying test removed.} \\
\text{(substitute new old sequence)} & \quad \triangleright \text{Make copy of sequence with all (or count) olds replaced by new.} \\
\text{(replace sequence-a sequence-b)} & \quad \triangleright \text{Replace elements of sequence-a with elements of sequence-b.} \\
\text{(map type function sequence+)} & \quad \triangleright \text{Apply function successively to corresponding elements of the sequences. Return values as a sequence of type. If type is \text{NIL}, return \text{NIL}.}
\end{align*}
\]
- **map-into** `result-sequence function sequence`*
  - Store into `result-sequence` successively values of `function` applied to corresponding elements of the sequences.

- **reduce** `function sequence`*
  - Starting with the first two elements of `sequence`, apply `function` successively to its last return value together with the next element of `sequence`. Return last value of function.

- **copy-seq** `sequence`*
  - Copy of `sequence` with shared elements.

---

### 7 Hash Tables

The Loop Facility provides additional hash table-related functionality; see `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.

- **hash-table-p** `foo`*
  - Return `T` if `foo` is of type `hash-table`.

- **make-hash-table**
  - Make a hash table.

- **gethash** `key hash-table [default NIL]`*
  - Return object with `key` if any or `default` otherwise; and `T` if found, `NIL` otherwise. `setf`.

- **hash-table-count** `hash-table`*
  - Number of entries in `hash-table`.

- **remhash** `key hash-table`*
  - Remove from `hash-table` entry with `key` and return `T` if it existed. Return `NIL` otherwise.

- **clrhash** `hash-table`*
  - Empty `hash-table`.

- **maphash** `function hash-table`*
  - Iterate over `hash-table` calling `function` on key and value. Return `NIL`.

- **with-hash-table-iterator** `(foo hash-table) (declare (deq* `form*)`)*
  - Return values of forms. In forms, invocations of (foo) return: `T` if an entry is returned; its key; its value.

- **hash-table-test** `hash-table`*
  - Test function used in `hash-table`.

- **hash-table-size** `hash-table`*

- **hash-table-rehash-size** `hash-table`*

- **hash-table-rehash-threshold** `hash-table`*
  - Current size, rehash-size, or rehash-threshold, respectively, as used in `make-hash-table`.

- **sxhash** `foo`*
  - Hash code unique for any argument `equal foo`.
8 Structures

(defun defstruct (foo)
  (let ((conc-name :foo)
         (constructor (make-fun (ord-lambda))))
    (define structure foo
      (include struct (slot init :type sl-type :read-only b))
      (slot :test (number) :named (initial-offset n))
      (predicate (print-function (f-printer))))

; Define structure foo together with functions MAKE-foo,
; COPY-foo and foo-;P; and setfable 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 with-
;out 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)
; T if foo and bar are identical.

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

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

(defun equalp foo bar)
; T 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)
; T if foo is NIL; NIL otherwise.

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

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

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

9.2 Variables

\(\text{defconstant} \ foo \ \text{form} \ \text{doc} \)
\text{Assign value of form to global constant/dynamic variable } foo.\n
\(\text{defparameter} \ foo \ \text{form} \ \text{doc} \)
\text{Unless bound already, assign value of form to dynamic variable } foo.\n
\(\text{set} \ \text{symbol foo} \)
\text{Set symbol's value cell to } foo. \text{Deprecated.}\n
\(\text{makunbound} \ foo \)
\text{Delete special variable } foo \text{ if any.}\n
\(\text{get} \ \text{symbol key} \ [\text{default} \ NIL] \)
\text{First entry } key \text{ from property list stored in symbol/in place, respectively, or } default \text{ if there is no key. settable.}\n
\(\text{get-properties} \ \text{property-list keys} \)
\text{Return key and value of first entry from property-list matching a key from keys, and tail of property-list starting with that key. Return NIL, NIL, and NIL if there was no matching key in property-list.}\n
\(\text{remprop} \ \text{symbol key} \)
\(\text{remf} \ \text{place key} \)
\text{Remove first entry } key \text{ from property list stored in symbol/in place, respectively. Return } \text{T} \text{ if } key \text{ was there, or } NIL \text{ otherwise.}\n
\(\text{progv} \ \text{symbols values form} \)
\text{Evaluate forms with locally established dynamic bindings of symbols to values or NIL. Return values of forms.}\n
\(\text{let} \ \{\text{name value} \} \) \(\text{let*} \ \{\text{name value} \} \)
\text{Evaluate forms with names lexically bound (in parallel or sequentially, respectively) to values. Return values of forms.}
9.3 Functions

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

\[
\begin{align*}
\text{(\&var\^{*} \text{\&optional } \text{\&key } \text{\&aux}) } & \text{\&rest } \text{\&allow-other-keys]} \\
\text{(\text{\text{\&rest} } \text{\text{\&allow-other-keys}]} \}
\end{align*}
\]

\(\text{\&rest } \text{\&allow-other-keys]}
\]

\(\text{\&aux } \text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-other-keys]}
\]

\(\text{\&allow-o
(constantly foo)  \(\triangleright\) Function of any number of arguments returning foo.

(identity foo)  \(\triangleright\) Return foo.

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

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

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

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

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

9.4 Macros

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

\[
\begin{align*}
\text{\{\&whole \ var\} \ [E] \ (\text{\var\ (macro-\(\lambda\)\(^*\)}) \ [E]} \\
\text{\{\&optional \ var\ (macro-\(\lambda\)\(^*\)) \ init\ (supplied-p)\ (supplied-p)\}\ (E)} \\
\text{\{\&rest \ \&body\ (macro-\(\lambda\)\(^*\)) \ [E]} \\
\text{\{\&key \ \{\&init \ var\ (macro-\(\lambda\)\(^*\)) \ init\ (supplied-p)\ (supplied-p)\}\ (E)} \\
\text{\{\&allow-other-keys\} \ [\&aux \ \{\&init \ var\ (supplied-p)\ (supplied-p)\}\ (E)}
\end{align*}
\]

One toplevel \([E]\) may be replaced by \&environment \ var. 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{\{\&define \ (macro-\(\lambda\)\(^*\)) \ (\text{\\{foo \ setffoo\} \ (macro-\(\lambda\)\(^*\)) \ (\text{\text{\&declare decl\(^*\)}\) \ \text{\&doc form\(^*\)}\})} \\
\text{\{\&define-compiler-macro \ \text{\{foo \ (macro-\(\lambda\)\(^*\)) \ (\text{\&declare decl\(^*\)}\) \ \text{\&doc form\(^*\)}\})}  \\
\text{\{\&define-symbol-macro \ \text{\{foo \ form\} \ (macro-\(\lambda\)\(^*\)) \ (\text{\&declare decl\(^*\)}\) \ \text{\&doc form\(^*\)}\})} \\
\text{\{\&macrolet \ \{\&init \ \{\&doc form\(^*\) \ \text{\&define-symbol-macro \ form\} \ (macro-\(\lambda\)\(^*\)) \ (\text{\text{\&declare decl\(^*\)}\) \ \text{\&doc form\(^*\)}\})} \\
\end{align*}
\]

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

\(\triangleright\) Define macro foo which on evaluation evaluates expanded form.

\(\triangleright\) Evaluate forms with locally defined mutually invisible macros foo which are enclosed in implicit blocks of the same name.
(symbol-macrolet (foo expansion-form*) (declare decl*) form*)
  ▷ Evaluate forms with locally defined symbol macros foo.

(defsetf function
  updater [doc]
  (self-λ*) (s-var*) (declare decl*)* [doc] form*)
  where defsetf lambda list (self-λ*) has the form
  (var* [optional var [init var] (supplied-p)])
  [rest var] [key var] [init var] (supplied-p)]
  [allow-other-keys] [environment var])
  ▷ Specify how to setf a place accessed by function.
  Short form: (setf function arg) value-form is replaced by
  (updater arg* value-form); the latter must return value-form.
  Long form: on invocation of (setf function arg*) value-form, forms
  must expand into code that sets the place accessed
  where self-λ* and s-var* describe the arguments of function
  and the value(s) to be stored, respectively; and that returns
  the value(s) of s-var*.
  forms are enclosed in an implicit 'block' named function.

(define-setf-expander function (macro-λ*) (declare decl*)* [doc] form*)
  ▷ Specify how to set a place accessed by function. On in-
  vocation of (setf function arg*) value-form, form* must expand
  into code returning args-vars, args, newval-vars, set-form, and
  get-form as described with get-setf-expansion where the elements of
  macro lambda list macro-λ* are bound to corresponding args.
  forms are enclosed in an implicit 'block' named function.

(get-setf-expansion place [environment var])
  ▷ Return lists of temporary variables arg-vars and of cor-
  responding args as given with place, list newval-vars with
  temporary variables corresponding to the new values, and
  set-form and get-form specifying in terms of args-vars and
  newval-vars how to setf and how to read place.

(define-modify-macro foo ([optional
  [var [init var] (supplied-p)]]
  [rest var] [key var]
  [allow-other-keys] [environment var])
  ▷ Define macro foo able to modify a place. On invocation of
  (foo place arg*), the value of function applied to place and
  args will be stored into place and returned.

.lambdalist-keywords
  ▷ List of macro lambda list keywords. These are at least:

  &whole var  ▷ Bind var to the entire macro call form.

  &optional var*  ▷ Bind vars to corresponding arguments if any.

  &rest [&body] var  ▷ Bind var to a list of remaining arguments.

  &key var*  ▷ Bind vars to corresponding keyword arguments.

  &allow-other-keys  ▷ Suppress keyword argument checking. Callers can do
  so using &allow-other-keys T.

  &environment var  ▷ Bind var to the lexical compilation environment.

  &aux var*  ▷ Bind vars as in &let.
9.5 Control Flow

(if test then else)
  ▷ Return values of then if test returns T; return values of else otherwise.

(mcond (test then...))
  ▷ Return the values of the first then* whose test returns T; return NIL if all tests return NIL.

(mwhen (unless test foo...))
  ▷ Evaluate foos and return their values if test returns T or NIL, respectively. Return NIL otherwise.

(mc case test (key...)
    (key...)
    foo...)
  ▷ Return the values of the first foo* one of whose keys is eql test. Return values of bars if there is no matching key.

(m and form*)
  ▷ Evaluate forms from left to right. Immediately return NIL if one form’s value is NIL. Return values of last form otherwise.

(m or form*)
  ▷ Evaluate forms from left to right. Immediately return primary value of first non-NIL-evaluating form, or all values if last form is reached. Return NIL if no form returns T.

(progn form*)
  ▷ Evaluate forms sequentially. Return values of last form.

(multiple-value-prog1 form-r form*)
(mprog1 form-r form*)
  ▷ Evaluate forms in order. Return values/primary value, respectively, of form-r.

(mprog2 form-a form-r form*)
  ▷ Evaluate forms in a lexical environment. Return values/primary value, respectively, of form-r.

(declare tag decl*)
  ▷ Evaluate 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 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 tag.
(catch tag form)  
▷ Evaluate forms and return their values unless interrupted by throw.

(throw tag form)  
▷ Have the nearest dynamically enclosing catch with a tag eq tag return with the values of form.

(sleep n)  
▷ Wait n seconds; return NIL.

9.6 Iteration

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

9.7 Loop Facility

\[
(\text{loop } \text{form})
\]
▷ Simple Loop. If forms do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit block named NIL.

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

\[
\{\text{with } \{\text{var-s } \{\text{var-s}\} \{d-type] = \text{foo}\}\} \{\text{and } \{\text{var-p } \{\text{var-p}\} \{d-type] = \text{bar}\}\}\}
\]
▷ Initialize (possibly trees of) local variables var-s sequentially and var-p in parallel.

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

\[
\{\text{upfrom from } \text{downfrom}\} \text{start}
\]
▷ Start stepping with start

\[
\{\text{upto downto below above}\} \text{form}
\]
▷ Specify form as the end value for stepping.

\[
\{\text{in on}\} \text{list}
\]
▷ Bind var to successive elements/tails, respectively, of list.

\[
\{\text{step \{function \#'cdr\}}\}
\]
▷ Specify the (positive) decrement or increment or the function of one argument returning the next part of the list.
This page contains a diagram of the Common Lisp loop facility. The diagram illustrates various loop constructs, such as `for`, `while`, and `do` forms, along with their associated syntax and examples. The page also includes a detailed overview of the loop facility, demonstrating how different loop constructs can be used to iterate over and manipulate data structures in Common Lisp. The page is likely part of a Common Lisp quick reference guide, providing a concise summary of the language's looping capabilities.
foobar

bind var initially to foo and later to bar.

don vector

bind var to successive elements of vector.

{the|each} {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.

{has-value|hash-values} {of|in} hash-table [using
 {has-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|j-clause}*[else k-clause
 {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 {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 {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 aloop after num iterations; num is evalu-
ated once.

{while|until} test

continue iteration until test returns NIL or T, respec-
ively.
{always} \texttt{never} test
\hspace{10pt} \triangleright \ \text{Terminate } \texttt{mloop} \text{ returning } \texttt{NIL} \text{ and skipping any } \texttt{finally} parts as soon as \texttt{test} is \texttt{NIL} or \texttt{T}, respectively. Otherwise continue \texttt{mloop} with its default return value set to \texttt{T}.

\texttt{thereis} test
\hspace{10pt} \triangleright \ \text{Terminate } \texttt{mloop} \text{ when test is } \texttt{T} \text{ and return value of test, skipping any } \texttt{finally} parts. Otherwise continue \texttt{mloop} with its default return value set to \texttt{NIL}.

\texttt{(mloop-finish)}
\hspace{10pt} \triangleright \ \text{Terminate } \texttt{mloop} \text{ immediately executing any } \texttt{finally} clauses and returning any accumulated results.

## 10 CLOS

### 10.1 Classes

\texttt{(slot-exists-p foo bar)} \hspace{10pt} \triangleright \ \texttt{T} \text{ if } \texttt{foo} \text{ has a slot } \texttt{bar}.

\texttt{(slot-boundp instance slot)} \hspace{10pt} \triangleright \ \texttt{T} \text{ if slot in } \texttt{instance} \text{ is bound}.

\texttt{(makeclass) \textit{foo} \ (superclass\textsuperscript{*}) \textit{standard object}}
\hspace{10pt} \triangleright \ \text{Define or modify } \texttt{class foo} \text{ as a subclass of superclasses. Transform existing instances, if any, by \texttt{make-instances-obsolete}. In a new instance } \texttt{i} \text{ of } \texttt{foo}, \texttt{a slot's value defaults to } \texttt{form} \text{ unless set via } \texttt{:initarg-name}; \texttt{it is readable via } \texttt{(reader i)} \text{ or } \texttt{(accessor i)}, \texttt{and writable via } \texttt{(writer value i)} \text{ or } \texttt{setf (accessor i) value} \text{. slots with } \texttt{:allocation :class} \text{ are shared by all instances of } \texttt{class foo}.

\texttt{(find-class symbol \textit{[error]} \textit{[environment]})}
\hspace{10pt} \triangleright \ \text{Return class named } \texttt{symbol}, \texttt{setfable}.

\texttt{(make-instance} \textit{class :initarg-name\textsuperscript{*}} \textit{other-keyarg\textsuperscript{*}} \texttt{)}
\hspace{10pt} \triangleright \ \text{Make new instance of } \texttt{class}.

\texttt{(reinitialize-instance} \textit{instance :initarg-value\textsuperscript{*}} \textit{other-keyarg\textsuperscript{*}} \texttt{)}
\hspace{10pt} \triangleright \ \text{Change local slots of } \texttt{instance} \text{ according to } \texttt{initargs} \text{ by means of } \texttt{shared-initialize}.

\texttt{(slot-value} \textit{slot} \textit{foo}) \hspace{10pt} \triangleright \ \text{Return value of } \texttt{slot} \text{ in } \textit{foo}, \texttt{setfable}.

\texttt{(slot-makunbound} \textit{instance} \textit{slot}) \hspace{10pt} \triangleright \ \text{Make slot in } \textit{instance} \text{ unbound}.

\texttt{(with-slots \{\textit{slot}\textsuperscript{*}} \texttt{instance (declare decl\textsuperscript{*}) form\textsuperscript{*})}
\hspace{10pt} \triangleright \ \text{Return values of } \texttt{forms} \text{ after evaluating them in a lexical environment with slots of } \texttt{instance} \text{ visible as } \texttt{setfable} \text{ slots or } \texttt{vars/with} \texttt{accessors of } \texttt{instance} \text{ visible as } \texttt{setfable} \texttt{vars.}

\texttt{(class-name} \textit{class}) \hspace{10pt} \triangleright \ \text{Get/set name of } \texttt{class}.

\texttt{(class-of} \textit{foo}) \hspace{10pt} \triangleright \ \texttt{Class foo} \text{ is a direct instance of}.

\textbf{25}

\textit{Common Lisp Quick Reference}
10.2 Generic Functions

(defgeneric foo (setf foo) (required-var+ [&optional var (var)]')
   [&rest var] [&key \{var (var)\} [\&allow-other-keys]]
   [:argument-precedence-order required-var+]
   [:documentation string]
   [:generic-function-class gf-class standard-generic-function]
   [:method-class method-class standard-method-class]
   [:method-combination method-class][c-type generic-function-class]
   [:method defmethod-args c-type generic-function-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.
Define or modify generic function foo. \texttt{gf-class} and \texttt{lambda-list} must be compatible with a pre-existing generic function or with existing methods, respectively. Changes to method-class do not propagate to existing methods. For \texttt{c-type} see section 10.3.

\begin{itemize}
  \item \texttt{(defmethod \{foo \{setf foo\}) \textbf{primary method}}
  \item \texttt{(add-method \{remove-method\}) \texttt{generic-function method}}
  \item \texttt{(find-method \{generic-function qualified-specializers \{error\}})}
  \item \texttt{(compute-applicable-methods \{generic-function args\})}
  \item \texttt{(call-next-method \{arg\})}
  \item \texttt{(no-applicable-method \{generic-function arg\})}
  \item \texttt{(invalid-method-error \{method-combination-error\}}
  \item \texttt{(no-next-method \{generic-function method \{arg\})}
  \item \texttt{(function-keywords \{method})}
  \item \texttt{(method-qualifiers \{method\}}
\end{itemize}

\texttt{\texttt{\{argument-precedence-order \{required-var\}}}}
\texttt{\texttt{:declare \{optimize \{method-selection-optimization\}}}}
\texttt{\texttt{:documentation \{string\}}}
\texttt{\texttt{:generic-function-class \{gf-class\}}}
\texttt{\texttt{:method-class \{method-class\}}}
\texttt{\texttt{:method-combination \{c-type \{c-arg\}}}
\texttt{\texttt{:lambda-list \{lambda-list\}}}
\texttt{\texttt{:environment \{environment\}}}

\texttt{\texttt{\{before \{after \{around \{qualifier\}} \}}}
10.3 Method Combination Types

- Evaluate most specific \texttt{:around} method supplying the values of the generic function. From within this method, \texttt{call-next-method} can call less specific \texttt{:around} methods if there are any. If not, or if there are no \texttt{:around} methods at all, call all \texttt{:before} methods, most specific first, and the most specific primary method which supplies the values of the calling \texttt{:around} method if any, or of the generic function; and which can call less specific primary methods via \texttt{call-next-method}.

After its return, call all \texttt{:after} methods, least specific first.

\begin{itemize}
  \item Simple built-in \texttt{method-combination} types; have the same usage as the \texttt{c-type}s defined by the short form of \texttt{n-declare-method-combination}.
  \begin{verbatim}
(defun \texttt{define-method-combination} \texttt{c-type} \begin{verbatim}
  \begin{enumerate}
    \item \begin{verbatim}
      (\texttt{|\begin{array}{l}
        \texttt{:documentation} \texttt{string} \\
        \texttt{:identity-with-one-argument} \texttt{bool}
      \end{array}|}) \end{verbatim}
    \end{verbatim}
  \end{enumerate}
\end{verbatim}

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

\begin{verbatim}
\begin{enumerate}
  \item \texttt{Long Form.} Define new method-combination \texttt{c-type}. A call to a generic function using \texttt{c-type} will be equivalent to a call to the forms returned by \texttt{body} with \texttt{ord-\ldots} bound to \texttt{c-arg} \texttt{\ldots} (cf. \texttt{n-defgeneric}), with \texttt{symbol} bound to the generic function, with \texttt{method-combination-\ldots} bound to the arguments of the generic function, and with \texttt{\texttt{groups}} bound to \texttt{lists} of \texttt{methods}. An applicable method becomes a member of the left-most \texttt{\texttt{group}} whose \texttt{predicate} or \texttt{qualifiers} match. \texttt{Methods} can be called via \texttt{n-call-method}. \texttt{Lambda lists} \texttt{(ord-\ldots)} and \texttt{(method-combination-\ldots)} according to \texttt{ord-\ldots} on page 18, the latter enhanced by an optional \texttt{&\texttt{whole}} argument.
\end{enumerate}
\end{verbatim}
\end{verbatim}
\end{itemize}

11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 32.
(define-condition foo (parent-type)  
(slot (reader reader)*)  
(slot (writer (self writer)*)*)  
(slot (accessor accessor)*)  
(allocation (instance :class Instance))  
(slot :initarg :initarg-name *)  
(slot :uniform form)  
(slot :type type)  
(slot :documentation slot-doc)  
{
{default-initargs (name value)}*)  
{documentation condition-doc}  
{string report report-function}  
}

Return new instance of condition-type.

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

Return new instance of condition-type.

(ignore-errors form*)  

Return values of forms or, in case of errors, NIL and the condition.

:invoke-debugger condition  

Invoke debugger with condition.

(assert test [(place*) (condition-type :initarg-name value*)]  

If test, which may depend on places, returns NIL, signal as correctable error condition or a new instance of condition-type or, with :format control and args (see page 38), simple-error. In the debugger, use :format arguments continue-control and continue-args to tag the continue option. Return NIL.

(handler-case foo (type ([var]) (declare decl*) condition-form*)  

If, on evaluation of foo, a condition of type is signalled, evaluate matching condition-forms with var bound to the condition, and return their values. Without a condition, bind ord-λs to values of foo and return values of forms or, without a :no-error clause, return values of foo. See page 18 for (ord-λ*).

(handler-bind ((condition-type handler-function)*) form*)  

Return values of forms after evaluating them with condition-types dynamically bound to their respective handler-functions of argument condition.
(with-simple-restart* {restart} control arg*) form*  
▷ Return values of forms unless restart is called during their evaluation. In this case, describe restart using }format control and arg (see page 38) and return NIL and T.  

(restart-case* form (restart (ord-λ*))  
{interactive arg-function}  
{report-function report-function}  
{test-function 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 #"test-function condition") returns T. If presented in the debugger, restarts are described by string or by #"report-function" (of a stream). A restart can be called by (invoke-restart restart arg*), where args match ord-λ*; or by (invoke-restart-interactively restart) where a list of the respective args is supplied by #"arg-function". See page 18 for ord-λ*.  

(restart-bind* (restart) nil*  
{interactive-function 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 value}  
{use-value value}  
▷ 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.
Common Lisp Quick Reference

(cell-error-name condition)  ▶ Name of cell which caused condition.

(unbound-slot-instance condition)  ▶ Instance with unbound slot which caused condition.

(print-not-readable-object condition)  ▶ The object not readabley printable under condition.

(package-error-package condition)  ▶ Package, path, or stream, respectively, which caused the condition of indicated type.

(type-error-datum condition)  ▶ Object which caused condition of type type-error, or its expected type, respectively.

(simple-condition-format-control condition)  ▶ Return format control or list of format arguments, respectively, of condition.

Types and Classes

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

(typep foo type [environment])  ▶ T if foo is of type.

(subtypep type-a type-b [environment])  ▶ Return T if type-a is a recognizable subtype of type-b, and NIL if the relationship could not be determined.

(type form)  ▶ Declare values of form to be of type.

(coerce object type)  ▶ Coerce object into type.

(check-type place type [string {a an} type])  ▶ Signal correctable type-error if place is not of type. Return NIL.

(stream-element-type stream)  ▶ Type of stream objects.

(array-element-type array)  ▶ Element type array can hold.

(upgraded-array-element-type type [environment])  ▶ 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 (macro-λ* (doc) form*)
  "Define type foo which when referenced as (foo any) (or as foo if macro-λ doesn’t contain any required parameters) applies expanded forms to args returning the new type. For (macro-λ*) see page 19 but with default value of * instead of NIL. forms are enclosed in an implicit \[block named foo."

(satisfies predicate)
  "Type specifier for all objects satisfying predicate."

(mod n)
  "Complement of type."

(and type* nil)
  "Type specifier for intersection of types."

(or type* nil)
  "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)

(pathnamep foo)  "T if foo is of indicated type."

(readtablep 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. (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, format, for control and args."

(with-standard-io-syntax forms*)
  "Evaluate forms with standard behaviour of reader and printer. Return values of forms."

(read string [eof-error nil] [eof-value nil] [start start] [end end] [preserve-whitespace bool])
  "Return object read from string and zero-indexed position of next character."

33
(read-delimited-list char [stream \[standard-input\] [recursive \[error\]]])  
▷ Continue reading until encountering char. Return list of objects read. Signal error if no char is found in stream.

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

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

(\peek-char \[mode \[error\]] [stream \[standard-input\] [eof-error \[error\]] [eof-val \[error\]] [recursive \[error\]]])  
▷ 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-char\ed character back into stream; return NIL.

(read-byte \[stream \[standard-input\] [eof-err \[error\]] [eof-val \[error\]]])  
▷ Read next byte from binary stream.

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

(read-sequence sequence \[stream \[standard-input\] \[\\] \[start \begin{array}{c} start \end{array} \[end \begin{array}{c} end \end{array}\] \[\\] \[\\]])  
▷ Replace elements of sequence between start and end with elements from binary or character stream. Return index of sequence’s first unmodified element.

(readtable-case readable \[upcase\] \[downcase\] \[preserve\] \[invert\])  
▷ Case sensitivity attribute (one of downcase, :inverse) of readable. setf

(set-syntax-from-char to-char from-char \[\\] \[rt \[\*\] \[readtable\] \[\*\])  
▷ Copy syntax of from-char to to-readtable. Return T.

(set-macro-character char function \[non-term-p \[error\]] \[\\] \[\\])  
▷ Make char a macro character associated with function of stream and char. Return T.

(get-macro-character char \[\\] \[\\])  
▷ Reader macro function associated with char, and T if char is a non-terminating macro character.

(make-dispatch-macro-character char \[non-term-p \[error\]] \[\\] \[\\])  
▷ Make char a dispatching macro character. Return T.
Common Lisp Quick Reference

(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.

(foo* [.. bar])  > List of foos with the terminating cdr bar.

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

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

n/d                      > The ratio n/d.

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

#C(a b)                  > (complex a b), the complex number a + bi.

#'(foo)                   > (function foo); the function named foo.

#(n)A 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.
#+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 v∗features, or \{(and | or) feature\}, or (not feature).

v∗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 readable, readable between a newline and a space, readable 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 readable or human-readable, respectively.

\(print\)-object object stream

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

\(print\)-unreadable-object (foo stream \{\type bool \identity bool\} form\)

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

\(terpri\)

⊲ Output a newline to stream. Return NIL.

\(fresh-line\)

⊲ 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.

\(write-string\)
\(write-line\)

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

\(write-byte\) byte stream

⊲ Write byte to binary stream.

\(write-sequence\) sequence stream \{\start \end\}

⊲ Write elements of sequence to binary or character stream.
Print a conditional newline if stream is a pretty printing stream. Return \texttt{NIL}.

\texttt{*print-newline*} \hspace{1em} \texttt{[stream \texttt{standard-output}]}

\textgreater{} If \texttt{T}, print arrays \texttt{readably}.

\texttt{*print-base*} \hspace{1em} Radix for printing rationals, from 2 to 36.
v ∗ print-case v
▶ Print symbol names all uppercase (upcase), all lowercase (downcase), capitalized (capitalize).

v ∗ print-circle v
▶ If T, avoid indefinite recursion while printing circular structure.

v ∗ print-escape v
▶ If NIL, avoid undefined recursion while printing circular structure.

v ∗ print-gensym v
▶ If T, print #: before uninterned symbols.

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

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

v ∗ print-pretty v
▶ If T, print prettily.

v ∗ print-radix v
▶ If T, print rationals with a radix indicator.

v ∗ print-readably v
▶ If T, print readably or signal error print-not-readable.

v ∗ print-right-margin v
▶ Right margin width in ems while pretty-printing.

\( \text{v ∗ set-print-dispatch \{ type, function, priority \}} \)
▶ 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.

\( \text{v ∗ print-dispatch \{ foo, table \}} \)
▶ Return highest priority function associated with type of foo and T if there was a matching type specifier in table.

\( \text{v ∗ copy-print-dispatch \{ table \}} \)
▶ Return copy of table or, if table is NIL, initial value of v ∗ print-pprint-dispatch ∗.

v ∗ print-pprint-dispatch ∗
▶ Current pretty print dispatch table.

13.5 Format

\( \text{v ∗ formatter \{ control \}} \)
▶ Return function of stream and arg ∗ applying \( \text{format} \) to stream, control, and arg ∗ returning NIL or any excess args.

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

- \[ \text{min-col} \] \[ \text{col-inc} \] \[ \text{min-pad} \] \[ \text{pad-char} \]
▶ 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.

- \[ \text{radix} \] \[ \text{width} \] \[ \text{pad-char} \] \[ \text{comma-char} \] \[ \text{comma-interval} \]
▶ 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|-0|R}  
> Roman. Take argument as number and print it as English cardinal number, as English ordinal number, as Roman numeral, or as old Roman numeral, respectively.

- [width] [:width|pad-char|[[:comma-interval]]]  
> Decimal/Binary/Octal/Hexadecimal. Print integer argument as number. With i, group digits comma-interval each; with 0, always prepend a sign.

- [width] [:[dec-digits]|[:shift|[:overflow-char]]  
> Fixed-Format Floating-Point. With 0, always prepend a sign.

> 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 0, always prepend a sign.

- [dec-digits]|[:int-digits]|[:width]|[:pad-char]|[@]|(@)|(||)|\]  
> Monetary Floating-Point. Print argument as fixed-format floating-point number. With ;, put sign before any padding; with 0, always prepend a sign.

{-#|#{-C}|#{-0C}|#{-0C}}  
> Character. Print, spell out, print in #\# syntax, or tell how to type, respectively, argument as (possibly non-printing) character.

- (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 y, otherwise print yes; do the same for the previous argument, respectively.

- n |  
> Newline. Print n newlines.

- n &  
> Fresh-Line. Print n – 1 newlines if output stream is at the beginning of a line, or n newlines otherwise.

- (#-|@|-@)}  
> Conditional Newline. Print a newline like pprint-newline with argument :linear, :fill, :miser, or :mandatory, respectively.

-#{-|@|->}  
> Ignored Newline. Ignore newline, or whitespace following newline, or both, respectively.

- n |  
> Page. Print n page separators.

- n -  
> Tilde. Print n tildes.

> Justification. Justify text produced by texts in a field of at least min-co columns. With t, right justify; with 0, left justify. If this would leave less than spare characters on the current line, output nl-text first.

- [v]|[: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 0, on the remaining arguments, which are extracted by pprint-pop. With ;, prefix and suffix default to ( and ). When closed by -@, spaces in body are replaced with conditional newlines.

---

39
- `[:direction] {input | output | io | probe}`
  \> \textbf{Indent.} Set indentation to \textit{n} relative to leftmost/to current position.

- `[:table]`  \> \textbf{Tabulate.} Move cursor forward to column number \textit{c + ki}, \textit{k} \geq 0 being as small as possible. With \textit{;} calculate column numbers relative to the immediately enclosing section. With \textit{\Theta}, move to column number \textit{c0 + c + ki} where \textit{c0} is the current position.

- `[:limit] \{ text \} \{ limit \}`  \> \textbf{Iteration.} Use \textit{text} repeatedly, up to \textit{limit}, as control string for the elements of the list argument or (with \textit{\Theta}) for the remaining arguments. With \textit{;} or \textit{\Theta}, list elements or remaining arguments should be lists of which a new one is used at each iteration step.

- `[:z | y | x] [z | x | y]`  \> \textbf{Escape Upward.} Leave immediately \textless \textless, \textless - >, \textless - : >, \textless - \textgreater, or the entire \textit{if\textunderscore format} operation. With one to three prefixes, act only if \textit{x} = 0, \textit{x} = \textit{y}, or \textit{y} \leq \textit{z}, respectively.

- `[:i | \Theta] \{ text \} [text] [:i | default]`  \> \textbf{Conditional Expression.} Use the zero-indexed argument (or \textit{th} if given) \textit{text} as a \textit{format} control subclause. With \textit{;} use the first \textit{text} if the argument value is \texttt{NIL}, or the second \textit{text} if it is \texttt{T}. With \textit{\Theta}, do nothing for an argument value of \texttt{NIL}. Use the only \textit{text} and leave the argument to be read again if it is \texttt{T}.

- `[-\Theta? ]`  \> \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.

- `[:prefix \{prefix\} \{i | \Theta\}] [package \{\} external-format \{function\}]`  \> \textbf{Call Function.} Call all-uppercase \texttt{package:function} with the arguments stream, format-argument, colon-p, at-sign-p and \texttt{prefixes} for printing format-argument.

- `[:i | \Theta]`  \> \textbf{Write.} Print argument of any type obeying every printer control variable. With \textit{;} pretty-print. With \textit{\Theta}, print without limits on length or depth.

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

\section*{13.6 Streams}

\begin{itemize}
  \item \texttt{\{open path\}}
  \item \texttt{:\{input | output | io | probe\} \{input\}}
  \item \texttt{:\{direction\} \{type\} \{new-version\} \{error\} \{rename\} \{overwrite\} \{append\} \{supersede\} \{if\textunderscore exists\} \{if\textunderscore does\textunderscore not\textunderscore exist\} \{create\} \{error\} \{NIL\}}
  \item \texttt{:\{external\textunderscore format\} \{NIL\} \{for\textunderscore direction\} \{probe\} \{create\} \{error\} \{NIL\} \{otherwise\}}
\end{itemize}
(make-concatenated-stream input-stream*)
  ▷ Return stream of indicated type.
(make-broadcast-stream output-stream*)
(make-two-way-stream input-stream-part output-stream-part)
(make-echo-stream from-input-stream to-output-stream)
(make-synonym-stream variable-bound-to-stream)
  ▷ Return stream of indicated type.

(make-string-input-stream string [start end])
  ▷ Return a string-stream supplying the characters from string.
(make-string-output-stream [element-type type character])
  ▷ Return a string-stream accepting characters (available via get-output-stream-string).
(concatenated-stream-streams concatenated-stream)
  ▷ Return list of streams concatenated-stream still has to read from/broadcast-stream is broadcasting to.
(two-way-stream-input-stream two-way-stream)
(two-way-stream-output-stream two-way-stream)
(echo-stream-input-stream echo-stream)
(echo-stream-output-stream echo-stream)
  ▷ Return source stream or sink stream of two-way-stream/echo-stream, respectively.
(synonym-stream-symbol synonym-stream)
  ▷ Return symbol of synonym-stream.

(get-output-stream-string string-stream)
  ▷ Clear and return as a string characters on string-stream.

(file-position stream [start end position])
  ▷ Return position within stream, or set it to position and return T on success.
(file-string-length stream foo)
  ▷ Length foo would have in stream.
(listen [stream standard-input])
  ▷ T if there is a character in input stream.

(clear-input [stream standard-input])
  ▷ Clear input from stream, return NIL.
(force-output) [stream standard-output])
  ▷ End output to stream and return NIL immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.

(close stream [abort bool])
  ▷ Close stream. Return T if stream had been open. If :abort is T, delete associated file.
(with-open-file (stream path open-arg) (declare decl)* form)
  ▷ Use 'open with open-args to temporarily create stream to path; return values of forms.
(with-open-stream (foo stream) (declare decl)* form)
  ▷ Evaluate forms with foo locally bound to stream. Return values of forms.
(with-input-from-string (foo string :index index :start start :end end) (declare decl)* form)
  ▷ 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-directory
 pathname-name
 pathname-type)

(pathname-version path-or-stream)

(parse-namestring foo [host]
 [default-pathname-defaults]
 [start |end| junk-allowed])

(merge-pathnames path-or-stream
 [default-pathname-defaults]
 [default-version])

(default-pathname-defaults)

(user-homedir-pathname [host])
(enough-namestring path-or-stream
  [root-path default-pathname-default])
  ▷ 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*}[.LISP]{version*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)
(keywordp foo)
  ▷ T if foo is of indicated type.
14.2 Packages

:*bar*

Keyword, evaluates to :bar.

`package:symbol` ➞ Exported symbol of package.

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

```
(defun package foo
  ;; (nicknames nick*)
  ;; (documentation string)
  ;; (intern interned-symbol*)
  ;; (use used-package*)
  ;; (import-from pkg imported-symbol*)
  ;; (shadowing-import-from pkg shd-symbol*)
  ;; (shadow shd-symbol*)
  ;; (export exported-symbol*)
  ;; (size int))

  ;; Create or modify package foo with interned-symbols, symbols
  ;; from used-packages, imported-symbols, and shd-symbols. Add
  ;; shd-symbols to foo’s shadowing list.

  (make-package foo
    ;; nicknames (nick* list))
    ;; use (used-package*)

  ;; Create package foo.

  (rename-package package new-name (new-nicknames list))
    ;; Rename package. Return renamed package.

  ;; Make package foo current.

  (use-package other-packages (package :package))
    ;; Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T.

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

  (package-used-by-list package)

  (delete-package package)
    ;; Delete package. Return T if successful.

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

  (unintern symbol (package :package))
    ;; Remove symbol from package, return T on success.

  (import shadowing-import symbols (package :package))
    ;; 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 :package))
    ;; Make symbols of package shadow any otherwise accessible, equally named symbols from other packages. Return T.
```

Common Lisp Quick Reference
实现了包名互相遮蔽的符号

- **package-shadowing-symbols [package]**
  - List of symbols of package that shadow any otherwise accessible, equally named symbols from other packages.

实现了包的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了包的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.

实现了符号的取消导出

- **unexport symbols [package]**
  - Revert symbols to internal status. Return T.

实现了符号的迭代

- **do-symbols**
  - Iterate over symbols that shadow any otherwise accessible, equally named symbols from other packages.

实现了符号的导出

- **export symbols [package]**
  - Make symbols external to package. Return T.
Common Lisp Quick Reference

Truth; the supertype of every type including t; the superclass of every class except t; *terminal-io*.

Falsity; the empty list; the empty type, subtype of every type; *standard-input*; *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

(special-operator-p foo) ▷ T if foo is a special operator.

(compiled-function-p foo) ▷ T if foo is of type compiled-function.

15.2 Compilation

(compile \{NIL definition
{name \{(setf name\) \{definition\})\}
▷ Return compiled function or replace name's function definition with the compiled function. Return T in case of warnings or errors, and T in case of warnings or errors excluding style-warnings.

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

(compile-file-pathname \{file \{other-keyargs\}\}
▷ Pathname \{compile-file\} writes to if invoked with the same arguments.

(load \{:verbose bool
:print bool
:if-does-not-exist bool
:external-format file-format \{default\}\}
▷ Load source file or compiled file into Lisp environment. Return T if successful.

\{compile-file\} \{pathname=NIL
|\{true-name=NIL\}\}
▷ Input file used by \{compile-file\}/by \{load\}.

\{print\} \{\{verbose\}
\{\load\}
▷ Defaults used by \{compile-file\}/by \{load\}.
```lisp
(eval-when ([compile-toplevel] [load-toplevel] [execute]) form)
▷ 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.)

(locally (declare decl*) form)
▷ Evaluate forms in a lexical environment with declarations decl in effect. Return values of forms.

(with-compilation-unit ([override bood] form))
▷ 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 [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 {[slot-names slots [all local slots]] [environment 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])

(compiler-macro-function {name [setf name]} [environment])
▷ Return specified macro function, or compiler macro function, respectively, if any. Return `nil` otherwise. setfable.

(eval 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 nil])
▷ Print interned symbols containing string.

(apropos-list string [package nil])
▷ 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 nil])
▷ Invoke editor if possible.

(macroexpand-1 form [environment])

(macroexpand) form [environment]
▷ Return macro expansion, once or entirely, respectively, of form and `true` if form was a macro form. Return form and `nil` otherwise.
```
• macroexpand-hook
  > Function of arguments expansion function, macro form, and environment called by \texttt{macroexpand-1} to generate macro expansions.

\texttt{\textbf{trace} \{\texttt{function} \{\texttt{setf function}\}\}}
  > Cause functions to be traced. With no arguments, return list of traced functions.

\texttt{\textbf{untrace} \{\texttt{function} \{\texttt{setf function}\}\}}
  > Stop functions, or each currently traced function, from being traced.

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

\texttt{\textbf{step} \texttt{form}}
  > Step through evaluation of \texttt{form}. Return values of \texttt{form}.

\texttt{\textbf{break} [\texttt{control arg\*}]}
  > Jump directly into debugger; return NIL. See page 38, \texttt{format}, for control and \texttt{arg}s.

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

\texttt{\textbf{inspect} foo}
  > Interactively give information about \texttt{foo}.

\texttt{\textbf{describe} foo [\texttt{stream}]}
  > Send information about \texttt{foo} to \texttt{stream}.

\texttt{\textbf{describe-object} foo [\texttt{stream}]}  
  > Send information about \texttt{foo} to \texttt{stream}. Called by \texttt{describe}.

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

\texttt{\textbf{room} [\texttt{NIL}]\{\texttt{default NIL}\}}
  > Print information about internal storage management to \texttt{standard-output*}.

15.4 Declarations

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

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

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

\texttt{(\textbf{ftype} type function\*)}
  > Suppress warnings about used/unused bindings.

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

▷ Current time, or computing time, respectively, in clock ticks.

/internal-time-units-per-second

▷ Number of clock ticks per second.

(/get-universal-time)

▷ Seconds from 1900-01-01, 00:00, ignoring leap seconds.

(/encode-universal-time sec min hour date month year [zone curr])

▷ Return second, minute, hour, date, month, year, day, daylight-p, and zone.

(/decode-universal-time universal-time [time-zone current])

▷ Computer name.

(/lisp-implementation)

▷ Name or version of implementation, operating system, or hardware, respectively.

(/long-site-name)

▷ String representing physical location of computer.

(/machine-instance)

▷ Computer name.