\texttt{\{f\texttt{atan}\ a \ [b]\}} \quad \triangleright \quad \arctan \frac{a}{b} \quad \text{in radians.}

\texttt{\{f\texttt{sinh}\ a\}} \quad \triangleright \quad \sinh a, \cosh a, \text{ or } \tanh a, \text{ respectively.}

\texttt{\{f\texttt{cosh}\ a\}} \quad \triangleright \quad \sinh a, \cosh a, \text{ or } \tanh a, \text{ respectively.}

\texttt{\{f\texttt{asinh}\ a\}} \quad \triangleright \quad \arcsin a, \cosh a, \text{ or } \tanh a, \text{ respectively.}

\texttt{\{f\texttt{acosh}\ a\}} \quad \triangleright \quad \sinh a, \cosh a, \text{ or } \tanh a, \text{ respectively.}

\texttt{\{f\texttt{atanh}\ a\}} \quad \triangleright \quad \arctan a, \text{ or } \tanh a, \text{ respectively.}

\texttt{\{f\texttt{cis}\ a\}} \quad \triangleright \quad e^{i a} = \cos a + i \sin a.

\texttt{\{f\texttt{conjugate}\ a\}} \quad \triangleright \quad \text{Return complex conjugate of } a.

\texttt{\{f\texttt{max}\ num\}} \quad \triangleright \quad \text{Greatest, or least, respectively, of } num_s.

\texttt{\{f\texttt{min}\ num\}} \quad \triangleright \quad \text{Greatest, or least, respectively, of } num_s.

\texttt{\{f\texttt{round}\ n\ [d]\}} \quad \triangleright \quad \text{Return as integer or float, respectively, } n/d \text{ rounded, or rounded towards } -\infty, +\infty, \text{ or } 0, \text{ respectively; and remainder.}

\texttt{\{f\texttt{floor}\ n\ [d]\}} \quad \triangleright \quad \text{Same as } f\texttt{round} \text{ or } f\texttt{truncate}, \text{ respectively, but return remainder only.}

\texttt{\{f\texttt{mod}\ n\ [d]\}} \quad \triangleright \quad \text{Same as } f\texttt{floor} \text{ or } f\texttt{truncate}, \text{ respectively, but return remainder only.}

\texttt{\{f\texttt{rem}\ n\ [d]\}} \quad \triangleright \quad \text{Same as } f\texttt{floor} \text{ or } f\texttt{truncate}, \text{ respectively, but return remainder only.}

\texttt{\{f\texttt{random}\ limit\ [state\ \texttt{\rightarrow random-state}]\}} \quad \triangleright \quad \text{Return non-negative random number less than } limit, \text{ and of the same type.}

\texttt{\{f\texttt{make-random-state}\ [\{state\ [T]\}\ \texttt{\ightarrow random-state}]\}} \quad \triangleright \quad \text{Copy of } \texttt{random-state} \text{ object state or of the current random state, or a randomly initialized fresh random state.}

\texttt{*random-state*} \quad \triangleright \quad \text{Current random state.}

\texttt{\{f\texttt{float-sign}\ num-a [num-b]\}} \quad \triangleright \quad \text{num-b with num-a’s sign.}

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

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

\texttt{\{f\texttt{denominator}\ rational\}} \quad \triangleright \quad \text{Numerator or denominator, respectively, of rational’s canonical form.}

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

\texttt{\{f\texttt{imagpart}\ number\}} \quad \triangleright \quad \text{Real part or imaginary part, respectively, of number.}

\texttt{\{f\texttt{complex}\ real\ [imag\]\}} \quad \triangleright \quad \text{Make a complex number.}

\texttt{\{f\texttt{phase}\ num\}} \quad \triangleright \quad \text{Angle of num’s polar representation.}

\texttt{\{f\texttt{abs}\ n\}} \quad \triangleright \quad \text{Return } |n|.

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

\texttt{\{f\texttt{rationalize}\ real\}} \quad \triangleright \quad \text{Convert real to rational. Assume complete/limited accuracy for real.}

\texttt{\{f\texttt{float}\ real\ [prototype]\}} \quad \triangleright \quad \text{Convert real into float with type of prototype.}
1 Numbers

1.1 Predicates

\( (= \text{number}^+) \)
\( (= \text{number}^+) \)
\( (\neq \text{number}^+) \)
\( (> \text{number}^+) \)
\( (\geq \text{number}^+) \)
\( (\leq \text{number}^+) \)

\( (\leq \text{number}^+) \)
\( \Rightarrow \) Return \( \top \) if \( \text{numbers} \) are monotonically non-decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

\( (\minusp a) \)
\( (\zerop a) \)
\( (\plusp a) \)
\( (\evenp \text{int}) \)
\( (\oddp \text{int}) \)
\( (\text{numberp foo}) \)
\( (\text{realp foo}) \)
\( (\text{rationalp foo}) \)
\( (\text{floatp foo}) \)
\( (\text{integerp foo}) \)
\( (\text{complexp foo}) \)
\( (\text{random-state-p foo}) \)

1.2 Numeric Functions

\( (+ \ a \ b) \)
\( (- a \ b) \)
\( (/ a \ b) \)
\( (\times a \ b) \)
\( (\div a \ b) \)
\( (\sqrt a) \)
\( (\sqrt[n] a) \)
\( (\text{lcm} \text{integer}^+) \)
\( (\text{gcd} \text{integer}^+) \)
\( (\pi) \)
\( (\approx \text{number}^+) \)
\( (\sin a) \)
\( (\cos a) \)
\( (\tan a) \)
\( (\text{asin a}) \)
\( (\text{acos a}) \)

Typographic Conventions

(name, \(\_\text{name}, \_\text{name}, \_\text{name}, \_\text{name}, \_\text{name}\)) \n\(\Rightarrow\) Symbol defined in Common Lisp; esp. function, generic function, macro, special operator, variable, constant.

them \(\Rightarrow\) Placeholder for actual code.

me \(\Rightarrow\) Literal text.

\(\text{foo}^{\text{bar}}\) \(\Rightarrow\) Either one \(\text{foo}\) or nothing; defaults to \(\text{bar}\).

\(\text{foo}^{+}, \{\text{foo}\}^{+}\) \(\Rightarrow\) Zero or more \(\text{foo}\).

\(\text{foo}^{-}, \{\text{foo}\}^{-}\) \(\Rightarrow\) One or more \(\text{foo}\).

\(\text{foo}\) \(\Rightarrow\) English plural denotes a list argument.

\(\{\text{foo}\text{bar}\text{baz}\}\) \(\Rightarrow\) Either \(\text{foo}, \text{bar}, \text{or baz}\).

\(\{\text{foo}\}^{\text{bar}\text{baz}}\) \(\Rightarrow\) Anything from none to each of \(\text{foo}, \text{bar}, \text{and baz}\).

\(\tilde{\text{foo}}\) \(\Rightarrow\) Argument \(\text{foo}\) is not evaluated.

\(\overline{\text{bar}}\) \(\Rightarrow\) Argument \(\text{bar}\) is possibly modified.

\(\overline{\text{foo}}^{\text{a}}\) \(\Rightarrow\) \(\text{foo}^{\text{a}}\) is evaluated as in \(\text{progn}\); see page 21.

\(\text{foo}^{\text{a}}\) \(\Rightarrow\) Primary, secondary, and ath return value.

T; \(\text{NIL}\) \(\Rightarrow t, \text{or truth in general}; \text{and \text{NIL} or \{\}}.\)
3 Strings

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

(make-string size \{initial-element char \element-type type {character}\})

\( \Rightarrow \) string of length size.

(string x)

\{string-capitalize \{string-upcase \{string-downcase \string\} \} \x \{\start \end\}\}

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

(string-trim \{string-left-trim \{string-right-trim \string\} \} char-bag string)

\( \Rightarrow \) Return string with all characters in sequence char-bag removed from both ends, from the beginning, or from the end, respectively.

(char string i)

(schar string i)

\( \Rightarrow \) Return zero-indexed \(i\)th character of string ignoring/obeying, respectively, fill pointer. setfable.

(parse-integer string \{\start \end \radix int \junk-allowed bool\})

\( \Rightarrow \) Return integer parsed from string and index of parse end.

4 Conses

4.1 Predicates

(consp foo)

\( \Rightarrow \) Return \(T\) if foo is of indicated type.

(listp foo)

\( \Rightarrow \) Return \(T\) if list/foo is NIL.

(endp list)

\( \Rightarrow \) Return \(T\) if list/foo is NIL.

(null foo)

\( \Rightarrow \) Return \(T\) if foo is of indicated type.

(1.3 Logic Functions)

Negative integers are used in two’s complement representation.

(boole operation int-a int-b)

\( \Rightarrow \) Return value of bitwise logical operation. operations are

\(\{\text{boole-1} \Rightarrow \text{int-a}.\}

\(\{\text{boole-2} \Rightarrow \text{int-b}.\}

\(\{\text{boole-c1} \Rightarrow \text{¬int-a}.\}

\(\{\text{boole-c2} \Rightarrow \text{¬int-b}.\}

\(\{\text{boole-set} \Rightarrow \text{All bits set.}\}

\(\{\text{boole-clr} \Rightarrow \text{All bits zero.}\}

\(\{\text{boole-eqv} \Rightarrow \text{int-a} \equiv \text{int-b}.\}

\(\{\text{boole-and} \Rightarrow \text{int-a} \land \text{int-b.}\}

\(\{\text{boole-andc1} \Rightarrow \text{¬int-a} \land \text{int-b.}\}

\(\{\text{boole-andc2} \Rightarrow \text{int-a} \land \text{¬int-b.}\}

\(\{\text{boole-nand} \Rightarrow \text{¬(int-a} \land \text{int-b).}\}

\(\{\text{boole-ior} \Rightarrow \text{int-a} \lor \text{int-b}.\}

\(\{\text{boole-orc1} \Rightarrow \text{¬int-a} \lor \text{int-b.}\}

\(\{\text{boole-orc2} \Rightarrow \text{int-a} \lor \text{¬int-b.}\}

\(\{\text{boole-xor} \Rightarrow \text{¬(int-a} \equiv \text{int-b).}\}

\(\{\text{boole-nor} \Rightarrow \text{¬(int-a} \lor \text{int-b).}\}

(lognot integer)

\( \Rightarrow \text{¬integer}.\)

(logeqv integer+)

\( \Rightarrow \text{logand integer+}.\)

\(\{\text{logandc1} \Rightarrow \text{int-a} \land \text{int-b.}\}

\(\{\text{logandc2} \Rightarrow \text{¬int-a} \land \text{int-b.}\}

\(\{\text{logand} \Rightarrow \text{int-a} \land \text{int-b.}\}

\(\{\text{logor} \Rightarrow \text{int-a} \lor \text{int-b.}\}

\(\{\text{logxor} \Rightarrow \text{¬(int-a} \lor \text{int-b).}\}

\(\{\text{logior} \Rightarrow \text{¬int-a} \lor \text{int-b.}\}

\(\{\text{lognot} \Rightarrow \text{¬integer}.\}

\(\{\text{logbitp} \Rightarrow \text{¬int-a} \lor \text{int-b.}\)

\(\{\text{logcount} \Rightarrow \text{Number of 1 bits in int} \geq 0, \text{number of 0 bits in int} < 0.\}

\(\{\text{logtest} \Rightarrow \text{¬int-a} \lor \text{int-b.}\}

\(\{\text{lognot} \Rightarrow \text{¬integer}.\)
1.4 Integer Functions

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

(integer.test byte-spec integer)  n
> Return t if any bit specified by byte-spec in integer is set.

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

(byte-spec integer)  n
> Extract byte denoted by byte-spec from integer. setfable.

{deposit-field} int-a byte-spec int-b)  n
> Return int-b with bits denoted by byte-spec replaced by cor-
responding bits of int-a, or by the low (byte-size byte-spec)
bits of int-a, respectively.

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

(byte size position)  n
> Byte specifier for a byte of size bits starting at a weight of
2^position.

{byte-size byte-spec)  n
> Size or position, respectively, of byte-spec.

1.5 Implementation-Dependent

{short-float)  n
> Smallest possible number making a difference when added or
subtracted, respectively.

{single-float)  n
> Epsilon.

{double-float)  n
> Negative-epsilon.

{long-float)  n
> Least-negative.

{least-negative-normalized)  n
> Least-positive.

{least-positive-normalized)  n
> Least-positive-normalized.

{most-negative)  n
> Most-negative.

{most-positive)  n
> Most-positive.

{short-float single-float double-float long-float)  n
> Fixednum.

{least-negative least-negative-normalized least-positive least-positive-normalized)  n
> Available numbers closest to -0 or +0, respectively.

2 Characters

The standard-char type comprises a-z, A-Z, 0-9, Newline, Space, and !?"'":;\;="/\-<>@\#%&()[]\{}.

{character foo)  n
> T if argument is of indicated type.

{standard-char-p char)  n
> T if character is visible, alphabetic, or alphanumeric, respectiv-
ely.

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

{digit-char-p character [radix])  n
> Return its weight if character is a digit, or NIL otherwise.

{char= character+)  n
> Return T if all characters, or none, respectively, are equal.

{char-< character+)  n
> Return T if all characters, or none, respectively, are equal
ignoring case.

(char| character+)  n
> Return T if all characters, or none, respectively, are equal
ignoring case.

{char| character+)  n
> Return T if characters are monotonically decreasing, mono-
tonically non-decreasing, respectively.

{char-greaterp character+)  n
> Return T if characters are monotonically decreasing, mono-
tonically non-increasing, monotonically increasing, or monoton-
ically non-decreasing, respectively.

{char-lessp character+)  n
> Return T if characters are monotonically decreasing, mono-
tonically non-increasing, monotonically increasing, or monoton-
ically non-decreasing, respectively.

(char-upcase character)  n
> Return corresponding uppercase/lowercase character, respectiv-
ely.

(char-downcase character)  n
> Return corresponding uppercase/lowercase character, respectiv-
ely.

{digit-char i [radix])  n
> Character representing digit i.

{name-char char)  n
> char’s name if any, or NIL.

{name-char foo)  n
> Character named foo if any, or NIL.

{char-int character)  n
> Code of character.

{char-code character)  n
> Character with code.

{char-code-limit)  n
> Upper bound of {character code char}; ≥ 96.

{character c)  n
> Return #\c.
5.3 Vector Functions

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

(\vector \foo) \rightarrow \text{Return fresh simple vector of } \foo.\n
(\svref \vector i) \rightarrow \text{Element } i \text{ of simple vector. } \text{setfable.}\n
(\vector-push \foo \vector) \rightarrow \text{Return } \text{NIL if vector's fill pointer equals size of vector. Otherwise replace element of vector pointed to by fill pointer with } \foo; \text{ then increment fill pointer.}\n
(\vector-push-extend \foo \vector \num) \rightarrow \text{Replace element of vector pointed to by fill pointer with } \foo; \text{ then increment fill pointer. Extend vector's size by } \num \text{ if necessary.}\n
(\vector-pop \vector) \rightarrow \text{Return element of vector its fill pointer points to after decrementation.}\n
(\fill-pointer \vector) \rightarrow \text{Fill pointer of vector. } \text{setfable.}\n
6 Sequences

6.1 Sequence Predicates

\{\every \test \sequence^{+}\} \rightarrow \text{Return } \text{NIL or } T \text{ respectively, as soon as } \test \text{ on any set of } \text{corresponding elements of } \text{sequences returns } \text{NIL.}\n
\{\atom \foo\} \rightarrow \text{Return } T \text{ if } \foo \text{ is not a } \text{cons.}\n
\{\tailp \\text{list}\} \rightarrow \text{Return } T \text{ if } \text{foo is a tail of list.}\n
\{\member \foo \\text{list}\} \rightarrow \text{Return } T \text{ if } \foo \text{ is a member of list.}\n
\{\member-if \\{\member-if-not\}\} \rightarrow \text{Return tail of list starting with its first element matching } \foo. \text{ Return } \text{NIL} \text{ if there is no such element.}\n
\{\subset \\text{list-a} \\text{list-b}\} \rightarrow \text{Return } T \text{ if list-a is a subset of list-b.}\n
4.2 Lists

\{\cons \foo \bar\} \rightarrow \text{Return new cons (} \foo. \bar)\text{.}\n
\{\\text{list} \\foo\} \rightarrow \text{Return list of } \foo.\n
\{\\text{list*} \\foo\} \rightarrow \text{Return list of } \foo \text{ with last } \foo \text{ becoming cdr of last cons. Return } \text{NIL} \text{ if only one } \foo \text{ given.}\n
\{\\text{make-list} \\num \{\\text{initial-element} \\foo\}\} \rightarrow \text{New list with } \num \text{ elements set to } \foo.\n
\{\\text{list-length} \\text{list}\} \rightarrow \text{Length of list; } \text{NIL} \text{ for circular list.}\n
\{\\text{car} \\text{list}\} \rightarrow \text{Car of list or } \text{NIL} \text{ if list is } \text{NIL. } \text{setfable.}\n
\{\\text{cdr} \\text{list}\} \rightarrow \text{Cdr of list or } \text{NIL} \text{ if list is } \text{NIL. } \text{setfable.}\n
\{\\text{rest} \\text{list}\} \rightarrow \text{Return tail of list after calling } \\text{cdr} \ n \text{ times.}\n
\{\{\text{first}\} \{\text{second}\} \{\text{third}\} \{\text{fourth}\} \{\text{fifth}\} \{\text{sixth}\} \ldots \{\text{ninth}\} \{\text{tenth}\} \\text{list}\} \rightarrow \text{Return nth element of list if any; or } \text{NIL} \text{ otherwise. } \text{setfable.}\n
\{\\text{n-th} \\\text{n} \\text{list}\} \rightarrow \text{Zero-indexed nth element of list. } \text{setfable.}\n
\{\\text{cXr} \\text{list}\} \rightarrow \text{With } X \text{ being one to four } \text{as and } \text{ds representing } \text{cars} \text{ and } \text{cdrs}, \text{ e.g. } \{\\text{cadr} \\text{bar}\} \text{ is equivalent to } \{\\text{car} \{\\text{cdr} \\text{bar}\}\}. \text{ } \text{setfable.}\n
\{\\text{last} \\text{list} \\text{num}\} \rightarrow \text{Return list of last } \text{num} \text{ conses of list.}\n
\{\\text{rbutlast} \\text{list} \\text{num}\} \rightarrow \text{list excluding last } \text{num} \text{ conses.}\n
\{\\text{rplaca} \\text{rplacd} \\text{cons} \text{object}\} \rightarrow \text{Replace car, or cdr, respectively, of } \text{cons} \text{ with } \text{object.}\n
\{\\text{ldiff} \\text{list} \foo\} \rightarrow \text{If } \foo \text{ is a tail of list, return preceding part of list. Otherwise return list.}\n
\{\\text{adjoin} \foo \\text{list}\} \rightarrow \text{Return list if } \foo \text{ is already member of list. If not, return } \{\\text{cons} \foo \text{list}\}.\n
\{\\text{pop} \text{place}\} \rightarrow \text{Set place to } \{\\text{cdr} \text{place}\}, \text{ return } \{\\text{car} \text{place}\}.
Common Lisp Quick Reference

4.3 Association Lists

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

acons key value alist

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

assoc foo list

\(\triangleright\) Return \texttt{list} with \texttt{foo} as the key.

assoc-if-not test alist

\(\triangleright\) First cons whose car, or cdr, respectively, satisfies \texttt{test}.

copy-alist alist

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

4.4 Trees

tree-equal foo bar \(\triangleright\) Return \texttt{T} if trees \texttt{foo} and \texttt{bar} have same shape and leaves satisfying \texttt{test}.

subst new old tree

\(\triangleright\) Make copy of \texttt{tree} with each subtree or leaf matching \texttt{old} replaced by \texttt{new}.

substit-if-not new test tree

\(\triangleright\) Make copy of \texttt{tree} with each subtree or leaf matching \texttt{test} replaced by \texttt{new}.

4.5 Sets

\(\triangleright\) Make copy of \texttt{tree} with each subtree or leaf matching a key in \texttt{association-list} replaced by that key’s value.

\(\triangleright\) Copy of \texttt{tree} with same shape and leaves.

5 Arrays

5.1 Predicates

\(\triangleright\) If \texttt{foo} is of indicated type.

\(\triangleright\) \texttt{T} if \texttt{array} is adjustable/has a fill pointer, respectively.

\(\triangleright\) Return \texttt{T} if \texttt{subsripts} are in \texttt{array}’s bounds.

5.2 Array Functions

\(\triangleright\) Return array element pointed to by \texttt{subsripts}. setfable.

\(\triangleright\) Return \texttt{T} if \texttt{array} is row-major order. setfable.

\(\triangleright\) Index in row-major order of the element denoted by \texttt{subsripts}.

\(\triangleright\) List containing the lengths of \texttt{array}’s dimensions.
8 Structures

(defstruct foo [slot-predicate slot-locator])

(defstruct foo (slot-predicate slot-locator))

(copy-structure foo)

(=copy-structure structure)  
> Return copy of structure with shared slot values.

9 Control Structure

9.1 Predicates

(eq foo bar)  
> T if foo and bar are identical.

(eqv foo bar)  
> T if foo and bar are identical, or the same character, or numbers of the same type and value.

(equal foo bar)  
> T if foo and bar are equalp, or are equivalent pathnames, or are conses with equal elements and cdrs, or are strings or bit-vectors with eqal elements below their fill pointers.

(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.

6.2 Sequence Functions

(make-sequence sequence-type size \[initial-element foo\])  
> 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 \[start end\])  
> Return number of elements in sequence which match foo.

(count-if \[count-if-not\] test sequence \[from-end \\]}  
> Return number of elements in sequence which satisfy test.

(eql sequence index)  
> Return element of sequence pointed to by zero-indexed index. setfable.

(subseq sequence start \[end\])  
> Return subsequence of sequence between start and end. setfable.

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

(reverse sequence)  
> Return sequence in reverse order.
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.

replace sequence-a sequence-b

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

map type function sequence
dep

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

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.
(define-symbol-macro foo form)

▷ Define symbol macro foo which on evaluation evaluates expanded form.

(macrolet ((foo (macro-λ*) \{\(\text{declare local-declt\(\*)}\话\}) macro-form\(\]*)\)})

(\(\text{declare decl}t\)*) form\(\]*)\})

▷ Evaluate forms with locally defined mutually invisible macros foo which are enclosed in implicit \(\text{blocks}\) of the same name.

(symbol-macrolet ((foo expansion-form\(\]*)\}) (\(\text{declare decl}t\)*) form\(\]*)\})

▷ Evaluate forms with locally defined symbol macros foo.

(mdefsetf function \{updater doc\}

\{\(\text{set}-\lambda\)\*(s-var*) \{\(\text{declare decl}t\)*) form\(\]*)\})

where defsetf lambda list (\(\text{set}-\lambda\)\*) has the form

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

\{\&key \{\&supplied-p\} \&rest var\}

\{\&allow-other-keys\} \&environment var\}

▷ Specify how to set\(\) a place accessed by function. Short form: (\(\text{setf}\) function arg\*) value-form is replaced by (updater arg\* value-form); the latter must return value-form. Long form: on invocation of (\(\text{setf}\) function arg\*) value-form, forms must expand into code that sets the place accessed where \(\text{set}-\lambda\)\* 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 \(\text{block}\) named function.

(define-setf-expander function (macro-λ*) \{\(\text{declare decl}t\)*) form\(\]*)\})

▷ Specify how to set\(\) a place accessed by function. On invocation of (\(\text{setf}\) function arg\*) value-form, form\* must expand into code returning arg-vars, args, newval-vars, set-form, and get-form as described with \(\text{get-setf-expansion}\) where the elements of macro lambda list macro-λ\* are bound to corresponding args. forms are enclosed in an implicit \(\text{block}\) named function.

(get-setf-expansion place \{environment\})

▷ Return lists of temporary variables arg-vars and of corresponding 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 arg-vars and newval-vars how to set\(\) and how to read place.

(define-modify-macro foo \{\(\text{optional \{\&optional \{\&rest var\} \{var [init [\&supplied-p]\}}\})\} \&rest var\}) function \{doc\}

▷ 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.

• lambda-list-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.

9.2 Variables

\(\text{defconstant}\) foo form \{doc\}

▷ Assign value of form to global constant/dynamic variable foo.

(\(\text{defvar}\) foo \{form \{doc\}\})

▷ Unless bound already, assign value of form to dynamic variable foo.

\{\(\text{setf}\) \&rest\&setq\} \{place form\*\}

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

\{\(\text{setq}\) \&rest\&setq\} \{symbol form\*\}

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

\(\text{set symbol foo}\)

▷ Set symbol’s value cell to foo. Deprecated.

\(\text{multiple-value-setq}\) vars form

▷ Set elements of vars to the values of form. Return form’s primary value.

\(\text{shift}\) place\* + foo

▷ Store value of foo in rightmost place shifting values of places left, returning first place.

\(\text{rotate}\) place\*

▷ Rotate values of places left, old first becoming new last place’s value. Return \(\text{NIL}\).

\(\text{makunbound}\) foo

▷ Delete special variable foo if any.

\(\text{get}\) symbol key \{default\}

\(\text{set}\) place key \{default\}

▷ First entry key from property list stored in symbol/in place, respectively, or default if there is no key. set\(\text{table}\).

\(\text{get-properties}\) property-list keys

▷ 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 \(\text{NIL}\), \(\text{NIL}\) and \(\text{NIL}\) if there was no matching key in property-list.

\(\text{remprop}\) symbol key

▷ Remove first entry key from property list stored in symbol/in place, respectively. Return \(\text{T}\) if key was there, or \(\text{NIL}\) otherwise.

\(\text{progn}\) symbol values form\(\]*)

▷ Evaluate forms with locally established dynamic bindings of symbols to values or \(\text{NIL}\). Return values of forms.
9.3 Functions

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

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

\$supplied-p\$ is \(T\) if there is a corresponding argument. \textit{init} forms can refer to any \textit{init} and \textit{supplied-p} to their left.

```lisp
(multiple-value-bind \(\text{vars}\) \(\text{values-form}\) (declare \(\text{decl}\)) \(\text{body-form}\))
```

- Evaluate forms with \textit{names} lexically bound (in parallel or sequentially, respectively) to \textit{values}. Return values of \textit{forms}.

```lisp
(multiple-value-bind \(\text{ddestructuring-bind}\) \(\lambda\) \(\text{bar}\) (declare \(\text{decl}\)) \(\text{form}\))
```

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

9.4 Macros

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

```lisp
([&whole var] \[E] \[var (macro-\*)] \[E])
```

- Multiple values can refer to any \textit{init} and \textit{supplied-p} to their left.

```lisp
[&optional \[\{var (macro-\*) \[init \(\text{supplied-p}\)\] \[rest \text{var}\]\}] \[E]]
```

- If \textit{var} is one of \texttt{aref}, \texttt{bit}, and \texttt{sbif}.

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

- Return values of \textit{function} called with \textit{args}.

```lisp
[[\text{multiple-value-call} \{\text{function} \text{form}\}]}
```

- Call \textit{function} with all the values of each \textit{form} as its arguments. Return values returned by \textit{function}.

```lisp
[[\text{values-list} \text{list}]
```

- Return elements of \textit{list}.

```lisp
[[\text{values} \text{foo}]
```

- Return as multiple values the primary values of the \textit{foos}. \texttt{setfable}.

```lisp
[[\text{multiple-value-list} \text{form}]
```

- List of the values of \textit{form}.

Returns new function with same arguments and same side effects as \textit{function}, but with complementary truth value.

- Functions of any number of arguments returning \textit{foo}.

- Return \textit{foo}.

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

- Remove global function or macro definition \textit{foo}.

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

- Upper bound of the number of values a multiple value can have; \(\geq 20\).
(upfrom|from|downfrom) start
  ▷ Start stepping with start
(upto|downto|to|below|above) form
  ▷ Specify form as the end value for stepping.
(in|on) list
  ▷ Bind var to successive elements/tails, respectively, of list.
by {step|function} form
  ▷ Specify the (positive) decrement or increment or the function of one argument returning the next part of the list.
= foo [then] bar [else]
  ▷ Bind var initially to foo and later to bar.
across vector
  ▷ Bind var to successive elements of vector.
being {the|each}
  ▷ Iterate over a hash table or a package.
{hash-key|hash-keys} {of|in} hash-table [using (hash-value value)]
  ▷ Bind var successively to the keys of hash-table; bind value to corresponding values.
{hash-value|hash-values} {of|in} hash-table [using (hash-key key)]
  ▷ Bind var successively to the values of hash-table; bind key to corresponding keys.
{symbol|symbols|present-symbol|present-symbols|external-symbol|external-symbols} {{of|in} package |package}
  ▷ Bind var successively to the accessible symbols, or the present symbols, or the external symbols respectively, of package.
{do|doing} form+
  ▷ Evaluate forms in every iteration.
{if|when|unless} test i-clause {and|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.
return {form|it}
  ▷ Return immediately, skipping any finally parts, with values of form or it.
{collect|collecting} {form|it} into list
  ▷ Collect values of form or it into list. If no list is given, collect into an anonymous list which is returned after termination.
{append|appending} nconc nconcing {form|it} into list
  ▷ Concatenate values of form or it, which should be lists, into list by the means of ,append or ,nconc, respectively. If no list is given, collect into an anonymous list which is returned after termination.
{count|counting} {form|it} into n [type]
  ▷ Count the number of times the value of form or of it is T. If no n is given, count into an anonymous variable which is returned after termination.
{sum|summing} {form|it} into sum [type]
  ▷ Calculate the sum of the primary values of form or of it. If no sum is given, sum into an anonymous variable which is returned after termination.
(maximize|maximizing) minimize |minimizing) {form|it} into max-min [type]
  ▷ Determine the maximum or minimum, respectively, of the primary values of form or of it. If no max-min is given, use an anonymous variable which is returned after termination.

&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 test) form
  ▷ Evaluate foos and return their values if test returns T or NIL, respectively. Return NIL otherwise.
(munless test) form
  ▷ Evaluate foos and return their values if test returns T or NIL, respectively. Return NIL otherwise.
(mcase test (key+) form+) {otherwise bar+}
  ▷ Return the values of the first foo+ one of whose keys is eql test. Return values of bars if there is no matching key.
(mcase test (key+) foo+)
  ▷ Return the values of the first foo+ one of whose keys is eql test. Signal non-correctable/correctable type-error if there is no matching key.
(mand form+)
  ▷ Evaluate forms from left to right. Immediately return NIL if any form’s value is NIL. Return values of last form otherwise.
(mor 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*)
(multiple-value-prog1 form-r form*)
  ▷ Evaluate forms in order. Return values/primary value, respectively, of form-r.
(mprog) {name (name value)} {declare decl=} {tag form}
  ▷ 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, cleanups. 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)
(mreturn 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 \texttt{go}. Return NIL.

(go tag)
  ⊳ Within the innermost possible enclosing \texttt{tagbody}, jump to a tag \texttt{eq} tag.

(catch tag form*)
  ⊳ Evaluate forms and return their values unless interrupted by \texttt{throw}.

(throw tag form)
  ⊳ Have the nearest dynamically enclosing \texttt{catch} with a tag \texttt{eq} tag return with the values of \texttt{form}.

(sleep n)
  ⊳ Wait \texttt{n} seconds; return NIL.

9.6 Iteration

\begin{equation}
\begin{aligned}
\texttt{do} & \quad \{ \texttt{var } \{ \texttt{[start [step]]} \}\} \quad \texttt{(stop result)} \quad \texttt{(declare decl*)} \quad \texttt{tagbody} \quad \texttt{form} \\
\texttt{dos} & \quad \{ \texttt{tag body} \quad \texttt{form} \}
\end{aligned}
\end{equation}

Evaluate \texttt{tagbody} like body with \texttt{var}s successively bound according to the values of the corresponding \texttt{start} and \texttt{step} \texttt{form}s. \texttt{var}s are bound in parallel/sequentially, respectively. Stop iteration when \texttt{stop} is \texttt{T}. Return values of \texttt{result}. Implicitly, the whole form is a \texttt{block} named NIL.

\begin{equation}
\begin{aligned}
\texttt{do-times} & \quad \{ \texttt{var i [result]} \} \quad \texttt{(declare decl*)} \quad \texttt{tag body} \quad \texttt{form} \\
\end{aligned}
\end{equation}

Evaluate \texttt{tagbody} like body with \texttt{var} successively bound to integers from 0 to \texttt{i} – 1. Upon evaluation of \texttt{result}, \texttt{var} is \texttt{i}. Implicitly, the whole form is a \texttt{block} named NIL.

\begin{equation}
\begin{aligned}
\texttt{do-list} & \quad \{ \texttt{var list [result]} \} \quad \texttt{(declare decl*)} \quad \texttt{tag body} \quad \texttt{form} \\
\end{aligned}
\end{equation}

Evaluate \texttt{tagbody} like body with \texttt{var} successively bound to the elements of \texttt{list}. Upon evaluation of \texttt{result}, \texttt{var} is NIL. Implicitly, the whole form is a \texttt{block} named NIL.

9.7 Loop Facility

\begin{equation}
\texttt{loop (form*)}
\end{equation}

▷ Simple Loop. If \texttt{forms} do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit \texttt{block} named NIL.

\begin{equation}
\texttt{loop clause*)}
\end{equation}

▷ Loop Facility. For Loop Facility keywords see below and Figure 1.

\texttt{named n}

▷ Give \texttt{n}’s implicit \texttt{block} a name.

\begin{equation}
\begin{aligned}
\texttt{with} & \quad \{ \texttt{var-s} \} \quad \{ \texttt{d-type} \} \quad \texttt{[= foo]} \\
\texttt{and} & \quad \{ \texttt{var-p} \} \quad \{ \texttt{d-type} \} \quad \texttt{[= bar]} \\
\end{aligned}
\end{equation}

where destructuring type specifier \texttt{d-type} has the form

\begin{equation}
\begin{aligned}
\texttt{fixnum} & \quad \texttt{float} \quad \texttt{nil} \quad \texttt{of-type} \quad \{ \texttt{type} \} \quad \texttt{NIL} \\
\end{aligned}
\end{equation}

▷ Initialize (possibly trees of) local variables \texttt{var-s} sequentially and \texttt{var-p} in parallel.

\begin{equation}
\begin{aligned}
\texttt{for-as} & \quad \{ \texttt{var-s} \} \quad \{ \texttt{d-type} \} \quad \texttt{[= foo]} \\
\texttt{and} & \quad \{ \texttt{var-p} \} \quad \{ \texttt{d-type} \} \quad \texttt{[= bar]} \\
\end{aligned}
\end{equation}

▷ Begin of iteration control clauses. Initialize and step (possibly trees of) local variables \texttt{var-s} sequentially and \texttt{var-p} in parallel. Destructuring type specifier \texttt{d-type} as with \texttt{with}.
10.3 Method Combination Types

standard

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

(and/or append list nconc nconc prog max min +)

- Simple built-in method-combination types; have the same usage as the c-types defined by the short form of define-method-combination.

(defun-method-combination c-type
  \{ :documentation string
       :identity-with-one-argument bool
       :operator operator
     \})

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

(defun-method-combination c-type (ord-lambda) ((group
  \{ \[ qualifier \[ * \] \] \}
  \{ predicate \}
  \{ :description control \}
  \{ :order \{ most-specific-first \} \}
  \{ :required bool \}
  \{ \{ arguments method-combination\-lambda \} \}
  \{ \{ generic-function symbol \} \}
  \{ \{ declare decl \} \}
  \{ \{ doc \} \}
\})

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

# 10 CLOS

10.1 Classes

(slot-exists-p foo bar)

- If foo has a slot bar.

(slot-boundp instance slot)

- If slot in instance is bound.

(defclass foo (superclass* standard-object)
  \{ :reader reader
       :writer \{ (setf writer) \} \}
  \{ :accessor accessor \}
  \{ slots \}
  \{ function \}
  \{ :documentation slot-doc \}
)

- Define or modify class foo as a subclass of superclasses. Transform existing instances, if any, by make-instances-obsolete. In a new instance i of foo, a slot’s value defaults to form unless set via \{ initial-value \}; it is readable via \{ reader \} or \{ accessor \}, and writable via \{ writer \} or \{ setf \} \{ accessor \} \{ value \}. Slots with \{ allocation \} \{ class \} are shared by all instances of class foo.

(find-class symbol [error [environment]])

- Return class named symbol. settable.

(make-instance class \{ initial-value \} other-keyarg*)

- Make new instance of class.

(reinitialize-instance instance \{ initial-value \} other-keyarg*)

- Change local slots of instance according to initargs by means of shared-initialize.

(slot-value foo slot)

- Return value of slot in foo, settable.

(slot-makunbound instance slot)

- Make slot in instance unbound.
Class methods suitable for c-arg values

See section 10.2 for details.

Generic Functions

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 defmethod. For c-type see section 10.3.

Generic Functions

Define new method for generic function foo. spec-vars specialize to either being of class or being eql bar, respectively. On invocation, vars and spec-vars of the new method act like parameters of a function with body form*; forms are enclosed in an implicit ,block foo. Applicable qualifiers depend on the method-combination type; see section 10.3.

Add (if necessary) or remove (if any) method to/from generic-function.

Return suitable method, or signal error.

List of methods suitable for args, most specific first.

From within a method, call next method with args; return its values.

Called on invocation of generic-function arg* on args if there is no applicable method. Default method signals error. Not to be called by user.

Called on invocation of next-method when there is no next method. Default method signals error. Not to be called by user.
11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 32.

\[
\text{define-condition } \texttt{foo} \ (\text{parent-type}\texttt{')} \ \\
\text{slot} \ \{\text{reader reader}'} \ \\
\{\text{writer } (\text{self writer}')\} \ \\
\{\text{accessor accessor}'\} \ \\
\{\text{allocation } \text{instance}\} \ \\
\{\text{class } \text{class}\} \\
\{\text{initarg } \text{initarg-name}'\} \\
\{\text{type } \text{type}\} \\
\{\text{documentation } \text{slot-doc}\} \\
\{\text{default-initargs } \{\text{name-value}'\}\} \\
\{\text{documentation } \text{condition-doc}\} \\
\{\text{report } \text{report-function}'\} \\
\]

\> Define, as a subtype of parent-types, condition type foo. In a new condition, a slot's value defaults to form unless set via \{\text{initarg-name}\}; it is readable via (reader i) or (accessor i), and writable via (writer value i) or (setf accessor i) value.

With \text{allocation} : \text{class}, slot is shared by all conditions of type foo. A condition is reported by \text{string} or by report-function of arguments condition and stream.

\[
\text{make-condition } \text{condition-type} \ (\{\text{initarg-name value}'\}) \\
\]

\> Return new instance of condition-type.

\[
\text{signal} \ (\text{condition-type} \ \{\text{initarg-name value}'\}) \ \\
\text{warn} \ (\text{condition-type} \ \{\text{initarg-name value}'\}) \\
\]

\> Unless handled, signal as condition, warning or error, respectively, condition or a new instance of condition-type or, with \text{format} control and args (see page 38), \text{simple-condition}, \text{simple-warning}, or \text{simple-error}, respectively. From \text{signal} and \text{warn}, return \texttt{NIL}.

\[
\text{error} \ (\text{condition-type} \ \{\text{initarg-name value}'\}) \\
\]

\> Unless handled, signal as correctable \text{error} condition or a new instance of condition-type or, with \text{format} control and args (see page 38), \text{simple-error}. In the debugger, use \text{format} arguments \text{continue-control} and \text{continue-args} to tag the continue option. Return \texttt{NIL}.

\[
\text{ignore-errors form}' \\
\]

\> Return values of forms or, in case of \texttt{errors}, \texttt{NIL} and the condition.

\[
\text{invoke-debugger } \text{condition} \\
\]

\> Invoke debugger with condition.

\[
\text{assert} \ [\text{test} \ \{\text{place}'\} \ \\
\{\text{condition-type} \ \{\text{initarg-name value}'\}\}] \ \\
\{\text{control arg}'\} \\
\]

\> If \text{test}, which may depend on \text{places}, returns \texttt{NIL}, signal as correctable \text{error} condition or a new instance of condition-type or, with \text{format} control and args (see page 38), \text{error}. When using the debugger’s continue option, \text{places} can be altered before re-evaluation of \text{test}. Return \texttt{NIL}.
(handler-case foo (type ([var]) (declare decl*)* condition-form)^* ([no-error (ord-^*) (declare decl*)* 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-^* to values of foo and return values of form^* or, without a no-error clause, return values of foo. See page 18 for (ord-^*).

(handler-bind ([(condition-type handler-function)*] form^*)
  ➤ Return values of form after evaluating them with condition-types dynamically bound to their respective handler-functions of argument condition.

(with-simple-restart ([restart NIL] control arg^*) form^*)
  ➤ Return values of form unless restart is called during their evaluation. In this case, describe restart using format control and args (see page 38) and return NIL and T.

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

(restart-bind ([{restart NIL} restart-function
  (declare decl*)* restart-form)^* form^*)
  ➤ 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^*)
  ➤ 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 [condition]
    {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 form^*.

(arithmetic-error-operation condition)
  ➤ List of function or of its operands respectively, used in the operation which caused condition.

(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 readable printable under condition.

(package-error-package condition)

(file-error-pathname condition)

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

(type-error-datum condition)

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

(simple-condition-format-control condition)

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

•break-on-signals
  ➤ Condition type debugger is to be invoked on.

•debugger-hook
  ➤ Function of condition and function itself. Called before debugger.

12 Types and Classes

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

(typep foo type [environment]) ➤ 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.

(typep foo type [environment])
  ➤ Declare values of form to be of type.

(coerce object type)
  ➤ Coerce object into type.

(typep foo (type a-form)^* ([otherwise b-form]^*))
  ➤ Return values of the first a-form^* whose type is foo of. Return values of b-forms if no type matches.

(typep foo (type b-form)^* )
  ➤ Return values of the first form^* whose type is foo of. Signal non-corretable/correctable type-error if no type matches.
Common Lisp Quick Reference

13.4 Printer

(print) → Print object to standard output.
(print stream) → Print to stream.
(print object stream) → Print object to stream.
(print-unreadable-object form stream) → Enclosed in #< and >, print form by means of forms to stream.
(sipri stream) → Output a newline to stream. Return NIL.
(fresh-line stream) → Output a newline to stream and return T unless stream is already at the start of a line.
(write-char char stream) → Output char to stream.
(write-string string stream) → Write string to stream without/with a trailing newline.
(write-byte byte stream) → Write byte to binary stream.

13 Input/Output

13.1 Predicates

(stream foo) → T if foo is of indicated type.
(pathnamep foo) → T if foo is of indicated type.
(readablep foo) → T if foo is of indicated type.
(input-stream-p stream) → T if stream is for input, for output, interactive, or open, respectively.
(interactive-stream-p stream) → T if stream is for input, for output, interactive, or open, respectively.
(open-stream-p stream) → T if stream is for input, for output, interactive, or open, respectively.
(pathname-match-p path wildcard) → T if path matches wildcard.
(wildpathnamep path) → T if indicated component in path is wildcard. (NIL indicates any component.)
13.3 Character Syntax

#| multi-line-comment | #
; one-line-comment

| foo | List of foos with the terminating cdr bar.
| " | Begin and end of a string.


\{ | Complex number \a + \bi.
\# | n-dimensional array.
\#[a b] | Vector of some (or \na) foos filled with last \foo if necessary.
Logical Block. Act like `pprint-logical-block` using `body` as the control string on the elements of the list argument or, with `@`, on the remaining arguments, which are extracted by `pprint-pop`. With `@`, `prefix` and `suffix` default to ( and ). When closed by `@>`, spaces in `body` are replaced with conditional newlines.

- (-1 | @ | [<prefix | ->] | [per-line-prefix | -]) body | [-; suffix | -] | (@ | >)
  ▶  *Indent.* Set indentation to n relative to leftmost/to current position.

- (+0 | @ | [i | | @ | ] | T)
  ▶  *Tabulate.* Move cursor forward to column number c+ki, k ≥ 0 being as small as possible. With `i`, calculate column numbers relative to the immediately enclosing section. With `@`, move to column number c0 + c + ki where c0 is the current position.

- (-1 | [m | [n | @ | ] | -] | [n | @ | ] | @)
  ▶  *Go-To.* Jump m arguments forward, or backward, or to argument n.

- ([limit] | @ | [text | -])
  ▶  *Iteration.* Use text repeatedly, up to limit, as control string for the elements of the list argument or (with `@`) for the remaining arguments. With `i` or `@i`, list elements or remaining arguments should be lists of which a new one is used at each iteration step.

- (+0 | [x | y | [z | i | ]])
  ▶  *Escape Upward.* Leave immediately < - >, < - i >, - ( - ), - ?, or the entire `format` operation. With one to three prefixes, act only if x = 0, x = y, or x ≤ y ≤ z, respectively.

- (-1 | [i | [i | @ | [([text | -])] | text | -; | default | -])
  ▶  *Conditional Expression.* Use the zero-indexed argument (or ith if given) text as a `format` control subclause. With `i`, use the first `text` if the argument value is `NIL`, or the second `text` if it is `T`. With `@`, do nothing for an argument value of `NIL` unless only `text` and leave the argument to be read again if it is `T`.

- (-1 | @ |)
  ▶  *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.

- [+0 | [prefix | [text | -])] | [i | @ | ] | [package | [i | ] | preferred | ] | function | /
  ▶  *Call Function.* Call all-upper-case `package::function` with the arguments stream, format-argument, colon-p, at-sign-p and prefixes for printing format-argument.

- (-1 | @ | W)
  ▶  *Write.* Print argument of any type obeying every printer control variable. With `@`, pretty-print. With `@`, print without limits on length or depth.

- (-1 | @ | #)
  ▶  In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.
13.5 Format

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

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

- [min-col]\{\{col-in\}\{min-pad\}\{pad-char\}\}
  | \{\} \{l\}\{S\}
> 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-chars on the left rather than on the right.

- [radix]\{width\}\{\{\}pad-char\}\{comma-char\}\{comma-interval\}\}
  \{0\} \{R\}
> Radix. (With one or more prefix arguments.) Print argument as number; with \, group digits comma-interval each; with \, always prepend a sign.

- [R]\{R\} \{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]\{\{\}pad-char\}\{comma-char\}\{comma-interval\}\}
  \{0\} \{0\} \{0\} \{0\} \{0\} \{X\}
> Decimal/Binary/Octal/Hexadecimal. Print integer argument as number. With \, group digits comma-interval each; with \, always prepend a sign.

- [width]\{\{dec-digits\}\{\{\}shift\}\{\{\}overflow-char\}\{\{\}pad-char\}\}\}
  \{0\} \{F\}
> Fixed-Format Floating-Point. With \, always prepend a sign.

- [width]\{\{dec-digits\}\{\{\}exp-digits\}\{\{\}scale-factor\}\{\{\}overflow-char\}\{\{\}pad-char\}\}\}
  \{0\} \{E\} \{0\}
> Exponential/General Floating-Point. Print argument as floating-point number with dec-digits after decimal point and exp-digits in the signed exponent. With \, choose either \ or \. With \, always prepend a sign.

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

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

- [\{text\}]{\{text\}} \{0\} \{text\} \{0\} \{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\} \{0\} \{P\}
> Plural. If argument eql 1 print nothing, otherwise print s; do the same for the previous argument; if argument eql 1 print y, otherwise print ies; do the same for the previous argument, respectively.

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

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

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

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

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

- [min-col]\{\{col-in\}\{min-pad\}\{\}pad-char\}\}
  \{0\} < [nl-text \{\} \{\} | \{\} text \->
> Justification. Justify text produced by texts in a field of at least min-col columns. With \, right justify; with \, left justify. If this would leave less than spare characters on the current line, output nl-text first.
14 Packages and Symbols

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

14.1 Predicates

(defun (symbolp foo) ⇒ T if foo is of indicated type.
(defun (packagep foo) ⇒ T if foo is of indicated type.
(defun (keywordp foo) ⇒ T if foo is of indicated type.

14.2 Packages

(defvar (keyword:bar) ⇒ Keyword, evaluates to :bar.
(defun (package:symbol) ⇒ Exported symbol of package.
(defun (package:shadowing-symbol) ⇒ Possibly unexported symbol of package.
(defun (package-nicknames package) ⇒ Nicknames of package.

(defun (make-package foo) ⇒ Create package foo.
(defun (min-package foo) ⇒ Make package foo current.

(defun (use-package other-packages package) ⇒ Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T.
(defun (package-use-list package) ⇒ List of other packages used by/using package.
(defun (delete-package package) ⇒ Delete package. Return T if successful.

(*package:*common-lisp-user) ⇒ The current package.
(defun (list-all-packages) ⇒ List of registered packages.
(defun (package-name package) ⇒ Name of package.
(defun (package-nicknames package) ⇒ Nicknames of package.

13.6 Streams

(defun (make-concatenated-stream input-stream *) ⇒ Create or modify package foo with interned-symbols, symbols from used-packages, imported-symbols, and std-symbols. Add std-symbols to foo’s shadowing list.
(defun (make-package foo) ⇒ Create package foo.
(defun (min-package foo) ⇒ Make package foo current.

(defun (use-package other-packages package) ⇒ Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T.
(defun (package-use-list package) ⇒ List of other packages used by/using package.
(defun (delete-package package) ⇒ Delete package. Return T if successful.

(*package:*common-lisp-user) ⇒ The current package.
(defun (list-all-packages) ⇒ List of registered packages.
(defun (package-name package) ⇒ Name of package.
(defun (package-nicknames package) ⇒ Nicknames of package.

(defun (open path) ⇒ Open file-stream to path.
(defun (make-concatenated-stream input-stream *) ⇒ Create or modify package foo with interned-symbols, symbols from used-packages, imported-symbols, and std-symbols. Add std-symbols to foo’s shadowing list.
(defun (make-package foo) ⇒ Create package foo.
(defun (min-package foo) ⇒ Make package foo current.

(defun (use-package other-packages package) ⇒ Make exported symbols of other-packages available in package, or remove them from package, respectively. Return T.
(defun (package-use-list package) ⇒ List of other packages used by/using package.
(defun (delete-package package) ⇒ Delete package. Return T if successful.

(*package:*common-lisp-user) ⇒ The current package.
(defun (list-all-packages) ⇒ List of registered packages.
(defun (package-name package) ⇒ Name of package.
(defun (package-nicknames package) ⇒ Nicknames of package.
13.7 Pathnames and Files

```lisp
(defun merge-pathnames path-or-stream default-pathname-defaults)
  (return pathname made by filling in components missing in
  path-or-stream from default-path-or-stream.

(defun *default-pathname-defaults)
  (pathname to use if one is needed and none supplied.

(defun user-homedir-pathname [host])
  (user's home directory.

(defun enough-namestring path-or-stream root-path)
  (return minimal path string that sufficiently describes the
  path of path-or-stream relative to root-path.

(defun namestring path-or-stream)
  (path of path-or-stream)

(defun directory-namestring path-or-stream)
  (directory namestring of path-or-stream)

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

(defun 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.

(defun pathname path-or-stream)
  (pathname of path-or-stream.

(defun logical-pathname logical-path-or-stream)
  (logical pathnames of logical-path-or-stream. Logical
  pathnames are represented as all-uppercase
  "[host:][[dir]++][name]+ [type]+ [version]+ [latest NEAREST]*".

(defun logical-pathname-translations logical-host)
  (list of (from-wildcard-to-wildcard) translations for
  logical-host. set/able.

(defun load-logical-pathname-translations logical-host)
  (load logical-host's translations. Return NIL if already
  loaded; return T if successful.

(defun translate-logical-pathname path-or-stream)
  (physical pathname corresponding to (possibly logical) pathname
  of path-or-stream.

(defun probe-file file)
  (canonical name of file. If file does not exist, return
  NIL/signal file-error, respectively.

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

(defun file-write-date file)
  (time at which file was last written.

(defun file-author file)
  (return name of file owner.

(defun file-length stream)
  (return length of stream.

(defun 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.

(defun delete-file file)
  (delete file. Return T.
```
15.3 REPL and Debugging

+T+++,.+++ 
+*+++.+++ 
+//(+// 

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

> Form currently being evaluated by the REPL.

(apropos string [package]) 
> Print interned symbols containing string.

(apropos-list string [package]) 
> List of interned symbols containing string.

(dribble [path]) 
> Save a record of interactive session to file at path. Without path, close that file.

(red [file-or-function]) 
> Invoke editor if possible.

{macroexpand-1, macroexpand} form [environment] 
> Return macro expansion, once or entirely, respectively, of form and T if form was a macro form. Return form and NIL otherwise.

*macroexpand-hook* 
> Function of arguments expansion function, macro form, and environment called by *macroexpand-1 to generate macro expansions.

{trace, untrace} function [self function] 
> Cause functions to be traced. With no arguments, return list of traced functions.

{trace, untrace} function [self function] 
> Stop functions, or each currently traced function, from being traced.

*trace-output* 
> Output stream *trace* and *time* send their output to.

(step form) 
> Step through evaluation of form. Return values of form.

(break [control arg*]) 
> Jump directly into debugger; return NIL. See page 38, *format*, for control and args.

(time form) 
> Evaluate forms and print timing information to *trace-output*. Return values of form.

(inspect foo) 
> Interactively give information about foo.

(describe foo [stream]) 
> Send information about foo to stream.

(describe-object foo [stream]) 
> Send information about foo to stream. Called by *describe*.

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

(room [NIL][default T])(default) 
> Print information about internal storage management to *standard-output*.

(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] 
> 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]) 
> Remove symbol from package, return T on success.

{import, prompting-import} symbols [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.

{package-shadowing-symbols package} 
> List of symbols of *package* that shadow any otherwise accessible, equally named symbols from other packages.

{export symbols [package]} 
> Make symbols external to package. Return T.

{unexport symbols [package]} 
> Revert symbols to internal status. Return T.

{do-symbols, do-external-symbols} (var [package] [result set]) 
> Evaluate *tagbody*-like body with var successively bound to every symbol from *package*, to every external symbol from *package*, or to every symbol from all registered packages, respectively. Return values of result. Implicitly, the whole form is a *block* named NIL.

(with-package-iterator (foo packages [internal*:external*:inherited]) 
> Return values of forms. In forms, successive invocations of (foo) return: T if a symbol is returned; a symbol from *packages*; accessibility (*internal*, *external*, or *inherited*); and the package the symbol belongs to.

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

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

*modules* 
> List of names of loaded modules.

14.3 Symbols

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

(make-symbol name) 
> Make fresh, uninterned symbol name.
(gensym [n])
  ➤ Return fresh, uninterned symbol #\+n with n from
  ‹gensym-counters›. Increment ‹gensym-counters›.

(gentemp [prefix [package [symbol]]])
  ➤ Intern fresh symbol in package. Deprecated.

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

(symbol-name symbol)
  ➤ Name or package, respectively, of symbol.
(symbol-package symbol)
  ➤ Property list, value, or function, respectively, of symbol.
(symbol-plist symbol)
  ➤ Get/set documentation string of foo of given type.

/documentation (self &documentation) new-doc foo
  ➤ Get/set documentation string of foo of given type.

(t)
  ➤ Truth; the supertype of every type including t; the superclass
  of every class except t; ‹*terminal-ios›.

NIL()
  ➤ 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
  ➤ Return compiled function or replace name's function defini-
  tion with the compiled function. Return T in case of
  warnings or errors, and T in case of warnings or errors excluding
  style-warnings.

(output-file out-path
  ➤ 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 [output-file path] [other-keyargs])
  ➤ Pathname, compile-file writes to if invoked with the same
  arguments.

(load path
  ➤ Load source file or compiled file into Lisp environment. Return
  T if successful.

(*compile-file* [pathname=&path] [true-name=&truepath]
  ➤ Input file used by *compile-file* by \load.

(*compile* \print\
  ➤ Defaults used by *compile-file* by \load.

(*eval-when [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 bool]) form)
  ➤ Return values of forms. Warnings deferred by the compiler
  until end of compilation are deferred until the end of evalua-
  tion of forms.

(load-time-value form [read-only]
  ➤ 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 evalua-
  tion 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] [environment environment]]
  ➤ Return a creation form and an initialization form which on
  evaluation construct an object equivalent to foo with slots ini-
  tialized with the corresponding values from foo.

(macro-function symbol [environment])
  ➤ Return specified macro function, or compiler macro function,
  respectively, if any. Return NIL otherwise. setable.

(eval arg)
  ➤ Return values of value of arg evaluated in global environment.
WHEN 23, 24
WHILE 25
WILD-PATHNAME-P 33
WITH 22
WITH-ACCESSORS 26
WITH-COMPILATION-UNIT 47
WITH-CONDITION-RESTARTS 31
WITH-HASH-ITERATOR 15
WITH-INPUT-FROM-STRING 42
WITH-OPEN-FILE 42
WITH-OPEN-STREAM 42
WITH-OUTPUT-TO-STRING 42
WITH-PACKAGE-ITERATOR 45
WITH-SIMPLE-RESTART 30
WITH-SLOTS 26
WITH-STANDARD-IO-SYNTAX 34
WRITE 37
WRITE-BYTE 36
WRITE-CHAR 36
WRITE-LINE 36
WRITE-SEQUENCE 37
WRITE-STRING 36
WRITE-TO-STRING 37
Y-OR-N-P 34
YES-OR-NO-P 34
ZEROP 3