

Quick Reference

cl

Common

lisp

Bert Burgemeister

Contents

1	Numbers	3	9.5	Control Flow . . .	19
1.1	Predicates	3	9.6	Iteration	21
1.2	Numeric Functns	3	9.7	Loop Facility	21
1.3	Logic Functions	4	10	CLOS	24
1.4	Integer Functions	5	10.1	Classes	24
1.5	Implementation-Dependent	6	10.2	Generic Functns	25
2	Characters	6	10.3	Method Combination Types	26
3	Strings	7	11	Conditions and Errors	27
4	Conses	8	12	Types and Classes	30
4.1	Predicates	8	13	Input/Output	32
4.2	Lists	8	13.1	Predicates	32
4.3	Association Lists	9	13.2	Reader	32
4.4	Trees	10	13.3	Character Syntax	33
4.5	Sets	10	13.4	Printer	34
5	Arrays	10	13.5	Format	36
5.1	Predicates	10	13.6	Streams	39
5.2	Array Functions	10	13.7	Paths and Files	40
5.3	Vector Functions	11	14	Packages and Symbols	42
6	Sequences	12	14.1	Predicates	42
6.1	Seq. Predicates	12	14.2	Packages	42
6.2	Seq. Functions	12	14.3	Symbols	43
7	Hash Tables	14	14.4	Std Packages	44
8	Structures	15	15	Compiler	44
9	Control Structure	15	15.1	Predicates	44
9.1	Predicates	15	15.2	Compilation	44
9.2	Variables	16	15.3	REPL & Debug	45
9.3	Functions	17	15.4	Declarations	46
9.4	Macros	18	16	External Environment	47

Typographic Conventions

name; *f***name**; *g***name**; *m***name**; *s***name**; *v****name***; *c***name**

▷ Symbol defined in Common Lisp; esp. function, generic function, macro, special operator, variable, constant.

them ▷ Placeholder for actual code.

me ▷ Literal text.

[*foo***bar**] ▷ Either one *foo* or nothing; defaults to **bar**.

*foo**; {*foo*}* ▷ Zero or more *foos*.

foo+; {*foo*}+ ▷ One or more *foos*.

foos ▷ English plural denotes a list argument.

{*foo*|*bar*|*baz*}; $\begin{cases} \textit{foo} \\ \textit{bar} \\ \textit{baz} \end{cases}$ ▷ Either *foo*, or *bar*, or *baz*.

$\begin{cases} \textit{foo} \\ \textit{bar} \\ \textit{baz} \end{cases}$ ▷ Anything from none to each of *foo*, *bar*, and *baz*.

$\widehat{\textit{foo}}$ ▷ Argument *foo* is not evaluated.

$\widetilde{\textit{bar}}$ ▷ Argument *bar* is possibly modified.

foo^P* ▷ *foo** is evaluated as in *sprogn*; see page 20.

$\underline{\textit{foo}}$; $\underline{\textit{bar}}$; $\underline{\textit{baz}}$ ▷ Primary, secondary, and *n*th return value.

T; **NIL** ▷ **t**, or truth in general; and **nil** or **()**.

1 Numbers

1.1 Predicates

(*f* = *number*⁺)
 (*f* /= *number*⁺)
 ▷ T if all *numbers*, or none, respectively, are equal in value.

(*f* > *number*⁺)
 (*f* >= *number*⁺)
 (*f* < *number*⁺)
 (*f* <= *number*⁺)
 ▷ Return T if *numbers* are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

(*f* minusp *a*)
 (*f* zerop *a*) ▷ T if *a* < 0, *a* = 0, or *a* > 0, respectively.
 (*f* plusp *a*)

(*f* evenp *int*)
 (*f* oddp *int*) ▷ T if *int* is even or odd, respectively.

(*f* numberp *foo*)
 (*f* realp *foo*)
 (*f* rationalp *foo*)
 (*f* floatp *foo*) ▷ T if *foo* is of indicated type.
 (*f* integerp *foo*)
 (*f* complexp *foo*)
 (*f* random-state-p *foo*)

1.2 Numeric Functions

(*f* + *a*<sub>[□]^{*})
 (*f* * *a*_{[□]^{*}) ▷ Return $\sum a$ or $\prod a$, respectively.}</sub>

(*f* - *a* *b*^{*})
 (*f* / *a* *b*^{*})
 ▷ Return $\frac{a - \sum b}{a}$ or $\frac{a}{\prod b}$, respectively. Without any *bs*, return $\frac{-a}{a}$ or $\frac{1}{a}$, respectively.

(*f* 1+ *a*)
 (*f* 1- *a*) ▷ Return $\frac{a + 1}{a}$ or $\frac{a - 1}{a}$, respectively.

($\left\{ \begin{matrix} m \\ m \end{matrix} \right\} \text{incf}$ *place* [*delta*_[□]])
 ($\left\{ \begin{matrix} m \\ m \end{matrix} \right\} \text{decf}$ *place* [*delta*_[□]])
 ▷ Increment or decrement the value of *place* by *delta*. Return new value.

(*f* exp *p*)
 (*f* expt *b* *p*) ▷ Return $\frac{e^p}{b^p}$ or $\frac{e^p}{b^p}$, respectively.

(*f* log *a* [*b*_[□]]) ▷ Return $\frac{\log a}{\log b}$ or, without *b*, $\frac{\log a}{1}$.

(*f* sqrt *n*)
 (*f* isqrt *n*) ▷ \sqrt{n} in complex numbers/natural numbers.

(*f* lcm *integer*_[□]^{*})
 (*f* gcd *integer*^{*})
 ▷ Least common multiple or greatest common denominator, respectively, of *integers*. (**gcd**) returns 0.

pi ▷ **long-float** approximation of π , Ludolph's number.

(*f* sin *a*)
 (*f* cos *a*) ▷ $\frac{\sin a}{\cos a}$, $\frac{\cos a}{\sin a}$, or $\frac{\tan a}{1}$, respectively. (*a* in radians.)
 (*f* tan *a*)

(*f* asin *a*)
 (*f* acos *a*) ▷ $\frac{\arcsin a}{1}$ or $\frac{\arccos a}{1}$, respectively, in radians.

(*f* atan *a* [*b*_[□]]) ▷ $\frac{\arctan \frac{a}{b}}{1}$ in radians.

(*f* sinh *a*)
 (*f* cosh *a*) ▷ $\frac{\sinh a}{\cosh a}$, $\frac{\cosh a}{\sinh a}$, or $\frac{\tanh a}{1}$, respectively.
 (*f* tanh *a*)

- (*f* **asinh** *a*)
(*f* **acosh** *a*) ▷ asinh *a*, acosh *a*, or atanh *a*, respectively.
(*f* **atanh** *a*)
- (*f* **cis** *a*) ▷ Return $e^{i a} = \cos a + i \sin a$.
- (*f* **conjugate** *a*) ▷ Return complex conjugate of *a*.
- (*f* **max** *num*⁺)
(*f* **min** *num*⁺) ▷ Greatest or least, respectively, of *nums*.
- ($\left. \begin{array}{l} \{ \text{fround} | \text{fround} \} \\ \{ \text{ffloor} | \text{ffloor} \} \\ \{ \text{fceiling} | \text{fceiling} \} \\ \{ \text{ftruncate} | \text{ftruncate} \} \end{array} \right\} n [d_{\square}])$)
▷ Return as **integer** or **float**, respectively, n/d rounded, or rounded towards $-\infty$, $+\infty$, or 0, respectively; and remainder.
- ($\left. \begin{array}{l} \{ \text{fmod} \} \\ \{ \text{frem} \} \end{array} \right\} n d$)
▷ Same as *f* **floor** or *f* **truncate**, respectively, but return remainder only.
- (*f* **random** *limit* [state_{*v**random-state*}])
▷ Return non-negative random number less than *limit*, and of the same type.
- (*f* **make-random-state** [state | NIL | T]_{*n*})
▷ Copy of **random-state** object *state* or of the current random state; or a randomly initialized fresh random state.
- v****random-state*** ▷ Current random state.
- (*f* **float-sign** *num-a* [*num-b*]_{*n*}) ▷ num-b with *num-a*'s sign.
- (*f* **signum** *n*)
▷ Number of magnitude 1 representing sign or phase of *n*.
- (*f* **numerator** *rational*)
(*f* **denominator** *rational*)
▷ Numerator or denominator, respectively, of *rational*'s canonical form.
- (*f* **realpart** *number*)
(*f* **imagpart** *number*)
▷ Real part or imaginary part, respectively, of *number*.
- (*f* **complex** *real* [*imag*]_{*n*}) ▷ Make a complex number.
- (*f* **phase** *num*) ▷ Angle of *num*'s polar representation.
- (*f* **abs** *n*) ▷ Return |*n*|.
- (*f* **rational** *real*)
(*f* **rationalize** *real*)
▷ Convert *real* to rational. Assume complete/limited accuracy for *real*.
- (*f* **float** *real* [*prototype*]_{*n*})
▷ Convert *real* into float with type of *prototype*.

1.3 Logic Functions

Negative integers are used in two's complement representation.

- (*f* **boole** *operation* *int-a* *int-b*)
▷ Return value of bitwise logical *operation*. *operations* are
- c* **boole-1** ▷ *int-a*.
c **boole-2** ▷ *int-b*.
c **boole-c1** ▷ \neg *int-a*.
c **boole-c2** ▷ \neg *int-b*.
c **boole-set** ▷ All bits set.
c **boole-clr** ▷ All bits zero.

- cboole-eqv** \triangleright $int-a \equiv int-b$.
- cboole-and** \triangleright $int-a \wedge int-b$.
- cboole-andc1** \triangleright $\neg int-a \wedge int-b$.
- cboole-andc2** \triangleright $int-a \wedge \neg int-b$.
- cboole-nand** \triangleright $\neg(int-a \wedge int-b)$.
- cboole-ior** \triangleright $int-a \vee int-b$.
- cboole-orc1** \triangleright $\neg int-a \vee int-b$.
- cboole-orc2** \triangleright $int-a \vee \neg int-b$.
- cboole-xor** \triangleright $\neg(int-a \equiv int-b)$.
- cboole-nor** \triangleright $\neg(int-a \vee int-b)$.
- (flognot integer)** \triangleright $\neg integer$.
- (flogeqv integer*)**
(flogand integer*)
 \triangleright Return value of exclusive-nored or anded integers, respectively. Without any *integer*, return -1.
- (flogandc1 int-a int-b)** \triangleright $\neg int-a \wedge int-b$.
- (flogandc2 int-a int-b)** \triangleright $int-a \wedge \neg int-b$.
- (flognand int-a int-b)** \triangleright $\neg(int-a \wedge int-b)$.
- (flogxor integer*)**
(flogior integer*)
 \triangleright Return value of exclusive-ored or ored integers, respectively. Without any *integer*, return 0.
- (flogorc1 int-a int-b)** \triangleright $\neg int-a \vee int-b$.
- (flogorc2 int-a int-b)** \triangleright $int-a \vee \neg int-b$.
- (flognor int-a int-b)** \triangleright $\neg(int-a \vee int-b)$.
- (flogbitp i int)** \triangleright T if zero-indexed *i*th bit of *int* is set.
- (flogtest int-a int-b)**
 \triangleright Return T if there is any bit set in *int-a* which is set in *int-b* as well.
- (flogcount int)**
 \triangleright Number of 1 bits in $int \geq 0$, number of 0 bits in $int < 0$.

1.4 Integer Functions

- (finteger-length integer)**
 \triangleright Number of bits necessary to represent *integer*.
- (fldb-test byte-spec integer)**
 \triangleright Return T if any bit specified by *byte-spec* in *integer* is set.
- (fash integer count)**
 \triangleright Return copy of integer arithmetically shifted left by *count* adding zeros at the right, or, for *count* < 0, shifted right discarding bits.
- (fldb byte-spec integer)**
 \triangleright Extract byte denoted by *byte-spec* from *integer*. **setfable**.
- ($\left. \begin{array}{l} \text{fdeposit-field} \\ \text{fdpb} \end{array} \right\} int-a \text{ byte-spec } int-b)$**
 \triangleright Return int-b with bits denoted by *byte-spec* replaced by corresponding bits of *int-a*, or by the low (**fbyte-size** *byte-spec*) bits of *int-a*, respectively.
- (fmask-field byte-spec integer)**
 \triangleright Return copy of integer with all bits unset but those denoted by *byte-spec*. **setfable**.
- (fbyte size position)**
 \triangleright Byte specifier for a byte of *size* bits starting at a weight of $2^{position}$.
- (fbyte-size byte-spec)**
(fbyte-position byte-spec)
 \triangleright Size or position, respectively, of *byte-spec*.

1.5 Implementation-Dependent

$\left. \begin{array}{l} \text{cshort-float} \\ \text{csingle-float} \\ \text{cdouble-float} \\ \text{clong-float} \end{array} \right\} \begin{cases} \text{epsilon} \\ \text{negative-epsilon} \end{cases}$

▷ Smallest possible number making a difference when added or subtracted, respectively.

$\left. \begin{array}{l} \text{cleast-negative} \\ \text{cleast-negative-normalized} \\ \text{cleast-positive} \\ \text{cleast-positive-normalized} \end{array} \right\} \begin{cases} \text{short-float} \\ \text{single-float} \\ \text{double-float} \\ \text{long-float} \end{cases}$

▷ Available numbers closest to -0 or $+0$, respectively.

$\left. \begin{array}{l} \text{cmost-negative} \\ \text{cmost-positive} \end{array} \right\} \begin{cases} \text{short-float} \\ \text{single-float} \\ \text{double-float} \\ \text{long-float} \\ \text{fixnum} \end{cases}$

▷ Available numbers closest to $-\infty$ or $+\infty$, respectively.

(*f* **decode-float** *n*)

(*f* **integer-decode-float** *n*)

▷ Return significand, exponent, and sign of **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*_{NIL}])

▷ Type of most specialized **complex** number able to hold parts of type *foo*.

2 Characters

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

(*f* **characterp** *foo*)

(*f* **standard-char-p** *char*)

▷ T if argument is of indicated type.

(*f* **graphic-char-p** *character*)

(*f* **alpha-char-p** *character*)

(*f* **alphanumericp** *character*)

▷ T if *character* is visible, alphabetic, or alphanumeric, respectively.

(*f* **upper-case-p** *character*)

(*f* **lower-case-p** *character*)

(*f* **both-case-p** *character*)

▷ Return T if *character* is uppercase, lowercase, or able to be in another case, respectively.

(*f* **digit-char-p** *character* [*radix*₁₀])

▷ Return its weight if *character* is a digit, or NIL otherwise.

(*f* **char=** *character*⁺)

(*f* **char/=** *character*⁺)

▷ Return T if all *characters*, or none, respectively, are equal.

(*f* **char-equal** *character*⁺)

(*f* **char-not-equal** *character*⁺)

▷ Return T if all *characters*, or none, respectively, are equal ignoring case.

(*f* **char>** *character*⁺)

(*f* **char>=** *character*⁺)

(*f* **char<** *character*⁺)

(*f* **char<=** *character*⁺)

▷ Return T if *characters* are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

- (*f*char-greaterp *character*⁺)
 (*f*char-not-lessp *character*⁺)
 (*f*char-lessp *character*⁺)
 (*f*char-not-greaterp *character*⁺)
 ▷ Return T if *characters* are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.
- (*f*char-upcase *character*)
 (*f*char-downcase *character*)
 ▷ Return corresponding uppercase/lowercase character, respectively.
- (*f*digit-char *i* [*radix*₁₀]) ▷ Character representing digit *i*.
- (*f*char-name *char*) ▷ *char*'s name if any, or NIL.
- (*f*name-char *foo*) ▷ Character named *foo* if any, or NIL.
- (*f*char-int *character*)
 (*f*char-code *character*) ▷ Code of *character*.
- (*f*code-char *code*) ▷ Character with *code*.
- cchar-code-limit ▷ Upper bound of (*f*char-code *char*); ≥ 96.
- (*f*character *c*) ▷ Return #\c.

3 Strings

Strings can as well be manipulated by array and sequence functions; see pages 10 and 12.

- (*f*stringp *foo*)
 (*f*simple-string-p *foo*) ▷ T if *foo* is of indicated type.
- (*f*string=) *foo bar* $\left\{ \begin{array}{l} \text{:start1 } \text{start-foo}_{\boxed{0}} \\ \text{:start2 } \text{start-bar}_{\boxed{0}} \\ \text{:end1 } \text{end-foo}_{\boxed{\text{NIL}}} \\ \text{:end2 } \text{end-bar}_{\boxed{\text{NIL}}} \end{array} \right\}$
 ▷ Return T if subsequences of *foo* and *bar* are equal. Obey/ignore, respectively, case.
- (*f*string{/= | -not-equal }
*f*string{> | -greaterp }
*f*string{>= | -not-lessp }
*f*string{< | -lessp }
*f*string{<= | -not-greaterp }
) *foo bar* $\left\{ \begin{array}{l} \text{:start1 } \text{start-foo}_{\boxed{0}} \\ \text{:start2 } \text{start-bar}_{\boxed{0}} \\ \text{:end1 } \text{end-foo}_{\boxed{\text{NIL}}} \\ \text{:end2 } \text{end-bar}_{\boxed{\text{NIL}}} \end{array} \right\}$
 ▷ If *foo* is lexicographically not equal, greater, not less, less, or not greater, respectively, then return position of first mismatching character in *foo*. Otherwise return NIL. Obey/ignore, respectively, case.
- (*f*make-string *size* $\left\{ \begin{array}{l} \text{:initial-element } \text{char} \\ \text{:element-type } \text{type}_{\boxed{\text{character}}} \end{array} \right\}$)
 ▷ Return string of length *size*.

- (*f*string *x*)
 (*f*string-capitalize)
 (*f*string-upcase)
 (*f*string-downcase)
) *x* $\left\{ \begin{array}{l} \text{:start } \text{start}_{\boxed{0}} \\ \text{:end } \text{end}_{\boxed{\text{NIL}}} \end{array} \right\}$
 ▷ Convert *x* (**symbol**, **string**, or **character**) into a string, a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

- (*f*nstring-capitalize)
 (*f*nstring-upcase)
 (*f*nstring-downcase)
) *string* $\left\{ \begin{array}{l} \text{:start } \text{start}_{\boxed{0}} \\ \text{:end } \text{end}_{\boxed{\text{NIL}}} \end{array} \right\}$
 ▷ Convert *string* into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

- (*f*string-trim)
 (*f*string-left-trim)
 (*f*string-right-trim)
) *char-bag string*
 ▷ Return string with all characters in sequence *char-bag* removed from both ends, from the beginning, or from the end, respectively.

$(\left\{ \begin{array}{l} \text{fbutlast } list \\ \text{fnbutlast } list \end{array} \right\} [num] \text{])}$ ▷ list excluding last *num* conses.

$(\left\{ \begin{array}{l} \text{frplaca} \\ \text{frplacd} \end{array} \right\} \widetilde{cons} \text{ object})$
 ▷ Replace car, or cdr, respectively, of cons with *object*.

$(\text{fldiff } list \text{ foo})$
 ▷ If *foo* is a tail of *list*, return preceding part of list. Otherwise return list.

$(\text{fadjoin } foo \text{ list } \left\{ \begin{array}{l} \text{:test } function \text{ [#'eq]} \\ \text{:test-not } function \\ \text{:key } function \end{array} \right\})$
 ▷ Return list if *foo* is already member of *list*. If not, return (fcons foo list).

$(\text{mpop } \widetilde{place})$
 ▷ Set *place* to (fcdr *place*), return (fcar *place*).

$(\text{mpush } foo \text{ } \widetilde{place})$ ▷ Set *place* to (fcons foo *place*).

$(\text{mpushnew } foo \text{ } \widetilde{place} \left\{ \begin{array}{l} \text{:test } function \text{ [#'eq]} \\ \text{:test-not } function \\ \text{:key } function \end{array} \right\})$
 ▷ Set *place* to (fadjoin foo *place*).

$(\text{fappend } [proper-list^* \text{ foo}] \text{])}$
 $(\text{fnconc } [non-circular-list^* \text{ foo}] \text{])}$
 ▷ Return concatenated list or, with only one argument, foo. *foo* can be of any type.

$(\text{frevappend } list \text{ foo})$
 $(\text{fnreconc } \widetilde{list} \text{ foo})$
 ▷ Return concatenated list after reversing order in *list*.

$(\left\{ \begin{array}{l} \text{fmapcar} \\ \text{fmaplist} \end{array} \right\} function \text{ list}^+)$
 ▷ Return list of return values of *function* successively invoked with corresponding arguments, either cars or cdrs, respectively, from each *list*.

$(\left\{ \begin{array}{l} \text{fmapcan} \\ \text{fmapcon} \end{array} \right\} function \text{ } \widetilde{list}^+)$
 ▷ Return list of concatenated return values of *function* successively invoked with corresponding arguments, either cars or cdrs, respectively, from each *list*. *function* should return a list.

$(\left\{ \begin{array}{l} \text{fmapc} \\ \text{fmapl} \end{array} \right\} function \text{ list}^+)$
 ▷ Return first list after successively applying *function* to corresponding arguments, either cars or cdrs, respectively, from each *list*. *function* should have some side effects.

$(\text{fcopy-list } list)$ ▷ Return copy of *list* with shared elements.

4.3 Association Lists

$(\text{fpairlis } keys \text{ values } [alist] \text{])}$
 ▷ Prepend to alist an association list made from lists *keys* and *values*.

$(\text{facons } key \text{ value } alist)$
 ▷ Return alist with a (*key* . *value*) pair added.

$(\left\{ \begin{array}{l} \text{fassoc} \\ \text{fassoc-if}[-not] \end{array} \right\} foo \text{ alist } \left\{ \begin{array}{l} \text{:test } test \text{ [#'eq]} \\ \text{:test-not } test \\ \text{:key } function \end{array} \right\})$
 $(\left\{ \begin{array}{l} \text{fassoc-if}[-not] \\ \text{fassoc-if}[-not] \end{array} \right\} test \text{ alist } [:key \text{ function}])$
 ▷ First cons whose car, or cdr, respectively, satisfies *test*.

$(\text{fcopy-alist } alist)$ ▷ Return copy of *alist*.

4.4 Trees

(*f*tree-equal *foo bar* {*:test* *test*_{#'eq1}
{*:test-not* *test*})

▷ Return T if trees *foo* and *bar* have same shape and leaves satisfying *test*.

{*f*subst *new old tree* } {*:test* *function*_{#'eq1}
{*f*nsubst *new old tree* } {*:test-not* *function*
{*:key* *function* }

▷ Make copy of tree with each subtree or leaf matching *old* replaced by *new*.

{*f*subst-if[-not] *new test tree* } [*:key* *function*]

▷ Make copy of tree with each subtree or leaf satisfying *test* replaced by *new*.

{*f*sublis *association-list tree* } {*:test* *function*_{#'eq1}
{*f*nsublis *association-list tree* } {*:test-not* *function*
{*:key* *function* }

▷ Make copy of tree with each subtree or leaf matching a key in *association-list* replaced by that key's value.

(*f*copy-tree *tree*)

▷ Copy of tree with same shape and leaves.

4.5 Sets

{*f*intersection } *a b* } {*:test* *function*_{#'eq1}
{*f*set-difference } *a b* } {*:test-not* *function*
{*f*union } *a b* } {*:key* *function*
{*f*set-exclusive-or } *a b* }
{*f*nintersection } *~a b* }
{*f*nset-difference } *~a b* }
{*f*nunion } *~a ~b* }
{*f*nset-exclusive-or } *~a ~b* }

▷ Return $\underline{a \cap b}$, $\underline{a \setminus b}$, $\underline{a \cup b}$, or $\underline{a \triangle b}$, respectively, of lists *a* and *b*.

5 Arrays

5.1 Predicates

(*f*arrayp *foo*)

(*f*vectorp *foo*)

(*f*simple-vector-p *foo*)

▷ T if *foo* is of indicated type.

(*f*bit-vector-p *foo*)

(*f*simple-bit-vector-p *foo*)

(*f*adjustable-array-p *array*)

(*f*array-has-fill-pointer-p *array*)

▷ T if *array* is adjustable/has a fill pointer, respectively.

(*f*array-in-bounds-p *array* [*subscripts*])

▷ Return T if *subscripts* are in *array*'s bounds.

5.2 Array Functions

{*f*make-array *dimension-sizes* [*:adjustable* *bool*_{NIL}]

{*f*adjust-array *array* *dimension-sizes*

{*:element-type* *type*_T
{*:fill-pointer* {*num*|*bool*}_{NIL}
{*:initial-element* *obj*
{*:initial-contents* *tree-or-array*
{*:displaced-to* *array*_{NIL} [*:displaced-index-offset* *i*₀]

▷ Return fresh, or readjust, respectively, vector or array.

(*f*aref *array* [*subscripts*])

▷ Return array element pointed to by *subscripts*. **setfable**.

(*f*row-major-aref *array* *i*)

▷ Return *i*th element of *array* in row-major order. **setfable**.

(*f***array-row-major-index** *array* [*subscripts*])
 ▷ Index in row-major order of the element denoted by *subscripts*.

(*f***array-dimensions** *array*)
 ▷ List containing the lengths of *array*'s dimensions.

(*f***array-dimension** *array* *i*)
 ▷ Length of *i*th dimension of *array*.

(*f***array-total-size** *array*) ▷ Number of elements in *array*.

(*f***array-rank** *array*) ▷ Number of dimensions of *array*.

(*f***array-displacement** *array*) ▷ Target array and offset.

(*f***bit** *bit-array* [*subscripts*])

(*f***sbit** *simple-bit-array* [*subscripts*])

▷ Return element of *bit-array* or of *simple-bit-array*. **setf**able.

(*f***bit-not** *bit-array* [*result-bit-array*_{NIL}])

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

(*f***bit-eqv**
*f***bit-and**
*f***bit-andc1**
*f***bit-andc2**
*f***bit-nand**
*f***bit-ior**
*f***bit-orc1**
*f***bit-orc2**
*f***bit-xor**
*f***bit-nor**) *bit-array-a* *bit-array-b* [*result-bit-array*_{NIL}])

▷ Return result of bitwise logical operations (cf. operations of *f***boole**, page 4) on *bit-array-a* and *bit-array-b*. If *result-bit-array* is T, put result in *bit-array-a*; if it is NIL, make a new array for result.

*c***array-rank-limit** ▷ Upper bound of array rank; ≥ 8 .

*c***array-dimension-limit**

▷ Upper bound of an array dimension; ≥ 1024 .

*c***array-total-size-limit**

▷ Upper bound of array size; ≥ 1024 .

5.3 Vector Functions

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

(*f***vector** *foo**) ▷ Return fresh simple vector of *foos*.

(*f***svref** *vector* *i*) ▷ Element *i* of simple *vector*. **setf**able.

(*f***vector-push** *foo* *vector*)

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

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

▷ Replace element of *vector* pointed to by fill pointer with *foo*, then increment fill pointer. Extend *vector*'s size by \geq *num* if necessary.

(*f***vector-pop** *vector*)

▷ Return element of *vector* its fillpointer points to after decrementation.

(*f***fill-pointer** *vector*) ▷ Fill pointer of *vector*. **setf**able.

6 Sequences

6.1 Sequence Predicates

$(\left\{ \begin{array}{l} \text{every} \\ \text{notevery} \end{array} \right\} \text{ test sequence}^+)$

▷ Return NIL or T, respectively, as soon as *test* on any set of corresponding elements of *sequences* returns NIL.

$(\left\{ \begin{array}{l} \text{some} \\ \text{notany} \end{array} \right\} \text{ test sequence}^+)$

▷ Return value of test or NIL, respectively, as soon as *test* on any set of corresponding elements of *sequences* returns non-NIL.

$(\text{f mismatch } \text{sequence-a } \text{sequence-b } \left\{ \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \left\{ \begin{array}{l} \text{:test } \text{function}_{\neq \text{eq}} \\ \text{:test-not } \text{function} \end{array} \right\} \\ \text{:start1 } \text{start-a}_{\text{0}} \\ \text{:start2 } \text{start-b}_{\text{0}} \\ \text{:end1 } \text{end-a}_{\text{NIL}} \\ \text{:end2 } \text{end-b}_{\text{NIL}} \\ \text{:key } \text{function} \end{array} \right\})$

▷ Return position in sequence-a where *sequence-a* and *sequence-b* begin to mismatch. Return NIL if they match entirely.

6.2 Sequence Functions

$(\text{f make-sequence } \text{sequence-type } \text{size } [\text{:initial-element } \text{foo}])$

▷ Make sequence of *sequence-type* with *size* elements.

$(\text{f concatenate } \text{type } \text{sequence}^*)$

▷ Return concatenated sequence of *type*.

$(\text{f merge } \text{type } \widetilde{\text{sequence-a}} \widetilde{\text{sequence-b}} \text{ test } [\text{:key function}_{\text{NIL}}])$

▷ Return interleaved sequence of *type*. Merged sequence will be sorted if both *sequence-a* and *sequence-b* are sorted.

$(\text{f fill } \widetilde{\text{sequence}} \text{ foo } \left\{ \begin{array}{l} \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \end{array} \right\})$

▷ Return sequence after setting elements between *start* and *end* to *foo*.

$(\text{f length } \text{sequence})$

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

$(\text{f count } \text{foo } \text{sequence } \left\{ \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \left\{ \begin{array}{l} \text{:test } \text{function}_{\neq \text{eq}} \\ \text{:test-not } \text{function} \end{array} \right\} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \end{array} \right\})$

▷ Return number of elements in *sequence* which match *foo*.

$(\left\{ \begin{array}{l} \text{f count-if} \\ \text{f count-if-not} \end{array} \right\} \text{ test } \text{sequence } \left\{ \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \end{array} \right\})$

▷ Return number of elements in *sequence* which satisfy *test*.

$(\text{f elt } \text{sequence } \text{index})$

▷ Return element of sequence pointed to by zero-indexed *index*. **setfable**.

$(\text{f subseq } \text{sequence } \text{start } [\text{end}_{\text{NIL}}])$

▷ Return subsequence of sequence between *start* and *end*. **setfable**.

$(\left\{ \begin{array}{l} \text{f sort} \\ \text{f stable-sort} \end{array} \right\} \widetilde{\text{sequence}} \text{ test } [\text{:key function}])$

▷ Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.

$(\text{f reverse } \text{sequence})$

$(\text{f nreverse } \text{sequence})$

▷ Return sequence in reverse order.

$$\left. \begin{array}{l} \{f \text{find} \\ f \text{position} \end{array} \right\} \text{foo sequence} \left. \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \{ \text{:test } \text{function}_{\#'\text{eq}} \\ \text{:test-not } \text{test} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \end{array} \right\}$$

▷ Return first element in *sequence* which matches *foo*, or its position relative to the begin of *sequence*, respectively.

$$\left. \begin{array}{l} \{f \text{find-if} \\ f \text{find-if-not} \\ f \text{position-if} \\ f \text{position-if-not} \end{array} \right\} \text{test sequence} \left. \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \end{array} \right\}$$

▷ Return first element in *sequence* which satisfies *test*, or its position relative to the begin of *sequence*, respectively.

$$(f \text{search } \text{sequence-a } \text{sequence-b}) \left. \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \{ \text{:test } \text{function}_{\#'\text{eq}} \\ \text{:test-not } \text{function} \\ \text{:start1 } \text{start-a}_{\text{0}} \\ \text{:start2 } \text{start-b}_{\text{0}} \\ \text{:end1 } \text{end-a}_{\text{NIL}} \\ \text{:end2 } \text{end-b}_{\text{NIL}} \\ \text{:key } \text{function} \end{array} \right\}$$

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

$$\left. \begin{array}{l} \{f \text{remove } \text{foo } \text{sequence} \\ f \text{delete } \text{foo } \text{sequence} \end{array} \right\} \left. \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \{ \text{:test } \text{function}_{\#'\text{eq}} \\ \text{:test-not } \text{function} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \\ \text{:count } \text{count}_{\text{NIL}} \end{array} \right\}$$

▷ Make copy of sequence without elements matching *foo*.

$$\left. \begin{array}{l} \{f \text{remove-if} \\ f \text{remove-if-not} \\ f \text{delete-if} \\ f \text{delete-if-not} \end{array} \right\} \text{test sequence} \left. \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \\ \text{:count } \text{count}_{\text{NIL}} \end{array} \right\}$$

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

$$\left. \begin{array}{l} \{f \text{remove-duplicates } \text{sequence} \\ f \text{delete-duplicates } \text{sequence} \end{array} \right\} \left. \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \{ \text{:test } \text{function}_{\#'\text{eq}} \\ \text{:test-not } \text{function} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \end{array} \right\}$$

▷ Make copy of sequence without duplicates.

$$\left. \begin{array}{l} \{f \text{substitute } \text{new } \text{old } \text{sequence} \\ f \text{nsubstitute } \text{new } \text{old } \text{sequence} \end{array} \right\} \left. \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \{ \text{:test } \text{function}_{\#'\text{eq}} \\ \text{:test-not } \text{function} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \\ \text{:count } \text{count}_{\text{NIL}} \end{array} \right\}$$

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

$$\left. \begin{array}{l} \{f \text{substitute-if} \\ f \text{substitute-if-not} \\ f \text{nsubstitute-if} \\ f \text{nsubstitute-if-not} \end{array} \right\} \text{new test sequence} \left. \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \\ \text{:count } \text{count}_{\text{NIL}} \end{array} \right\}$$

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

(*f* **replace** *sequence-a* *sequence-b* $\left\{ \begin{array}{l} \text{:start1 } \textit{start-a}_{\mathbb{0}} \\ \text{:start2 } \textit{start-b}_{\mathbb{0}} \\ \text{:end1 } \textit{end-a}_{\mathbb{NIL}} \\ \text{:end2 } \textit{end-b}_{\mathbb{NIL}} \end{array} \right\}$)

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

(*f* **map** *type function sequence*⁺)

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

(*f* **map-into** *result-sequence function sequence*^{*})

▷ Store into *result-sequence* successively values of *function* applied to corresponding elements of the *sequences*.

(*f* **reduce** *function sequence* $\left\{ \begin{array}{l} \text{:initial-value } \textit{foo}_{\mathbb{NIL}} \\ \text{:from-end } \textit{bool}_{\mathbb{NIL}} \\ \text{:start } \textit{start}_{\mathbb{0}} \\ \text{:end } \textit{end}_{\mathbb{NIL}} \\ \text{:key } \textit{function} \end{array} \right\}$)

▷ 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.

(*f* **copy-seq** *sequence*)

▷ Copy of *sequence* with shared elements.

7 Hash Tables

The Loop Facility provides additional hash table-related functionality; see **loop**, page 21.

Key-value storage similar to hash tables can as well be achieved using association lists and property lists; see pages 9 and 16.

(*f* **hash-table-p** *foo*) ▷ Return T if *foo* is of type **hash-table**.

(*f* **make-hash-table** $\left\{ \begin{array}{l} \text{:test } \{ \textit{f} \textit{eq} | \textit{f} \textit{eql} | \textit{f} \textit{equal} | \textit{f} \textit{equalp} \}_{\#'\textit{eql}} \\ \text{:size } \textit{int} \\ \text{:rehash-size } \textit{num} \\ \text{:rehash-threshold } \textit{num} \end{array} \right\}$)

▷ Make a hash table.

(*f* **gethash** *key hash-table* [*default*₂₂])

▷ Return object with *key* if any or *default* otherwise; and T if found, NIL otherwise. **setfable**.

(*f* **hash-table-count** *hash-table*)

▷ Number of entries in *hash-table*.

(*f* **remhash** *key hash-table*)

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

(*f* **clrhash** *hash-table*) ▷ Empty hash-table.

(*f* **maphash** *function hash-table*)

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

(*m* **with-hash-table-iterator** (*foo hash-table*) (**declare** *decl*^{*})^{*} *form*^{P*})

▷ Return values of *forms*. In *forms*, invocations of (*foo*) return: T if an entry is returned; its key; its value.

(*f* **hash-table-test** *hash-table*)

▷ Test function used in *hash-table*.

(*f* **hash-table-size** *hash-table*)

(*f* **hash-table-rehash-size** *hash-table*)

(*f* **hash-table-rehash-threshold** *hash-table*)

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

(*f* **sxhash** *foo*)

▷ Hash code unique for any argument *f* **equal** *foo*.

8 Structures

(*m*defstruct

```

{foo
  {
    (:conc-name
     [slot-prefix foo-])
    (:constructor
     [maker MAKE-foo [(ord-λ*)]])
    (:copier
     [copier COPY-foo])
    (:include struct
     (slot [init
            {
              :type sl-type
              :read-only b
            }
          ]])
    (:type
     {
       list
       vector
       (vector type)
     }
     {
       :named
       (:initial-offset n)
     }
    )
    (:print-object [o-printer])
    (:print-function [f-printer])
    (:predicate
     [p-name foo-P])
  }
  (slot [init
         {
           :type slot-type
           :read-only bool
         }
        ]])
}
[doc]

```

▷ 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-λ* (see page 17) is given, by (*maker arg** {**:key value**}*). In the latter case, *args* and **:keys** correspond to the positional and keyword parameters defined in *ord-λ* whose *vars* in turn correspond to *slots*. **:print-object**/**:print-function** generate a **print-object** method for an instance *bar* of *foo* calling (*o-printer bar stream*) or (*f-printer bar stream print-level*), respectively. If **:type** without **:named** is given, no *foo-P* is created.

(*f*copy-structure *structure*)

▷ Return copy of *structure* with shared slot values.

9 Control Structure

9.1 Predicates

(*f*eq *foo bar*) ▷ T if *foo* and *bar* are identical.

(*f*eql *foo bar*)

▷ T if *foo* and *bar* are identical, or the same **character**, or **numbers** of the same type and value.

(*f*equal *foo bar*)

▷ T if *foo* and *bar* are *f*eql, or are equivalent **pathnames**, or are **conses** with *f*equal cars and cdrs, or are **strings** or **bit-vectors** with *f*eql elements below their fill pointers.

(*f*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 *f*equalp elements; or are structures of the same type with *f*equalp elements; or are **hash-tables** of the same size with the same **:test** function, the same keys in terms of **:test** function, and *f*equalp elements.

(*f*not *foo*)

▷ T if *foo* is NIL; NIL otherwise.

(*f*boundp *symbol*)

▷ T if *symbol* is a special variable.

(*f*constantp *foo* [*environment* NIL])

▷ T if *foo* is a constant form.

(*f*functionp *foo*)

▷ T if *foo* is of type **function**.

(*f*fboundp $\left\{ \begin{array}{l} \widehat{foo} \\ (\text{setf } \widehat{foo}) \end{array} \right\}$)

▷ T if *foo* is a global function or macro.

9.2 Variables

($\left\{ \begin{array}{l} \text{mdefconstant} \\ \text{mdefparameter} \end{array} \right\}$ \widehat{foo} *form* [*doc*])

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

(*m*defvar \widehat{foo} [*form* [*doc*]])

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

($\left\{ \begin{array}{l} \text{msetf} \\ \text{mpsetf} \end{array} \right\}$ {*place form*}*)

▷ Set *places* to primary values of *forms*. Return values of last *form*/NIL; work sequentially/in parallel, respectively.

($\left\{ \begin{array}{l} \text{ssetq} \\ \text{mpsetq} \end{array} \right\}$ {*symbol form*}*)

▷ Set *symbols* to primary values of *forms*. Return value of last *form*/NIL; work sequentially/in parallel, respectively.

(*f*set $\widetilde{\text{symbol } foo}$)

▷ Set *symbol*'s value cell to *foo*. Deprecated.

(*m*multiple-value-setq *vars form*)

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

(*m*shiftf $\widetilde{\text{place}^+}$ *foo*)

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

(*m*rotatef $\widetilde{\text{place}^*}$)

▷ Rotate values of *places* left, old first becoming new last *place*'s value. Return NIL.

(*f*makunbound \widetilde{foo})

▷ Delete special variable *foo* if any.

(*f*get *symbol key* [*default*_{NIL}])

(*f*getf *place key* [*default*_{NIL}])

▷ First entry *key* from property list stored in *symbol*/in *place*, respectively, or *default* if there is no *key*. **setfable**.

(*f*get-properties *property-list keys*)

▷ Return *key* and *value*₂ of first entry from *property-list* matching a key from *keys*, and tail of *property-list* starting with that key. Return NIL, NIL₂, and NIL₃ if there was no matching key in *property-list*.

(*f*remprop $\widetilde{\text{symbol } key}$)

(*m*remf $\widetilde{\text{place } key}$)

▷ Remove first entry *key* from property list stored in *symbol*/in *place*, respectively. Return T if *key* was there, or NIL otherwise.

(*s*progv *symbols values form*^{P_k})

▷ Evaluate *forms* with locally established dynamic bindings of *symbols* to *values* or NIL. Return values of *forms*.

($\left\{ \begin{array}{l} \text{slet} \\ \text{slet*} \end{array} \right\}$ ($\left\{ \begin{array}{l} \text{name} \\ (\text{name } [\text{value}_{\text{NIL}}]) \end{array} \right\}^*$) (**declare** $\widehat{\text{decl}^*}$)^{*} *form*^{P_k})

▷ Evaluate *forms* with *names* lexically bound (in parallel or sequentially, respectively) to *values*. Return values of *forms*.

(*m*multiple-value-bind ($\widehat{\text{var}^*}$) *values-form* (**declare** $\widehat{\text{decl}^*}$)^{*} *body-form*^{P_k})

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

(*mdestructuring-bind* *destruct-λ bar (declare decl*)* form^{P*}*)
 ▷ Evaluate *forms* with variables from tree *destruct-λ* bound to corresponding elements of tree *bar*, and return their values. *destruct-λ* resembles *macro-λ* (section 9.4), but without any **&environment** clause.

9.3 Functions

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

$$(var^* [\&optional \left\{ \begin{array}{l} var \\ (var [init_{NIL} [supplied-p]]) \end{array} \right\}^*] [\&rest var] \\ [\&key \left\{ \begin{array}{l} var \\ (\{var \\ (:key var)\} [init_{NIL} [supplied-p]]) \end{array} \right\}^* \\ [\&allow-other-keys] [\&aux \left\{ \begin{array}{l} var \\ (var [init_{NIL}]) \end{array} \right\}^*]]).$$

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

$$\left(\begin{array}{l} mdefun \\ mlambda \end{array} \left\{ \begin{array}{l} foo (ord-\lambda^*) \\ (setf foo) (new-value ord-\lambda^*) \end{array} \right\} (\text{declare } \widehat{decl}^*)^* [\widehat{doc}] \right. \\ \left. form^{P*} \right)$$

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

$$\left(\begin{array}{l} s\text{flet} \\ s\text{labels} \end{array} \right) \left(\left(\begin{array}{l} foo (ord-\lambda^*) \\ (setf foo) (new-value ord-\lambda^*) \end{array} \right) (\text{declare } \widehat{local-decl}^*)^* \right. \\ \left. [\widehat{doc}] local-form^{P*} \right) (\text{declare } \widehat{decl}^*)^* form^{P*}$$

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

$$(s\text{function} \left\{ \begin{array}{l} foo \\ (m\text{lambda } form^*) \end{array} \right\})$$

▷ Return lexically innermost function named *foo* or a lexical closure of the *mlambda* expression.

$$(f\text{apply} \left\{ \begin{array}{l} function \\ (setf function) \end{array} \right\} arg^* args)$$

▷ Values of function called with *args* and the list elements of *args*. **setfable** if *function* is one of *faref*, *fbit*, and *fbit*.

(*ffuncall function arg**)
 ▷ Values of function called with *args*.

(*smultiple-value-call function form**)
 ▷ Call *function* with all the values of each *form* as its arguments. Return values returned by function.

(*fvalues-list list*) ▷ Return elements of list.

(*fvalues foo**)
 ▷ Return as multiple values the primary values of the *foos*. **setfable**.

(*fmultiple-value-list form*) ▷ List of the values of form.

(*mnth-value n form*)
 ▷ Zero-indexed nth return value of *form*.

(*fcomplement function*)
 ▷ Return new function with same arguments and same side effects as *function*, but with complementary truth value.

(*fconstantly foo*)
 ▷ Function of any number of arguments returning *foo*.

(*fidentity foo*) ▷ Return foo.

(*f* **function-lambda-expression** *function*)

- ▷ If available, return lambda expression of *function*, NIL if *function* was defined in an environment without bindings, and name of *function*.

(*f* **fdefinition** $\left\{ \begin{array}{l} \textit{foo} \\ (\textit{setf} \textit{foo}) \end{array} \right\}$)

- ▷ Definition of global function *foo*. **setfable**.

(*f* **fmakunbound** *foo*)

- ▷ Remove global function or macro definition *foo*.

c **call-arguments-limit**

c **lambda-parameters-limit**

- ▷ Upper bound of the number of function arguments or lambda list parameters, respectively; ≥ 50 .

c **multiple-values-limit**

- ▷ Upper bound of the number of values a multiple value can have; ≥ 20 .

9.4 Macros

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

([**&whole** *var*] [*E*] $\left\{ \begin{array}{l} \textit{var} \\ (\textit{macro-}\lambda^*) \end{array} \right\}^*$ [*E*])

[**&optional** $\left\{ \begin{array}{l} \textit{var} \\ \left(\left\{ \begin{array}{l} \textit{var} \\ (\textit{macro-}\lambda^*) \end{array} \right\} [\textit{init}_{\text{NIL}} [\textit{supplied-p}]] \right) \end{array} \right\}^*$] [*E*]

[**&rest** $\left\{ \begin{array}{l} \textit{rest-var} \\ (\textit{macro-}\lambda^*) \end{array} \right\}$] [*E*]

[**&key** $\left\{ \begin{array}{l} \textit{var} \\ \left(\left\{ \begin{array}{l} \textit{var} \\ (:key \left\{ \begin{array}{l} \textit{var} \\ (\textit{macro-}\lambda^*) \end{array} \right\}) \end{array} \right\} [\textit{init}_{\text{NIL}} [\textit{supplied-p}]] \right) \end{array} \right\}^*$] [*E*]

[**&allow-other-keys**] [**&aux** $\left\{ \begin{array}{l} \textit{var} \\ (\textit{var} [\textit{init}_{\text{NIL}}]) \end{array} \right\}^*$] [*E*])

or

([**&whole** *var*] [*E*] $\left\{ \begin{array}{l} \textit{var} \\ (\textit{macro-}\lambda^*) \end{array} \right\}^*$ [*E*] [**&optional** $\left\{ \begin{array}{l} \textit{var} \\ \left(\left\{ \begin{array}{l} \textit{var} \\ (\textit{macro-}\lambda^*) \end{array} \right\} [\textit{init}_{\text{NIL}} [\textit{supplied-p}]] \right) \end{array} \right\}^*$] [*E*] . *rest-var*).

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.

($\left\{ \begin{array}{l} \textit{m} \textit{defmacro} \\ \textit{f} \textit{define-compiler-macro} \end{array} \right\}$ $\left\{ \begin{array}{l} \textit{foo} \\ (\textit{setf} \textit{foo}) \end{array} \right\}$ (*macro-λ**) (**declare** $\widehat{\textit{decl}^*}$)^{*} [*doc*] $\textit{form}^{\text{P}^*}$)

- ▷ Define macro *foo* which on evaluation as (*foo tree*) applies expanded *forms* to arguments from *tree*, which corresponds to *tree-shaped macro-λs*. *forms* are enclosed in an implicit **sblock** named *foo*.

(*m* **define-symbol-macro** *foo form*)

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

(*s* **macrolet** ((*foo* (*macro-λ**) (**declare** $\widehat{\textit{local-decl}^*}$)^{*} [*doc*] $\textit{macro-form}^{\text{P}^*}$)^{*}) (**declare** $\widehat{\textit{decl}^*}$)^{*} $\textit{form}^{\text{P}^*}$)

- ▷ Evaluate *forms* with locally defined mutually invisible macros *foo* which are enclosed in implicit **sblocks** of the same name.

(*s* **symbol-macrolet** ((*foo expansion-form*)^{*}) (**declare** $\widehat{\textit{decl}^*}$)^{*} $\textit{form}^{\text{P}^*}$)

- ▷ Evaluate *forms* with locally defined symbol macros *foo*.

(*m* **defsetf** *function*

$\left\{ \begin{array}{l} \widehat{\textit{updater}} [\widehat{\textit{doc}}] \\ \left((\textit{setf-}\lambda^*) (\textit{s-var}^*) (\textit{declare} \widehat{\textit{decl}^*})^* [\widehat{\textit{doc}}] \textit{form}^{\text{P}^*} \right) \end{array} \right\}$)

where defsetf lambda list (*setf-λ**) has the form (*var*^{*}

[&optional {var
 ((var [init_{NIL}] [supplied-p]))}]*] [&rest var]
 [&key {var
 ((:key var)) [init_{NIL}] [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 *setf-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 *s***block** named *function*.

(*m***define-setf-expander** *function* (*macro-λ**) (**declare** *decl**)* [*doc*]
*form^P**)

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

(*f***get-setf-expansion** *place* [*environment_{NIL}*])

▷ 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 **setf** and how to read *place*.

(*m***define-modify-macro** *foo* ([&optional
 {var
 ((var [init_{NIL}] [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.

*c***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.

&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 *s***let***.

9.5 Control Flow

(*s***if** *test* *then* [*else_{NIL}*])

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

(*m***cond** (*test then^P* [*test_{NIL}*])*)

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

{*m***when**
*m***unless**} *test* *foo^{P*}*)

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

(*m*case *test* ($\left\{ \begin{array}{c} \widehat{key}^* \\ key \end{array} \right\} foo^P$)^{*} [($\left\{ \begin{array}{c} \text{otherwise} \\ T \end{array} \right\} bar^P$)_{NIL}])

▷ Return the values of the first *foo*^{*} one of whose *keys* is **eq** *test*. Return values of *bars* if there is no matching *key*.

($\left\{ \begin{array}{c} m\text{ecase} \\ m\text{ccase} \end{array} \right\}$ *test* ($\left\{ \begin{array}{c} \widehat{key}^* \\ key \end{array} \right\} foo^P$)^{*})

▷ Return the values of the first *foo*^{*} one of whose *keys* is **eq** *test*. Signal non-correctable/correctable **type-error** if there is no matching *key*.

(*m*and *form*^{*}_T)

▷ 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*^{*}_{NIL})

▷ 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.

(*s*progn *form*^{*}_{NIL})

▷ Evaluate *forms* sequentially. Return values of last *form*.

(*s*multiple-value-prog1 *form-r form*^{*})

(*m*prog1 *form-r form*^{*})

(*m*prog2 *form-a form-r form*^{*})

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

($\left\{ \begin{array}{c} m\text{prog} \\ m\text{prog}^* \end{array} \right\}$ ($\left\{ \begin{array}{c} name \\ (name [value_{NIL}]) \end{array} \right\}$)^{*} (**declare** \widehat{decl} ^{*})^{*} $\left\{ \begin{array}{c} \widehat{tag} \\ form \end{array} \right\}$ ^{*})

▷ Evaluate **stagbody**-like body with *names* lexically bound (in parallel or sequentially, respectively) to *values*. Return NIL or explicitly *m*returned values. Implicitly, the whole form is a **sblock** named NIL.

(*s*unwind-protect *protected cleanup*^{*})

▷ Evaluate *protected* and then, no matter how control leaves *protected*, *cleanups*. Return values of *protected*.

(*s*block *name form*^{P*})

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

(*s*return-from *foo* [*result*_{NIL}])

(*m*return [*result*_{NIL}])

▷ Have nearest enclosing **sblock** named *foo*/named NIL, respectively, return with values of *result*.

(*s*tagbody $\{\widehat{tag} | form\}$ ^{*})

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

(*s*go \widehat{tag})

▷ Within the innermost possible enclosing **stagbody**, jump to a tag *f***eq** *tag*.

(*s*catch *tag form*^{P*})

▷ Evaluate *forms* and return their values unless interrupted by **sthrow**.

(*s*throw *tag form*)

▷ Have the nearest dynamically enclosing **scatch** with a tag *f***eq** *tag* return with the values of *form*.

(*f*sleep *n*)

▷ Wait *n* seconds; return NIL.

9.6 Iteration

$(\widehat{m\text{do}} \left\{ \begin{array}{l} \{var \\ (var [start [step]]) \}^* \\ \widehat{tag} \\ \widehat{form} \end{array} \right\} (stop \text{result}^P) (\text{declare } \widehat{decl}^*)^*$

▷ Evaluate $\text{s}\text{tagbody}$ -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*^P. Implicitly, the whole form is a sblock named NIL.

$(m\text{dotimes } (var\ i\ [result_{\text{NIL}}]) (\text{declare } \widehat{decl}^*)^* \{\widehat{tag}|\widehat{form}\}^*)$

▷ Evaluate $\text{s}\text{tagbody}$ -like body with *var* successively bound to integers from 0 to *i* - 1. Upon evaluation of *result*, *var* is *i*. Implicitly, the whole form is a sblock named NIL.

$(m\text{dolist } (var\ list\ [result_{\text{NIL}}]) (\text{declare } \widehat{decl}^*)^* \{\widehat{tag}|\widehat{form}\}^*)$

▷ Evaluate $\text{s}\text{tagbody}$ -like body with *var* successively bound to the elements of *list*. Upon evaluation of *result*, *var* is NIL. Implicitly, the whole form is a sblock named NIL.

9.7 Loop Facility

$(m\text{loop } form^*)$

▷ **Simple Loop.** If *forms* do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit sblock named NIL.

$(m\text{loop } clause^*)$

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

named n_{NIL} ▷ Give $m\text{loop}$'s implicit sblock a name.

with $\left\{ \begin{array}{l} \{var-s \\ (var-s^*) \} \\ [d-type] [= foo] \end{array} \right\}^+$

and $\left\{ \begin{array}{l} \{var-p \\ (var-p^*) \} \\ [d-type] [= bar] \end{array} \right\}^*$

where destructuring type specifier *d-type* has the form

$\left\{ \begin{array}{l} \text{fixnum|float|T|NIL|} \\ \{\text{of-type } \left\{ \begin{array}{l} type \\ (type^*) \end{array} \right\} \} \end{array} \right\}$

▷ Initialize (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel.

$\left\{ \left\{ \begin{array}{l} \text{for|as} \\ \left\{ \begin{array}{l} \{var-s \\ (var-s^*) \} \\ [d-type] \end{array} \right\}^+ \end{array} \right\} \left\{ \begin{array}{l} \text{and} \\ \left\{ \begin{array}{l} \{var-p \\ (var-p^*) \} \\ [d-type] \end{array} \right\}^* \end{array} \right\} \right\}$

▷ 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**.

$\{\text{upfrom|from|downfrom}\} start$

▷ Start stepping with *start*

$\{\text{upto|downto|to|below|above}\} form$

▷ Specify *form* as the end value for stepping.

$\{\text{in|on}\} list$

▷ Bind *var* to successive elements/tails, respectively, of *list*.

by $\{step_{\square} | function_{\#'\text{cdr}}\}$

▷ Specify the (positive) decrement or increment or the *function* of one argument returning the next part of the list.

$= foo [\text{then } bar_{\square}]$

▷ Bind *var* initially to *foo* and later to *bar*.

across *vector*

▷ Bind *var* to successive elements of *vector*.

being $\{\text{the|each}\}$

▷ Iterate over a hash table or a package.

$\{\text{hash-key|hash-keys}\} \{\text{of|in}\} hash-table [\text{using } (\text{hash-value } value)]$

▷ 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*.

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

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

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

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

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

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

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

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

repeat *num*
 ▷ Terminate *m***loop** after *num* iterations; *num* is evaluated once.

{while|until} *test*
 ▷ Continue iteration until *test* returns NIL or T, respectively.

{always|never} *test*
 ▷ Terminate *m***loop** returning NIL and skipping any **finally** parts as soon as *test* is NIL or T, respectively. Otherwise continue *m***loop** with its default return value set to T.

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

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

(**gupdate-instance-for-redefined-class** *new-instance added-slots discarded-slots discarded-slots-property-list* *{:initarg value}* other-keyarg**)
 ▷ On behalf of **gmake-instances-obsolete** and by means of **gshared-initialize**, set any *initarg* slots to their corresponding *values*; set any remaining *added-slots* to the values of their **:initform** forms. Not to be called by user.

(**gallocate-instance** *class* *{:initarg value}* other-keyarg**)
 ▷ Return uninitialized instance of *class*. Called by **gmake-instance**.

(**gshared-initialize** *instance* $\left\{ \begin{array}{l} \text{initform-slots} \\ \text{T} \end{array} \right\}$ *{:initarg-slot value}* other-keyarg**)
 ▷ Fill the *initarg-slots* of *instance* with the corresponding *values*, and fill those *initform-slots* that are not *initarg-slots* with the values of their **:initform** forms.

(**gslot-missing** *class instance slot* $\left\{ \begin{array}{l} \text{setf} \\ \text{slot-boundp} \\ \text{slot-makunbound} \\ \text{slot-value} \end{array} \right\}$ [*value*])

(**gslot-unbound** *class instance slot*)
 ▷ Called on attempted access to non-existing or unbound *slot*. Default methods signal **error/unbound-slot**, respectively. Not to be called by user.

10.2 Generic Functions

(**fnext-method-p**)
 ▷ T if enclosing method has a next method.

(**mdefgeneric** $\left\{ \begin{array}{l} \text{foo} \\ \text{(setf foo)} \end{array} \right\}$ (*required-var** [**&optional** $\left\{ \begin{array}{l} \text{var} \\ \text{(var)} \end{array} \right\}^*$] [**&rest** *var*] [**&key** $\left\{ \begin{array}{l} \text{var} \\ \text{(var | (:key var))} \end{array} \right\}^*$] [**&allow-other-keys**]))
 $\left. \begin{array}{l} \text{:argument-precedence-order } \text{required-var}^+ \\ \text{:declare (optimize method-selection-optimization)}^+ \\ \text{:documentation } \text{string} \\ \text{:generic-function-class } \text{gf-class} \text{standard-generic-function} \\ \text{:method-class } \text{method-class} \text{standard-method} \\ \text{:method-combination } \text{c-type} \text{standard } \text{c-arg}^* \\ \text{:method } \text{defmethod-args}^* \end{array} \right\}$)
 ▷ Define or modify generic function *foo*. Remove any methods previously defined by **defgeneric**. *gf-class* and the lambda paramters *required-var** and *var** must be compatible with existing methods. *defmethod-args* resemble those of **mdefmethod**. For *c-type* see section 10.3.

(**fensure-generic-function** $\left\{ \begin{array}{l} \text{foo} \\ \text{(setf foo)} \end{array} \right\}$
 $\left. \begin{array}{l} \text{:argument-precedence-order } \text{required-var}^+ \\ \text{:declare (optimize method-selection-optimization)} \\ \text{:documentation } \text{string} \\ \text{:generic-function-class } \text{gf-class} \\ \text{:method-class } \text{method-class} \\ \text{:method-combination } \text{c-type } \text{c-arg}^* \\ \text{:lambda-list } \text{lambda-list} \\ \text{:environment } \text{environment} \end{array} \right\}$)
 ▷ Define or modify generic function *foo*. *gf-class* and *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 *c-type* see section 10.3.

(**mdefmethod** $\left\{ \begin{array}{l} \text{foo} \\ \text{(setf foo)} \end{array} \right\}$ $\left[\begin{array}{l} \text{:before} \\ \text{:after} \\ \text{:around} \\ \text{qualifier}^* \end{array} \right\}$ $\left. \begin{array}{l} \text{primary method} \\ \text{primary method} \end{array} \right]$
 $\left(\begin{array}{l} \text{var} \\ \text{(spec-var } \left\{ \begin{array}{l} \text{class} \\ \text{(eql bar)} \end{array} \right\})^* \end{array} \right) \left[\text{\&optional} \right]$

$$\left\{ \begin{array}{l} \text{var} \\ (\text{var } [\text{init } [\text{supplied-p}]]) \end{array} \right\}^*] [\&\text{rest } \text{var}] [\&\text{key} \\ \left\{ \begin{array}{l} \text{var} \\ (\text{:key } \text{var}) \end{array} \right\} [\text{init } [\text{supplied-p}]] \right\}^* [\&\text{allow-other-keys}] \\ [\&\text{aux } \left\{ \begin{array}{l} \text{var} \\ (\text{var } [\text{init}]) \end{array} \right\}^*] \left\{ \begin{array}{l} (\text{declare } \widehat{\text{decl}^*})^* \\ \text{doc} \end{array} \right\} \text{form}^*)$$

▷ Define new method for generic function *foo*. *spec-vars* specialize to either being of *class* or being **eq1** *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.

($\left\{ \begin{array}{l} \text{gadd-method} \\ \text{gremove-method} \end{array} \right\}$ *generic-function method*)

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

(**gfind-method** *generic-function qualifiers specializers* [*error* \square])

▷ Return suitable method, or signal **error**.

(**gcompute-applicable-methods** *generic-function args*)

▷ List of methods suitable for *args*, most specific first.

(**fcall-next-method** *arg** [current args])

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

(**gno-applicable-method** *generic-function arg**)

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

($\left\{ \begin{array}{l} \text{finvalid-method-error} \\ \text{fmethod-combination-error} \end{array} \right\}$ *control arg**)

▷ Signal **error** on applicable method with invalid qualifiers, or on method combination. For *control* and *args* see **format**, page 36.

(**gno-next-method** *generic-function method arg**)

▷ Called on invocation of **call-next-method** when there is no next method. Default method signals **error**. Not to be called by user.

(**gfunction-keywords** *method*)

▷ Return list of keyword parameters of *method* and \mathbb{T} if other keys are allowed.

(**gmethod-qualifiers** *method*) ▷ List of qualifiers of *method*.

10.3 Method Combination Types

standard

▷ Evaluate most specific **:around** method supplying the values of the generic function. From within this method, **fcall-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 **fcall-next-method** if any, or of the generic function; and which can call less specific primary methods via **fcall-next-method**. After its return, call all **:after** methods, least specific first.

and|or|append|list|nconc|progn|max|min|+

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

(**mdefine-method-combination** *c-type* $\left\{ \begin{array}{l} \text{:documentation } \widehat{\text{string}} \\ \text{:identity-with-one-argument } \text{bool}_{\mathbb{NTT}} \\ \text{:operator } \text{operator}_{\text{c-type}} \end{array} \right\}$)

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

```
(mdefine-method-combination c-type (ord-λ*) ((group
  { *
    (qualifier* [*])
    predicate
  }
  { :description control
    :order { :most-specific-first | :most-specific-last | :most-specific-first }
    :required bool
  }
  { (:arguments method-combination-λ*)
    (:generic-function symbol)
    (declare decl*)*
    doc
  } bodyP*
  )*)
```

▷ **Long Form.** Define new **method-combination** *c-type*. A call to a generic function using *c-type* will be equivalent to a call to the forms returned by *body** with *ord-λ** bound to *c-arg** (cf. *mdefgeneric*), with *symbol* bound to the generic function, with *method-combination-λ** bound to the arguments of the generic function, and with *groups* bound to lists of methods. An applicable method becomes a member of the leftmost *group* whose *predicate* or *qualifiers* match. Methods can be called via **mcall-method**. Lambda lists (*ord-λ**) and (*method-combination-λ**) according to *ord-λ* on page 17, the latter enhanced by an optional **&whole** argument.

```
(mcall-method
  { method
    { (mmake-method form) }
  } [ ( { next-method
    { (mmake-method form) }
  } ) ] )
```

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

11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 31.

```
(mdefine-condition foo (parent-type* condition)
  { slot
    { (:reader reader)*
      (:writer { writer
        { (self writer) }
      })*
      (:accessor accessor)*
      :allocation { :instance
        { :class }
      }
      { :initarg :initarg-name }*
      :initform form
      :type type
      :documentation slot-doc
    }
  }
  { (:default-initargs { name value }*)
    (:documentation condition-doc)
    (:report { string
      { 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 *initarg-name*; it is readable via (*reader i*) or (*accessor i*), and writable via (*writer value i*) or (**self** (*accessor i value*)). With **:allocation :class**, *slot* is shared by all conditions of type *foo*. A condition is reported by *string* or by *report-function* of arguments *condition* and *stream*.

(*f* **make-condition** *condition-type* *{:initarg-name value}**)

▷ Return new instance of *condition-type*.

$\left(\begin{array}{l} \text{f} \text{signal} \\ \text{f} \text{warn} \\ \text{f} \text{error} \end{array} \right) \left\{ \begin{array}{l} \textit{condition} \\ \textit{condition-type} \textit{ } \{:\textit{initarg-name value}\}^* \\ \textit{control arg}^* \end{array} \right\}$

▷ Unless handled, signal as **condition**, **warning** or **error**, respectively, *condition* or a new instance of *condition-type* or, with *f* **format** *control* and *args* (see page 36), **simple-condition**, **simple-warning**, or **simple-error**, respectively. From *f* **signal** and *f* **warn**, return NIL.

(*f* **error** *continue-control*

$\left\{ \begin{array}{l} \textit{condition} \textit{ } \textit{continue-arg}^* \\ \textit{condition-type} \textit{ } \{:\textit{initarg-name value}\}^* \\ \textit{control arg}^* \end{array} \right\}$)

▷ Unless handled, signal as correctable **error** *condition* or a new instance of *condition-type* or, with *f* **format** *control* and *args* (see page 36), **simple-error**. In the debugger, use *f* **format** arguments *continue-control* and *continue-args* to tag the continue option. Return NIL.

(*m* **ignore-errors** *form*^{P*})

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

(*f* **invoke-debugger** *condition*)

▷ Invoke debugger with *condition*.

(*m* **assert** *test* [*(place)**)

$\left[\begin{array}{l} \textit{condition} \textit{ } \textit{continue-arg}^* \\ \textit{condition-type} \textit{ } \{:\textit{initarg-name value}\}^* \\ \textit{control arg}^* \end{array} \right]$)

▷ If *test*, which may depend on *places*, returns NIL, signal as correctable **error** *condition* or a new instance of *condition-type* or, with *f* **format** *control* and *args* (see page 36), **error**. When using the debugger's continue option, *places* can be altered before re-evaluation of *test*. Return NIL.

(*m* **handler-case** *foo*

(type [*var*]) (**declare** $\widehat{\textit{decl}}$ *)* *condition-form*^{P*}*)

[(no-error (*ord-λ**) (**declare** $\widehat{\textit{decl}}$ *)* *form*^{P*}*)])

▷ 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 17 for (*ord-λ**)).

(*m* **handler-bind** (*(condition-type handler-function)**) *form*^{P*})

▷ Return values of *forms* after evaluating them with *condition-types* dynamically bound to their respective *handler-functions* of argument condition.

(*m* **with-simple-restart** ($\left\{ \begin{array}{l} \textit{restart} \\ \text{NIL} \end{array} \right\}$ *control arg**) *form*^{P*})

▷ Return values of *forms* unless *restart* is called during their evaluation. In this case, describe *restart* using *f* **format** *control* and *args* (see page 36) and return NIL and T.

(*m* **restart-case** *form* (*restart* (*ord-λ**) $\left\{ \begin{array}{l} \text{:interactive} \textit{ } \textit{arg-function} \\ \text{:report} \left\{ \begin{array}{l} \textit{report-function} \\ \textit{string}^{\text{restart}} \end{array} \right\} \\ \text{:test} \textit{ } \textit{test-function}^{\square} \end{array} \right\}$)

(declare $\widehat{\textit{decl}}$ *)* *restart-form*^{P*}*)

▷ 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 17 for *ord-λ** .

(*m*restart-bind (($\widehat{\begin{matrix} \text{restart} \\ \text{NIL} \end{matrix}}$) restart-function

{ (:interactive-function arg-function
:report-function report-function
:test-function test-function }*) form^{P*})

▷ 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*.

(*f*invoke-restart *restart* *arg**)

(*f*invoke-restart-interactively *restart*)

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

{ (*f*find-restart
*f*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.

(*f*restart-name *restart*) ▷ Name of restart.

{ (*f*abort
*f*muffle-warning
*f*continue
*f*store-value *value*
*f*use-value *value*) } [*condition*NIL]

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

(*m*with-condition-restarts *condition* *restarts* form^{P*})

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

(*f*arithmetic-error-operation *condition*)

(*f*arithmetic-error-operands *condition*)

▷ List of function or of its operands respectively, used in the operation which caused *condition*.

(*f*cell-error-name *condition*)

▷ Name of cell which caused *condition*.

(*f*unbound-slot-instance *condition*)

▷ Instance with unbound slot which caused *condition*.

(*f*print-not-readable-object *condition*)

▷ The object not readably printable under *condition*.

(*f*package-error-package *condition*)

(*f*file-error-pathname *condition*)

(*f*stream-error-stream *condition*)

▷ Package, path, or stream, respectively, which caused the *condition* of indicated type.

(*f*type-error-datum *condition*)

(*f*type-error-expected-type *condition*)

▷ Object which caused *condition* of type **type-error**, or its expected type, respectively.

(*f*simple-condition-format-control *condition*)

(*f*simple-condition-format-arguments *condition*)

▷ Return *f*format control or list of *f*format arguments, respectively, of *condition*.

break-on-signalsNIL

▷ Condition type debugger is to be invoked on.

v*debugger-hook*_{NIL}

▷ 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.

(**f**typep *foo type* [*environment*_{NIL}]) ▷ T if *foo* is of *type*.

(**f**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.

(**s**the *type* *form*) ▷ Declare values of form to be of *type*.

(**f**coerce *object type*) ▷ Coerce object into *type*.

(**m**typecase *foo* (*type a-form*^{P*})* [(otherwise) *b-form*_{NIL}^{P*}])

▷ Return values of the first a-form* whose *type* is *foo* of. Return values of b-forms if no *type* matches.

({**m**typecase
}**m**ctypecase) *foo* (*type form*^{P*})*

▷ Return values of the first form* whose *type* is *foo* of. Signal non-correctable/correctable **type-error** if no *type* matches.

(**f**type-of *foo*) ▷ Type of foo.

(**m**check-type *place type* [*string*_{{a an} type}])

▷ Signal correctable **type-error** if *place* is not of *type*. Return NIL.

(**f**stream-element-type *stream*) ▷ Type of *stream* objects.

(**f**array-element-type *array*) ▷ Element type *array* can hold.

(**f**upgraded-array-element-type *type* [*environment*_{NIL}])

▷ Element type of most specialized array capable of holding elements of *type*.

(**m**deftype *foo* (*macro-λ**) (**declare** *decl**)* [*doc*] *form*^{P*})

▷ Define type foo which when referenced as (*foo arg**) (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 18 but with default value of * instead of NIL. *forms* are enclosed in an implicit **sblock** named *foo*.

(**eq** *foo*)

(**member** *foo**) ▷ Specifier for a type comprising *foo* or *foos*.

(**satisfies** *predicate*)

▷ Type specifier for all objects satisfying *predicate*.

(**mod** *n*) ▷ Type specifier for all non-negative integers < *n*.

(**not** *type*) ▷ Complement of type.

(**and** *type**_⊓) ▷ Type 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.

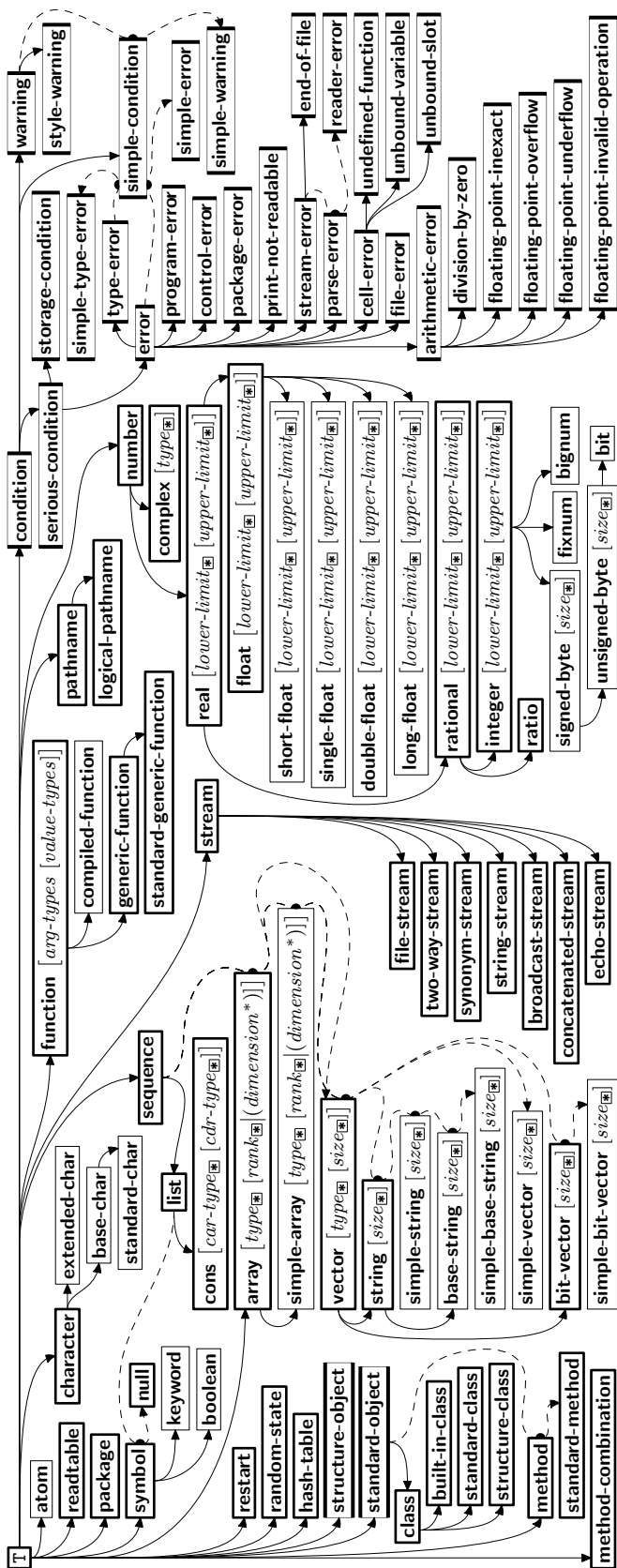


Figure 2: Precedence Order of System Classes (), Classes (), Types (), and Condition Types ().

Every type is also a supertype of NIL, the empty type.

13 Input/Output

13.1 Predicates

(*f* **streamp** *foo*)

(*f* **pathnamep** *foo*) ▷ T if *foo* is of indicated type.

(*f* **readtablep** *foo*)

(*f* **input-stream-p** *stream*)

(*f* **output-stream-p** *stream*)

(*f* **interactive-stream-p** *stream*)

(*f* **open-stream-p** *stream*)

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

(*f* **pathname-match-p** *path wildcard*)

▷ T if *path* matches *wildcard*.

(*f* **wild-pathname-p** *path* [[:host|:device|:directory|:name|:type|:version[NIL]]])

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

13.2 Reader

{ (*f* **y-or-n-p**)
(*f* **yes-or-no-p**) } [*control arg**]

▷ Ask user a question and return T or NIL depending on their answer. See page 36, *f* **format**, for *control* and *args*.

(*m* **with-standard-io-syntax** *form^P**)

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

{ (*f* **read**)
(*f* **read-preserving-whitespace**) } [*stream* v*standard-input* [*eof-err* T
[*eof-val* NIL [*recursive* NIL]]]]])

▷ Read printed representation of object.

(*f* **read-from-string** *string* [*eof-error* T [*eof-val* NIL

[{ :start *start* T
:end *end* NIL
:preserve-whitespace *bool* NIL }]]])

▷ Return object read from string and zero-indexed position₂ of next character.

(*f* **read-delimited-list** *char* [*stream* v*standard-input* [*recursive* NIL]])

▷ Continue reading until encountering *char*. Return list of objects read. Signal error if no *char* is found in stream.

(*f* **read-char** [*stream* v*standard-input* [*eof-err* T [*eof-val* NIL
[*recursive* NIL]]]])

▷ Return next character from *stream*.

(*f* **read-char-no-hang** [*stream* v*standard-input* [*eof-error* T [*eof-val* NIL
[*recursive* NIL]]]])

▷ Next character from *stream* or NIL if none is available.

(*f* **peek-char** [*mode* NIL [*stream* v*standard-input* [*eof-error* T
[*eof-val* NIL [*recursive* NIL]]]])

▷ Next, or if *mode* is T, next non-whitespace character, or if *mode* is a character, next instance of it, from *stream* without removing it there.

(*f* **unread-char** *character* [*stream* v*standard-input*])

▷ Put last *f* **read-char**ed *character* back into *stream*; return NIL.

(*f* **read-byte** [*stream* v*standard-input* [*eof-err* T [*eof-val* NIL]])

▷ Read next byte from binary *stream*.

(*f* **read-line** [*stream* v*standard-input* [*eof-err* T [*eof-val* NIL
[*recursive* NIL]]]])

▷ Return line of text from *stream* and T if line has been ended by end of file.

- (*f* **read-sequence** *sequence stream* [:start *start*₀] [:end *end*_{NIL}])
 ▷ Replace elements of *sequence* between *start* and *end* with elements from binary or character *stream*. Return index of *sequence*'s first unmodified element.
- (*f* **readtable-case** *readtable*):upcase
 ▷ Case sensitivity attribute (one of **:upcase**, **:downcase**, **:preserve**, **:invert**) of *readtable*. **setfable**.
- (*f* **copy-readtable** [*from-readtable**v**readtable*] [*to-readtable*_{NIL}])
 ▷ Return copy of from-readtable.
- (*f* **set-syntax-from-char** *to-char from-char* [*to-readtable**v**readtable*] [*from-readtable*standard-readtable])
 ▷ Copy syntax of *from-char* to *to-readtable*. Return T.
- v****readtable*** ▷ Current readtable.
- v****read-base***₁₀ ▷ Radix for reading **integers** and **ratios**.
- v****read-default-float-format***single-float
 ▷ Floating point format to use when not indicated in the number read.
- v****read-suppress***_{NIL}
 ▷ If T, reader is syntactically more tolerant.
- (*f* **set-macro-character** *char function* [*non-term-p*_{NIL}] [*rt**v**readtable*])
 ▷ Make *char* a macro character associated with *function* of stream and *char*. Return T.
- (*f* **get-macro-character** *char* [*rt**v**readtable*])
 ▷ Reader macro function associated with *char*, and T if *char* is a non-terminating macro character.
- (*f* **make-dispatch-macro-character** *char* [*non-term-p*_{NIL}] [*rt**v**readtable*])
 ▷ Make *char* a dispatching macro character. Return T.
- (*f* **set-dispatch-macro-character** *char sub-char function* [*rt**v**readtable*])
 ▷ Make *function* of stream, *n*, *sub-char* a dispatch function of *char* followed by *n*, followed by *sub-char*. Return T.
- (*f* **get-dispatch-macro-character** *char sub-char* [*rt**v**readtable*])
 ▷ Dispatch function associated with *char* followed by *sub-char*.

13.3 Character Syntax

#| *multi-line-comment** **|#**

; *one-line-comment**

▷ Comments. There are stylistic conventions:

- ;;; title** ▷ Short title for a block of code.
;; intro ▷ Description before a block of code.
;; state ▷ State of program or of following code.
;*explanation* ▷ Regarding line on which it appears.
;*continuation*

(*foo** [*.* *bar*_{NIL}]) ▷ List of *foos* with the terminating cdr *bar*.

" ▷ Begin and end of a string.

'foo ▷ (*squote foo*); *foo* unevaluated.

`([foo] [,bar] [,@baz] [.,*quux*] [*bing*])
 ▷ Backquote. *squote foo* and *bing*; evaluate *bar* and splice the lists *baz* and *quux* into their elements. When nested, outermost commas inside the innermost backquote expression belong to this backquote.

#\c ▷ (*fcharacter "c"*), the character *c*.

#B*n*; **#O*n***; *n*.; **#X*n***; **#rR*n***

▷ Integer of radix 2, 8, 10, 16, or *r*; 2 ≤ *r* ≤ 36.

- n/d ▷ The **ratio** $\frac{n}{d}$.
- $\{[m].n\{\mathbf{S|F|D|L|E}\}x_{\mathbf{FO}}\}[m].[n]\{\mathbf{S|F|D|L|E}\}x\}$
 ▷ $m.n \cdot 10^x$ as **short-float**, **single-float**, **double-float**, **long-float**, or the type from ***read-default-float-format***.
- #C**($a\ b$) ▷ (f **complex** $a\ b$), the complex number $a + bi$.
- #'** foo ▷ (s **function** foo); the function named foo .
- #nA** $sequence$ ▷ n -dimensional array.
- #** $[n](foo^*)$
 ▷ Vector of some (or n) $foos$ filled with last foo if necessary.
- #** $[n]*b^*$
 ▷ Bit vector of some (or n) bs filled with last b if necessary.
- #S**($type\ \{slot\ value\}^*$) ▷ Structure of $type$.
- #P** $string$ ▷ A pathname.
- #:** foo ▷ Uninterned symbol foo .
- #.** $form$ ▷ Read-time value of $form$.
- v ***read-eval*** \square ▷ 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.
- $|c^*|; \backslash c$
 ▷ Treat arbitrary character(s) c as alphabetic preserving case.

13.4 Printer

- $\left(\begin{array}{l} f\mathbf{prin1} \\ f\mathbf{print} \\ f\mathbf{pprint} \\ f\mathbf{princ} \end{array} \right) foo\ [\widetilde{stream}_{v\mathbf{*standard-output*}}]$
 ▷ Print foo to $stream$ f **readably**, f **readably** between a newline and a space, f **readably** after a newline, or human-readably without any extra characters, respectively. f **prin1**, f **print** and f **princ** return foo .
- (f **prin1-to-string** foo)
 (f **princ-to-string** foo)
 ▷ Print foo to $string$ f **readably** or human-readably, respectively.
- (g **print-object** $object\ \widetilde{stream}$)
 ▷ Print $object$ to $stream$. Called by the Lisp printer.
- (m **print-unreadable-object** ($foo\ \widetilde{stream}\ \left\{ \begin{array}{l} \mathbf{:type}\ bool_{\mathbf{NIL}} \\ \mathbf{:identity}\ bool_{\mathbf{NIL}} \end{array} \right\}$) $form^P$)
 ▷ Enclosed in **#<** and **>**, print foo by means of $forms$ to $stream$. Return NIL.
- (f **terpri** [$\widetilde{stream}_{v\mathbf{*standard-output*}}$])
 ▷ Output a newline to $stream$. Return NIL.
- (f **fresh-line** [$\widetilde{stream}_{v\mathbf{*standard-output*}}$])
 ▷ Output a newline to $stream$ and return T unless $stream$ is already at the start of a line.

(*f*write-char *char* [*stream*_{*v**standard-output*}])

▷ Output *char* to *stream*.

(*f*write-string *string* [*stream*_{*v**standard-output*} [[:start *start*₀]]])

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

(*f*write-byte *byte* *stream*)

▷ Write *byte* to binary *stream*.

(*f*write-sequence *sequence* *stream* [[:start *start*₀]] [[:end *end*_{NIL}]])

▷ Write elements of *sequence* to binary or character *stream*.

(*f*write *foo* {
*f*write-to-string
}) {
:array *bool*
:base *radix*
:case {
:uppercase
:downcase
:capitalize
}
:circle *bool*
:escape *bool*
:gensym *bool*
:length {*int*|NIL}
:level {*int*|NIL}
:lines {*int*|NIL}
:miser-width {*int*|NIL}
:pprint-dispatch *dispatch-table*
:pretty *bool*
:radix *bool*
:readably *bool*
:right-margin {*int*|NIL}
:stream *stream*_{*v**standard-output*}
}

▷ Print *foo* to *stream* and return *foo*, or print *foo* into *string*, respectively, after dynamically setting printer variables corresponding to keyword parameters (**print-bar** becoming *:bar*). (*:stream* keyword with *f*write only).

(*f*pprint-fill *stream* *foo* [*parenthesis*₀ [*noop*]])

(*f*pprint-tabular *stream* *foo* [*parenthesis*₀ [*noop* [*n*₁₀]])])

(*f*pprint-linear *stream* *foo* [*parenthesis*₀ [*noop*]])

▷ Print *foo* to *stream*. If *foo* is a list, print as many elements per line as possible; do the same in a table with a column width of *n* ems; or print either all elements on one line or each on its own line, respectively. Return NIL. Usable with *f*format directive ~//.

(*m*pprint-logical-block (*stream* *list* {
[:prefix *string*
[:per-line-prefix *string*
[:suffix *string*₀
}

(*declare* *decl*^P)* *form*^P)

▷ Evaluate *forms*, which should print *list*, with *stream* locally bound to a pretty printing stream which outputs to the original *stream*. If *list* is in fact not a list, it is printed by *f*write. Return NIL.

(*m*pprint-pop)

▷ Take next element off *list*. If there is no remaining tail of *list*, or *v**print-length* or *v**print-circle* indicate printing should end, send element together with an appropriate indicator to *stream*.

(*f*pprint-tab {
:line
:line-relative
:section
:section-relative
}) *c* *i*

[*stream*_{*v**standard-output*}])

▷ Move cursor forward to column number *c* + *k**i*, *k* ≥ 0 being as small as possible.

(*f*pprint-indent {
:block
:current
}) *n* [*stream*_{*v**standard-output*}])

▷ Specify indentation for innermost logical block relative to leftmost position/to current position. Return NIL.

(*m*pprint-exit-if-list-exhausted)

▷ If *list* is empty, terminate logical block. Return NIL otherwise.

(*f* **pprint-newline** $\left. \begin{array}{l} \text{:linear} \\ \text{:fill} \\ \text{:miser} \\ \text{:mandatory} \end{array} \right\} [stream \underline{v*standard-output*}])$

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

*v****print-array*** ▷ If T, print arrays *f* **readably**.

*v****print-base***₁₀ ▷ Radix for printing rationals, from 2 to 36.

*v****print-case***_{uppercase}

▷ Print symbol names all uppercase (**uppercase**), all lowercase (**downcase**), capitalized (**capitalize**).

*v****print-circle***_{NIL}

▷ If T, avoid indefinite recursion while printing circular structure.

*v****print-escape***_T

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

*v****print-gensym***_T

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

*v****print-length***_{NIL}

*v****print-level***_{NIL}

*v****print-lines***_{NIL}

▷ 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***

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

*v****print-pretty*** ▷ If T, print prettily.

*v****print-radix***_{NIL}

▷ If T, print rationals with a radix indicator.

*v****print-readably***_{NIL}

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

*v****print-right-margin***_{NIL}

▷ Right margin width in ems while pretty-printing.

(*f* **set-pprint-dispatch** *type function* [*priority*₀ [*table* *v**print-pprint-dispatch*]])

▷ Install entry comprising *function* of arguments *stream* and object to print; and *priority* as *type* into *table*. If *function* is NIL, remove *type* from *table*. Return NIL.

(*f* **pprint-dispatch** *foo* [*table* *v**print-pprint-dispatch*])

▷ Return highest priority *function* associated with type of *foo* and T if there was a matching type specifier in *table*.

(*f* **copy-pprint-dispatch** [*table* *v**print-pprint-dispatch*])

▷ 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

(*m* **formatter** $\widehat{control}$)

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

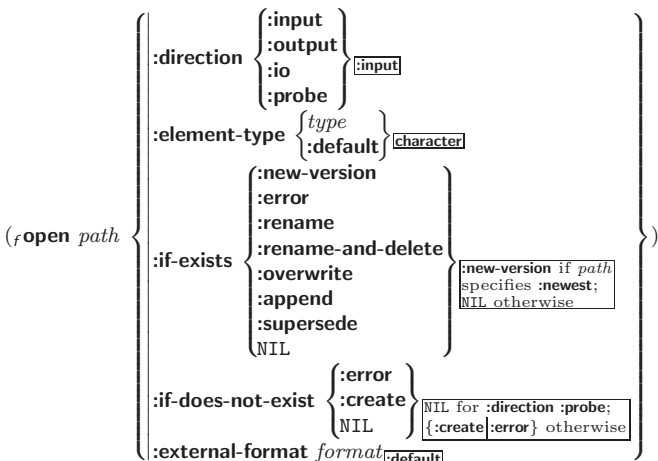
(*f* **format** {T|NIL|*out-string*|*out-stream*} *control arg**)

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

- ~ [*min-col*₀] [, [*col-inc*₁] [, [*min-pad*₀] [, [*'pad-char*₁]]]]
 [:] [C] {A|S}
 ▷ **Aesthetic/Standard.** Print argument of any type for consumption by humans/by the reader, respectively. With :, print NIL as () rather than nil; with C, add *pad-chars* on the left rather than on the right.
- ~ [*radix*₁₀] [, [*width*] [, [*'pad-char*₁] [, [*'comma-char*₀] [, [*comma-interval*₃]]]]] [:] [C] R
 ▷ **Radix.** (With one or more prefix arguments.) Print argument as number; with :, group digits *comma-interval* each; with C, always prepend a sign.
- {~R|~:R|~@R|~@: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*₁] [, [*'comma-char*₀] [, [*comma-interval*₃]]]] [:] [C] {D|B|O|X}
 ▷ **Decimal/Binary/Octal/Hexadecimal.** Print integer argument as number. With :, group digits *comma-interval* each; with C, always prepend a sign.
- ~ [*width*] [, [*dec-digits*] [, [*shift*₀] [, [*'overflow-char*] [, [*'pad-char*₁]]]]] [C] F
 ▷ **Fixed-Format Floating-Point.** With C, always prepend a sign.
- ~ [*width*] [, [*dec-digits*] [, [*exp-digits*] [, [*scale-factor*₁] [, [*'overflow-char*] [, [*'pad-char*₁] [, [*'exp-char*]]]]]]] [C] {E|G}
 ▷ **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 C, always prepend a sign.
- ~ [*dec-digits*₂] [, [*int-digits*₁] [, [*width*₀] [, [*'pad-char*₁]]]] [:] [C] \$
 ▷ **Monetary Floating-Point.** Print argument as fixed-format floating-point number. With :, put sign before any padding; with C, always prepend a sign.
- {~C|~:C|~@C|~@:C}
 ▷ **Character.** Print, spell out, print in #\ syntax, or tell how to type, respectively, argument as (possibly non-printing) character.
- {~(text ~)|~:(text ~)|~@(text ~)|~@:(text ~)}
 ▷ **Case-Conversion.** Convert *text* to lowercase, convert first letter of each word to uppercase, capitalize first word and convert the rest to lowercase, or convert to uppercase, respectively.
- {~P|~:P|~@P|~@:P}
 ▷ **Plural.** If argument *eq1* print nothing, otherwise print *s*; do the same for the previous argument; if argument *eq1* print *y*, otherwise print *ies*; do the same for the previous argument, respectively.
- ~ [*n*₁] % ▷ **Newline.** Print *n* newlines.
- ~ [*n*₁] &
 ▷ **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*₁] | ▷ **Page.** Print *n* page separators.
- ~ [*n*₁] ~ ▷ **Tilde.** Print *n* tildes.
- ~ [*min-col*₀] [, [*col-inc*₁] [, [*min-pad*₀] [, [*'pad-char*₁]]]] [:] [C] < [*nl-text* ~ [*spare*₀] [, [*width*]]::] {*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.
- ~ [:] [**@**] < { [prefix_{mn} ~;] [per-line-prefix ~@;] } body [~; suffix_{mn}] ~: [**@**] >
- ▷ **Logical Block.** Act like **pprint-logical-block** using *body* as *f***format** control string on the elements of the list argument or, with **@**, on the remaining arguments, which are extracted by **pprint-pop**. With **:**, *prefix* and *suffix* default to (and). When closed by ~@>, spaces in *body* are replaced with conditional newlines.
- {~ [n₀] i|~ [n₀] :i}
- ▷ **Indent.** Set indentation to *n* relative to leftmost/current position.
- ~ [c₀] [,i₀] [:] [**@**] T
- ▷ **Tabulate.** Move cursor forward to column number $c+ki$, $k \geq 0$ being as small as possible. With **:**, calculate column numbers relative to the immediately enclosing section. With **@**, move to column number $c_0 + c + ki$ where c_0 is the current position.
- {~ [m₀] *|~ [m₀] :*|~ [n₀] @*}
- ▷ **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 **:** or **@:**, list elements or remaining arguments should be lists of which a new one is used at each iteration step.
- ~ [x [,y [,z]]] ^
- ▷ **Escape Upward.** Leave immediately ~< ~>, ~< ~:>, ~{ ~}, ~?, or the entire *f***format** operation. With one to three prefixes, act only if $x = 0$, $x = y$, or $x \leq y \leq z$, respectively.
- ~ [i] [:] [**@**] [[{text ~;} * text] [~;; default] ~]
- ▷ **Conditional Expression.** Use the zero-indexed argument (or *i*th if given) *text* as a *f***format** control subclause. With **:**, 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. Use the only *text* and leave the argument to be read again if it is T.
- {~?|~@?}
- ▷ **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}*] [:] [**@**] / [package [:]:cl-user:]function/
- ▷ **Call Function.** Call all-uppercase *package::function* with the arguments stream, format-argument, colon-p, at-sign-p and *prefixes* for printing format-argument.
- ~ [:] [**@**] W
- ▷ **Write.** Print argument of any type obeying every printer control variable. With **:**, pretty-print. With **@**, print without limits on length or depth.
- {V|#}
- ▷ In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.

13.6 Streams



▷ Open file-stream to *path*.

(*f* **make-concatenated-stream** *input-stream**)

(*f* **make-broadcast-stream** *output-stream**)

(*f* **make-two-way-stream** *input-stream-part* *output-stream-part*)

(*f* **make-echo-stream** *from-input-stream* *to-output-stream*)

(*f* **make-synonym-stream** *variable-bound-to-stream*)

▷ Return stream of indicated type.

(*f* **make-string-input-stream** *string* [*start* T] [*end* NIL])

▷ Return a string-stream supplying the characters from *string*.

(*f* **make-string-output-stream** [**:element-type** *type* character])

▷ Return a string-stream accepting characters (available via *f* **get-output-stream-string**).

(*f* **concatenated-stream-streams** *concatenated-stream*)

(*f* **broadcast-stream-streams** *broadcast-stream*)

▷ Return list of streams *concatenated-stream* still has to read from/*broadcast-stream* is broadcasting to.

(*f* **two-way-stream-input-stream** *two-way-stream*)

(*f* **two-way-stream-output-stream** *two-way-stream*)

(*f* **echo-stream-input-stream** *echo-stream*)

(*f* **echo-stream-output-stream** *echo-stream*)

▷ Return source stream or sink stream of *two-way-stream*/*echo-stream*, respectively.

(*f* **synonym-stream-symbol** *synonym-stream*)

▷ Return symbol of *synonym-stream*.

(*f* **get-output-stream-string** *string-stream*)

▷ Clear and return as a string characters on *string-stream*.

(*f* **file-position** *stream* [{ **:start** }] [**:end** }] [*position*])

▷ Return position within stream, or set it to *position* and return T on success.

(*f* **file-string-length** *stream* *foo*)

▷ Length *foo* would have in *stream*.

(*f* **listen** [*stream* v*standard-input*])

▷ T if there is a character in input *stream*.

(*f* **clear-input** [*stream* v*standard-input*])

▷ Clear input from *stream*, return NIL.

{ *f* **clear-output** } [*stream* v*standard-output*]
 { *f* **force-output** }
 { *f* **finish-output** }

▷ End output to *stream* and return NIL immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.

(*f* **close** *stream* [:**abort** *bool*_{NIL}])
 ▷ Close *stream*. Return T if *stream* had been open. If **:abort** is T, delete associated file.

(*m* **with-open-file** (*stream path open-arg**) (**declare** *decl**)^{*} *form*^{P*})
 ▷ Use *f* **open** with *open-args* to temporarily create *stream* to *path*; return values of forms.

(*m* **with-open-stream** (*foo stream*) (**declare** *decl**)^{*} *form*^{P*})
 ▷ Evaluate *forms* with *foo* locally bound to *stream*. Return values of forms.

(*m* **with-input-from-string** (*foo string* {**:index** *index*
:start *start*₀
:end *end*_{NIL}}) (**declare** *decl**)^{*} *form*^{P*})
 ▷ Evaluate *forms* with *foo* locally bound to input **string-stream** from *string*. Return values of forms; store next reading position into *index*.

(*m* **with-output-to-string** (*foo* [*string*_{NIL} [:**element-type** *type*_{character}]]) (**declare** *decl**)^{*} *form*^{P*})
 ▷ Evaluate *forms* with *foo* locally bound to an output **string-stream**. Append output to *string* and return values of forms if *string* is given. Return string containing output otherwise.

(*f* **stream-external-format** *stream*)
 ▷ External file format designator.

v*terminal-io* ▷ Bidirectional stream to user terminal.

v*standard-input*

v*standard-output*

v*error-output*

▷ Standard input stream, standard output stream, or standard error output stream, respectively.

v*debug-io*

v*query-io*

▷ Bidirectional streams for debugging and user interaction.

13.7 Pathnames and Files

(*f* **make-pathname** {**:host** {*host*|NIL|:**unspecific**}
:device {*device*|NIL|:**unspecific**}
:directory { {*directory*|:**wild**|NIL|:**unspecific**}
 {**:absolute** | **:relative**} {*directory*
:wild
:wild-inferiors
:up
:back}^{*} }
:name {*file-name*|:**wild**|NIL|:**unspecific**}
:type {*file-type*|:**wild**|NIL|:**unspecific**}
:version {:**newest**|*version*|:**wild**|NIL|:**unspecific**}
:defaults *path*_{host from v*default-pathname-defaults*}
:case {:**local**|:**common**}_{local} })

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

{*f* **pathname-host**
f **pathname-device**
f **pathname-directory**
f **pathname-name**
f **pathname-type** } *path-or-stream* [:**case** {:**local**
:common}_{local}])

(*f* **pathname-version** *path-or-stream*)
 ▷ Return pathname component.

(*f* **parse-namestring** *foo* [*host*
 [*default-pathname* *v****default-pathname-defaults***
 {
 :start *start*₀
 :end *end*_{NIL}
 :junk-allowed *bool*_{NIL}
 }]])
 ▷ Return pathname converted from string, pathname, or stream *foo*; and position where parsing stopped.

(*f* **merge-pathnames** *path-or-stream*
 [*default-path-or-stream* *v****default-pathname-defaults***
 [*default-version* **newest**]])
 ▷ Return pathname made by filling in components missing in *path-or-stream* from *default-path-or-stream*.

*v****default-pathname-defaults***
 ▷ Pathname to use if one is needed and none supplied.

(*f* **user-homedir-pathname** [*host*]) ▷ User's home directory.

(*f* **enough-namestring** *path-or-stream*
 [*root-path* *v****default-pathname-defaults***])
 ▷ Return minimal path string that sufficiently describes the path of *path-or-stream* relative to *root-path*.

(*f* **namestring** *path-or-stream*)

(*f* **file-namestring** *path-or-stream*)

(*f* **directory-namestring** *path-or-stream*)

(*f* **host-namestring** *path-or-stream*)

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

(*f* **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.

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

(*f* **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** | **NEWEST** }]]".

(*f* **logical-pathname-translations** *logical-host*)

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

(*f* **load-logical-pathname-translations** *logical-host*)

▷ Load *logical-host*'s translations. Return NIL if already loaded; return T if successful.

(*f* **translate-logical-pathname** *path-or-stream*)

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

(*f* **probe-file** *file*)

(*f* **truename** *file*)

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

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

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

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

(*f* **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.

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

(*f* **directory** *path*) ▷ List of pathnames matching *path*.

(*f*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 21.

14.1 Predicates

(*f*symbolp *foo*)
 (*f*packagep *foo*) ▷ T if *foo* is of indicated type.
 (*f*keywordp *foo*)

14.2 Packages

bar|keyword:*bar* ▷ Keyword, evaluates to :bar.

package:*symbol* ▷ Exported *symbol* of *package*.

package::*symbol* ▷ Possibly unexported *symbol* of *package*.

(*m*defpackage *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.

(*f*make-package *foo* {(:nicknames (*nick**)NTL)
 (:use (*used-package**)*)})
 ▷ Create package *foo*.

(*f*rename-package *package* *new-name* [*new-nicknames*NTL])
 ▷ Rename *package*. Return renamed package.

(*m*in-package *foo*) ▷ Make package *foo* current.

{
 (*f*use-package
 (*f*unuse-package)
 } *other-packages* [*package*v*package*])
 ▷ Make exported symbols of *other-packages* available in *package*, or remove them from *package*, respectively. Return T.

(*f*package-use-list *package*)
 (*f*package-used-by-list *package*)
 ▷ List of other packages used by/using *package*.

(*f*delete-package *package*)
 ▷ Delete *package*. Return T if successful.

v*package*common-lisp-user ▷ The current package.

(*f*list-all-packages) ▷ List of registered packages.

(*f*package-name *package*) ▷ Name of *package*.

(*f*package-nicknames *package*) ▷ Nicknames of *package*.

(*f*find-package *name*) ▷ Package with *name* (case-sensitive).

(*f*find-all-symbols *foo*)
 ▷ List of symbols *foo* from all registered packages.

$\left\{ \begin{array}{l} \text{fintern} \\ \text{ffind-symbol} \end{array} \right\} \text{foo } [package_{\underline{v}*\text{package}*}]$

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

$(\text{funintern } symbol [package_{\underline{v}*\text{package}*}])$

▷ Remove *symbol* from *package*, return **T** on success.

$\left\{ \begin{array}{l} \text{fimport} \\ \text{fshadowing-import} \end{array} \right\} symbols [package_{\underline{v}*\text{package}*}]$

▷ Make *symbols* internal to *package*. Return **T**. In case of a name conflict signal correctable **package-error** or shadow the old symbol, respectively.

$(\text{fshadow } symbols [package_{\underline{v}*\text{package}*}])$

▷ Make *symbols* of *package* shadow any otherwise accessible, equally named symbols from other packages. Return **T**.

$(\text{fpackage-shadowing-symbols } package)$

▷ List of symbols of *package* that shadow any otherwise accessible, equally named symbols from other packages.

$(\text{fexport } symbols [package_{\underline{v}*\text{package}*}])$

▷ Make *symbols* external to *package*. Return **T**.

$(\text{funexport } symbols [package_{\underline{v}*\text{package}*}])$

▷ Revert *symbols* to internal status. Return **T**.

$\left\{ \begin{array}{l} \text{mdo-symbols} \\ \text{mdo-external-symbols} \\ \text{mdo-all-symbols} \end{array} \right\} (\widehat{var} [package_{\underline{v}*\text{package}*} [result_{\underline{NIL}}]]) \left\{ \begin{array}{l} \text{declare } \widehat{decl}^* \end{array} \right\} \left\{ \begin{array}{l} \text{tag} \\ \text{form} \end{array} \right\}^*$

▷ 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 **sblock** named **NIL**.

$(\text{mwith-package-iterator } (foo \text{ packages } [:\text{internal}|\text{:external}|\text{:inherited}]) (\text{declare } \widehat{decl}^*)^* \text{ form}^P)$

▷ 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.

$(\text{frequire } module [paths_{\underline{NIL}}])$

▷ If not in **v*modules***, try *paths* to load *module* from. Signal **error** if unsuccessful. Deprecated.

$(\text{fprovide } module)$

▷ If not already there, add *module* to **v*modules***. Deprecated.

v*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*).

$(\text{fmake-symbol } name)$

▷ Make fresh, uninterned symbol *name*.

$(\text{fgensym } [s_{\underline{NIL}}])$

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

$(\text{fgentemp } [prefix_{\underline{NIL}} [package_{\underline{v}*\text{package}*}])$

▷ Intern fresh symbol in package. Deprecated.

$(\text{fcopy-symbol } symbol [props_{\underline{NIL}}])$

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

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

(*g*documentation (setf *g*documentation) *new-doc*) *foo* { 'variable | 'function | 'compiler-macro | 'method-combination | 'structure | 'type | 'setf | T }

▷ Get/set documentation string of *foo* of given type.

ct

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

cnil_{c()}

▷ Falsity; the empty list; the empty type, subtype of every type; ***standard-input***; ***standard-output***; the global environment.

14.4 Standard Packages

common-lisp_{cl}

▷ Exports the defined names of Common Lisp except for those in the **keyword** package.

common-lisp-user_{cl-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

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

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

15.2 Compilation

(*f*compile { NIL *definition* } { *name* } [*definition*])
 (setf *name*)
 ▷ 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**.

(*f*compile-file *file* { :output-file *out-path* | :verbose *bool* v*compile-verbose* | :print *bool* v*compile-print* | :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**.

(*f*compile-file-pathname *file* [:output-file *path*] [*other-keyargs*])
 ▷ Pathname *f*compile-file writes to if invoked with the same arguments.

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

v^* **compile-file** } { **pathname***NIL
 v^* **load** } { **true-name***NIL }
 ▷ Input file used by f **compile-file**/by f **load**.

v^* **compile** } { **print***
 v^* **load** } { **verbose*** }
 ▷ Defaults used by f **compile-file**/by f **load**.

(s **eval-when** ({ { **:compile-toplevel**|**compile** }
 { { **:load-toplevel**|**load** }
 { { **:execute**|**eval** } }) $form^R$)
 ▷ Return values of forms if s **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.)

(s **locally** (**declare** \widehat{decl}^*) * $form^R$)
 ▷ Evaluate $forms$ in a lexical environment with declarations $decl$ in effect. Return values of forms.

(m **with-compilation-unit** ([**:override** $bool^*$]NIL]) $form^R$)
 ▷ Return values of forms. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of $forms$.

(s **load-time-value** $form$ [$\widehat{read-only}$]NIL])
 ▷ Evaluate $form$ at compile time and treat its value as literal at run time.

(s **quote** \widehat{foo}) ▷ Return unevaluated foo .

(g **make-load-form** foo [$environment$])
 ▷ Its methods are to return a creation form which on evaluation at f **load** time returns an object equivalent to foo , and an optional initialization form which on evaluation performs some initialization of the object.

(f **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 .

(f **macro-function** $symbol$ [$environment$])
 (f **compiler-macro-function** { $name$
 { **(setf** $name$) } } [$environment$])
 ▷ Return specified macro function, or compiler macro function, respectively, if any. Return NIL otherwise. **setfable**.

(f **eval** arg)
 ▷ Return values of value of arg evaluated in global environment.

15.3 REPL and Debugging

$v+$ | $v++$ | $v+++$
 v^* | v^{**} | v^{***}
 $v/$ | $v//$ | $v///$
 ▷ 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.

(f **apropos** $string$ [$package$]NIL])
 ▷ Print interned symbols containing $string$.

(f **apropos-list** $string$ [$package$]NIL])
 ▷ List of interned symbols containing $string$.

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

(f **ed** [$file-or-function$]NIL) ▷ Invoke editor if possible.

$(\left\{ \begin{array}{l} \text{macroexpand-1} \\ \text{macroexpand} \end{array} \right\} \text{form } [\text{environment}_{\text{NIL}}])$

▷ 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.

$(\text{mtrace } \left\{ \begin{array}{l} \text{function} \\ (\text{setf } \text{function}) \end{array} \right\}^*)$

▷ Cause *functions* to be traced. With no arguments, return list of traced functions.

$(\text{muntrace } \left\{ \begin{array}{l} \text{function} \\ (\text{setf } \text{function}) \end{array} \right\}^*)$

▷ Stop *functions*, or each currently traced function, from being traced.

trace-output

▷ Output stream **mtrace** and **mtime** send their output to.

$(\text{mstep } \text{form})$

▷ Step through evaluation of *form*. Return values of form.

$(\text{fbreak } [\text{control } \text{arg}^*])$

▷ Jump directly into debugger; return NIL. See page 36, **format**, for *control* and *args*.

$(\text{mtime } \text{form})$

▷ Evaluate *forms* and print timing information to ***trace-output***. Return values of form.

$(\text{finspect } \text{foo})$ ▷ Interactively give information about *foo*.

$(\text{fdescribe } \text{foo } [\widetilde{\text{stream}}_{\text{*standard-output*}}])$

▷ Send information about *foo* to *stream*.

$(\text{gdescribe-object } \text{foo } [\widetilde{\text{stream}}])$

▷ Send information about *foo* to *stream*. Called by **fdescribe**.

$(\text{fdisassemble } \text{function})$

▷ Send disassembled representation of *function* to ***standard-output***. Return NIL.

$(\text{froom } [\{\text{NIL}|\text{default}|T\}_{\text{default}}])$

▷ Print information about internal storage management to ***standard-output***.

15.4 Declarations

$(\text{fproclaim } \text{decl})$

$(\text{mdeclaim } \widehat{\text{decl}}^*)$

▷ Globally make declaration(s) *decl*. *decl* can be: **declaration**, **type**, **ftype**, **inline**, **notinline**, **optimize**, or **special**. See below.

$(\text{declare } \widehat{\text{decl}}^*)$

▷ Inside certain forms, locally make declarations *decl**. *decl* can be: **dynamic-extent**, **type**, **ftype**, **ignorable**, **ignore**, **inline**, **notinline**, **optimize**, or **special**. See below.

$(\text{declaration } \text{foo}^*)$

▷ Make *foos* names of declarations.

$(\text{dynamic-extent } \text{variable}^* (\text{function } \text{function})^*)$

▷ Declare lifetime of *variables* and/or *functions* to end when control leaves enclosing block.

$([\text{type}] \text{type } \text{variable}^*)$

$(\text{ftype } \text{type } \text{function}^*)$

▷ Declare *variables* or *functions* to be of *type*.

$(\left\{ \begin{array}{l} \text{ignorable} \\ \text{ignore} \end{array} \right\} \left\{ \begin{array}{l} \text{var} \\ (\text{function } \text{function}) \end{array} \right\}^*)$

▷ Suppress warnings about used/unused bindings.

(**inline** *function**)

(**notinline** *function**)

▷ Tell compiler to integrate/not to integrate, respectively, called *functions* into the calling routine.

(**optimize** $\left\{ \begin{array}{l} \text{compilation-speed} | (\text{compilation-speed } n_{\text{3}}) \\ \text{debug} | (\text{debug } n_{\text{3}}) \\ \text{safety} | (\text{safety } n_{\text{3}}) \\ \text{space} | (\text{space } n_{\text{3}}) \\ \text{speed} | (\text{speed } n_{\text{3}}) \end{array} \right\}$)

▷ 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

(**f**get-internal-real-time)

(**f**get-internal-run-time)

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

cinternal-time-units-per-second

▷ Number of clock ticks per second.

(**f**encode-universal-time *sec min hour date month year* [*zone*_{current}])

(**f**get-universal-time)

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

(**f**decode-universal-time *universal-time* [*time-zone*_{current}])

(**f**get-decoded-time)

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

(**f**short-site-name)

(**f**long-site-name)

▷ String representing physical location of computer.

$\left(\begin{array}{l} \text{f} \text{lisp-implementation} \\ \text{f} \text{software} \\ \text{f} \text{machine} \end{array} \right) - \left\{ \begin{array}{l} \text{type} \\ \text{version} \end{array} \right\}$

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

(**f**machine-instance) ▷ Computer name.

Index

- " 33
' 33
(33
) 44
) 33
* 3, 30, 31, 41, 45
** 41, 45
*** 45
*BREAK-
 ON-SIGNALS* 29
*COMPILE-FILE-
 PATHNAME* 45
*COMPILE-FILE-
 TRUENAME* 45
COMPILE-PRINT 45
*COMPILE-
 VERBOSE* 45
DEBUG-IO 40
DEBUGGER-HOOK
 30
*DEFAULT-
 PATHNAME-
 DEFAULTS* 41
ERROR-OUTPUT 40
FEATURES 34
*GENSYM-
 COUNTER* 43
LOAD-PATHNAME
 45
LOAD-PRINT 45
LOAD-TRUENAME
 45
LOAD-VERBOSE 45
*MACROEXPAND-
 HOOK* 46
MODULES 43
PACKAGE 42
PRINT-ARRAY 36
PRINT-BASE 36
PRINT-CASE 36
PRINT-CIRCLE 36
PRINT-ESCAPE 36
PRINT-GENSYM 36
PRINT-LENGTH 36
PRINT-LEVEL 36
PRINT-LINES 36
*PRINT-
 MISER-WIDTH* 36
*PRINT-PPRINT-
 DISPATCH* 36
PRINT-PRETTY 36
PRINT-RADIX 36
PRINT-READABLY
 36
*PRINT-RIGHT-
 MARGIN* 36
QUERY-IO 40
RANDOM-STATE 4
READ-BASE 33
*READ-DEFAULT-
 FLOAT-FORMAT*
 33
READ-EVAL 34
READ-SUPPRESS 33
READTABLE 33
STANDARD-INPUT
 40
*STANDARD-
 OUTPUT* 40
TERMINAL-IO 40
TRACE-OUTPUT 46
+ 3, 26, 45
++ 45
+++ 45
, 33
,. 33
,@ 33
- 3, 45
. 33
/ 3, 34, 45
// 45
/// 45
/= 3
: 42
:: 42
:ALLOW-
 OTHER-KEYS 19
; 33
< 3
<= 3
= 3, 21
> 3
>= 3
\ 34
38
#\ 33
#' 34
#(34
34
#+ 34
#- 34
#. 34
#: 34
#< 34
#= 34
#A 34
#B 33
#C(34
#O 33
#P 34
#R 33
#S(34
#X 33
34
#| |# 33
- &ALLOW-
 OTHER-KEYS 19
&AUX 19
&BODY 19
&ENVIRONMENT 19
&KEY 19
&OPTIONAL 19
&REST 19
&WHOLE 19
~(~) 37
~* 38
~/ / 38
~< ~> 38
~< ~> 38
~? 38
~A 37
~B 37
~C 37
~D 37
~E 37
~F 37
~G 37
~I 38
~O 37
~P 37
~R 37
~S 37
~T 38
~W 38
~X 37
~[~] 38
~\$ 37
~% 37
~& 37
~ 38
~- 37
~| 37
~{ ~} 38
~ 37
~← 37
` 33
| 34
1+ 3
1- 3
- ABORT 29
ABOVE 21
ABS 4
ACONS 9
ACOS 3
ACOSH 4
ACROSS 21
ADD-METHOD 26
ADJOIN 9
ADJUST-ARRAY 10
ADJUSTABLE-
 ARRAY-P 10
ALLOCATE-INSTANCE
 25
ALPHA-CHAR-P 6
ALPHANUMERICP 6
ALWAYS 23
AND
 20, 21, 23, 26, 30, 34
APPEND 9, 23, 26
APPENDING 23
APPLY 17
APPROPOS 45
APPROPOS-LIST 45
AREF 10
ARITHMETIC-ERROR
 31
ARITHMETIC-ERROR-
 OPERANDS 29
ARITHMETIC-ERROR-
 OPERATION 29
ARRAY 31
ARRAY-DIMENSION 11
ARRAY-DIMENSION-
 LIMIT 11
ARRAY-DIMENSIONS
 11
ARRAY-
 DISPLACEMENT 11
ARRAY-
 ELEMENT-TYPE 30
ARRAY-HAS-
 FILL-POINTER-P 10
ARRAY-IN-BOUNDS-P
 10
ARRAY-RANK 11
ARRAY-RANK-LIMIT
 11
ARRAY-ROW-
 MAJOR-INDEX 11
ARRAY-TOTAL-SIZE
 11
ARRAY-TOTAL-
 SIZE-LIMIT 11
ARRAYP 10
AS 21
ASH 5
ASIN 3
ASINH 4
ASSERT 28
ASSOC 9
ASSOC-IF 9
ASSOC-IF-NOT 9
ATAN 3
ATANH 4
ATOM 8, 31
- BASE-CHAR 31
BASE-STRING 31
BEING 21
- BELOW 21
BIGNUM 31
BIT 11, 31
BIT-AND 11
BIT-ANDC1 11
BIT-ANDC2 11
BIT-EQV 11
BIT-IOR 11
BIT-NAND 11
BIT-NOR 11
BIT-NOT 11
BIT-ORC1 11
BIT-ORC2 11
BIT-VECTOR 31
BIT-VECTOR-P 10
BIT-XOR 11
BLOCK 20
BOOLE 4
BOOLE-1 4
BOOLE-2 4
BOOLE-AND 5
BOOLE-ANDC1 5
BOOLE-ANDC2 5
BOOLE-C1 4
BOOLE-C2 4
BOOLE-CLR 4
BOOLE-EQV 5
BOOLE-IOR 5
BOOLE-NAND 5
BOOLE-NOR 5
BOOLE-ORC1 5
BOOLE-ORC2 5
BOOLE-SET 4
BOOLE-XOR 5
BOOLEAN 31
BOTH-CASE-P 6
BOUNDP 15
BREAK 46
BROADCAST-
 STREAM 31
BROADCAST-
 STREAM-STREAMS
 39
BUILT-IN-CLASS 31
BUTLAST 9
BY 21
BYTE 5
BYTE-POSITION 5
BYTE-SIZE 5
- CAAR 8
CADR 8
CALL-ARGUMENTS-
 LIMIT 18
CALL-METHOD 27
CALL-NEXT-METHOD
 26
CAR 8
CASE 20
CATCH 20
CCASE 20
CDAR 8
CDDR 8
CDR 8
CEILING 4
CELL-ERROR 31
CELL-ERROR-NAME
 29
CERROR 28
CHANGE-CLASS 24
CHAR 8
CHAR-CODE 7
CHAR-CODE-LIMIT 7
CHAR-DOWNCASE 7
CHAR-EQUAL 6
CHAR-GREATERP 7
CHAR-INT 7
CHAR-LESSP 7
CHAR-NAME 7
CHAR-NOT-EQUAL 6
CHAR-
 NOT-GREATERP 7
CHAR-NOT-LESSP 7
CHAR-UPCASE 7
CHAR/= 6
CHAR< 6
CHAR<= 6
CHAR= 6
CHAR> 6
CHAR>= 6
CHARACTER 7, 31, 33
CHARACTERP 6
CHECK-TYPE 30
CIS 4
CL 44
CL-USER 44
CLASS 31
CLASS-NAME 24
CLASS-OF 24
CLEAR-INPUT 39
CLEAR-OUTPUT 39
CLOSE 40
CLQR 1
CLRHASH 14
CODE-CHAR 7
COERCE 30
COLLECT 23
COLLECTING 23
COMMON-LISP 44
COMMON-LISP-USER
 44
COMPILATION-SPEED
 47
COMPILE 44
COMPILE-FILE 44
- COMPILE-FILE-
 PATHNAME 44
COMPILED-
 FUNCTION 31
COMPILED-
 FUNCTION-P 44
COMPILER-MACRO 44
COMPILER-MACRO-
 FUNCTION 45
COMPLEMENT 17
COMPLEX 4, 31, 34
COMPLEXP 3
COMPUTE-
 APPLICABLE-
 METHODS 26
COMPUTE-RESTARTS
 29
CONCATENATE 12
CONCATENATED-
 STREAM 31
CONCATENATED-
 STREAM-STREAMS
 39
COND 19
CONDITION 31
CONJUGATE 4
CONS 8, 31
CONSP 8
CONSTANTLY 17
CONSTANTP 15
CONTINUE 29
CONTROL-ERROR 31
COPY-ALIST 9
COPY-LIST 9
COPY-PPRINT-
 DISPATCH 36
COPY-READTABLE 33
COPY-SEQ 14
COPY-STRUCTURE 15
COPY-SYMBOL 43
COPY-TREE 10
COS 3
COSH 3
COUNT 12, 23
COUNT-IF 12
COUNT-IF-NOT 12
COUNTING 23
CTYPECASE 30
- DEBUG 47
DEC 3
DECLAIM 46
DECLARATION 46
DECLARE 46
DECODE-FLOAT 6
DECODE-UNIVERSAL-
 TIME 47
DEFCLASS 24
DEFCONSTANT 16
DEFGENERIC 25
DEFINE-COMPILER-
 MACRO 18
DEFINE-CONDITION
 27
DEFINE-METHOD-
 COMBINATION
 26, 27
DEFINE-MODIFY-
 MACRO 19
DEFINE-SETF-
 EXPANDER 19
DEFINE-SYMBOL-
 MACRO 18
DEFMACRO 18
DEFMETHOD 25
DEFPACKAGE 42
DEFPARAMETER 16
DEFSETF 18
DEFSTRUCT 15
DEFTYPE 30
DEFUN 17
DEFVAR 16
DELETE 13
DELETE-DUPLICATES
 13
DELETE-FILE 41
DELETE-IF 13
DELETE-IF-NOT 13
DELETE-PACKAGE 42
DENOMINATOR 4
DEPOSIT-FIELD 5
DESCRIBE 46
DESCRIBE-OBJECT 46
DESTRUCTURING-
 BIND 17
DIGIT-CHAR 7
DIGIT-CHAR-P 6
DIRECTORY 41
DIRECTORY-
 NAMESTRING 41
DISASSEMBLE 46
DIVISION-BY-ZERO 31
DO 21, 23
DO-ALL-SYMBOLS 43
DO-EXTERNAL-
 SYMBOLS 43
DO-SYMBOLS 43
DO* 21
DOCUMENTATION 44
DOING 23
DOLIST 21
DOTIMES 21
DOUBLE-FLOAT 31, 34
DOUBLE-
 FLOAT-EPSILON 6

- DOUBLE-FLOAT-NEGATIVE-EPSILON 6
 DOWNFROM 21
 DOWNTO 21
 DPB 5
 DRIBBLE 45
 DYNAMIC-EXTENT 46
- EACH 21
 ECASE 20
 ECHO-STREAM 31
 ECHO-STREAM-INPUT-STREAM 39
 ECHO-STREAM-OUTPUT-STREAM 39
 ED 45
 EIGHTH 8
 ELSE 23
 ELT 12
 ENCODE-UNIVERSAL-TIME 47
 END 23
 END-OF-FILE 31
 ENDP 8
 ENOUGH-NAMESTRING 41
 ENSURE-DIRECTORIES-EXIST 42
 ENSURE-GENERIC-FUNCTION 25
 EQ 15
 EQL 15, 30
 EQUAL 15
 EQUALP 15
 ERROR 28, 31
 ETYPCASE 30
 EVAL 45
 EVAL-WHEN 45
 EVENP 3
 EVERY 12
 EXP 3
 EXPORT 43
 EXPT 3
 EXTENDED-CHAR 31
 EXTERNAL-SYMBOL 23
 EXTERNAL-SYMBOLS 23
- FBOUNDP 16
 FCEILING 4
 FDEFINITION 18
 FFLOOR 4
 FIFTH 8
 FILE-AUTHOR 41
 FILE-ERROR 31
 FILE-ERROR-PATHNAME 29
 FILE-LENGTH 41
 FILE-NAMESTRING 41
 FILE-POSITION 39
 FILE-STREAM 31
 FILE-STRING-LENGTH 39
 FILE-WRITE-DATE 41
 FILL 12
 FILL-POINTER 11
 FINALLY 23
 FIND 13
 FIND-ALL-SYMBOLS 42
 FIND-CLASS 24
 FIND-IF 13
 FIND-IF-NOT 13
 FIND-METHOD 26
 FIND-PACKAGE 42
 FIND-RESTART 29
 FIND-SYMBOL 43
 FINISH-OUTPUT 39
 FIRST 8
 FIXNUM 31
 FLET 17
 FLOAT 4, 31
 FLOAT-DIGITS 6
 FLOAT-PRECISION 6
 FLOAT-RADIX 6
 FLOAT-SIGN 4
 FLOATING-POINT-INEXACT 31
 FLOATING-POINT-INVALID-OPERATION 31
 FLOATING-POINT-OVERFLOW 31
 FLOATING-POINT-UNDERFLOW 31
 FLOOR 3
 FLOOR 4
 FMAKUNBOUND 18
 FOR 21
 FORCE-OUTPUT 39
 FORMAT 36
 FORMATTER 36
 FOURTH 8
 FRESH-LINE 34
 FROM 21
 FROUND 4
 FTRUNCATE 4
 FTYPE 46
 FUNCALL 17
 FUNCTION 17, 31, 34, 44
 FUNCTION-KEYWORDS 26
- FUNCTION-LAMBDA-EXPRESSION 18
 FUNCTIONP 15
- GCD 3
 GENERIC-FUNCTION 31
 GENSYM 43
 GENTEMP 43
 GET 16
 GET-DECODED-TIME 47
 GET-DISPATCH-MACRO-CHARACTER 33
 GET-INTERNAL-REAL-TIME 47
 GET-INTERNAL-RUN-TIME 47
 GET-MACRO-CHARACTER 33
 GET-OUTPUT-STREAM-STRING 39
 GET-PROPERTIES 16
 GET-SETF-EXPANSION 19
 GET-UNIVERSAL-TIME 47
 GETF 16
 GETHASH 14
 GO 20
 GRAPHIC-CHAR-P 6
- HANDLER-BIND 28
 HANDLER-CASE 28
 HASH-KEY 21, 23
 HASH-KEYS 21
 HASH-TABLE 31
 HASH-TABLE-COUNT 14
 HASH-TABLE-P 14
 HASH-TABLE-REHASH-SIZE 14
 HASH-TABLE-REHASH-THRESHOLD 14
 HASH-TABLE-SIZE 14
 HASH-TABLE-TEST 14
 HASH-VALUE 21, 23
 HASH-VALUES 23
 HOST-NAMESTRING 41
- IDENTITY 17
 IF 19, 23
 IGNORABLE 46
 IGNORE 46
 IGNORE-ERRORS 28
 IMAGPART 4
 IMPORT 43
 IN-PACKAGE 42
 INCF 3
 INITIALIZE-INSTANCE 24
 INITIALLY 23
 INLINE 47
 INPUT-STREAM-P 32
 INSPECT 46
 INTEGER 31
 INTEGER-DECODE-FLOAT 6
 INTEGER-LENGTH 5
 INTEGERP 3
 INTERACTIVE-STREAM-P 32
 INTERN 43
 INTERNAL-TIME-UNITS-PER-SECOND 47
 INTERSECTION 10
 INTO 23
 INVALID-METHOD-ERROR 26
 INVOKE-DEBUGGER 28
 INVOKE-RESTART 29
 INVOKE-RESTART-INTERACTIVELY 29
 ISQRT 3
 IT 23
- KEYWORD 31, 42, 44
 KEYWORDP 42
- LABELS 17
 LAMBDA 17
 LAMBDA-LIST-KEYWORDS 19
 LAMBDA-PARAMETERS-LIMIT 18
 LAST 8
 LCM 3
 LDB 5
 LDB-TEST 5
 LDIFF 9
 LEAST-NEGATIVE-DOUBLE-FLOAT 6
 LEAST-NEGATIVE-LONG-FLOAT 6
 LEAST-NEGATIVE-NORMALIZED-DOUBLE-FLOAT 6
 LEAST-NEGATIVE-NORMALIZED-LONG-FLOAT 6
- LEAST-NEGATIVE-NORMALIZED-SHORT-FLOAT 6
 LEAST-NEGATIVE-NORMALIZED-SINGLE-FLOAT 6
 LEAST-NEGATIVE-SHORT-FLOAT 6
 LEAST-NEGATIVE-SINGLE-FLOAT 6
 LEAST-POSITIVE-DOUBLE-FLOAT 6
 LEAST-POSITIVE-LONG-FLOAT 6
 LEAST-POSITIVE-NORMALIZED-DOUBLE-FLOAT 6
 LEAST-POSITIVE-NORMALIZED-LONG-FLOAT 6
 LEAST-POSITIVE-NORMALIZED-SHORT-FLOAT 6
 LEAST-POSITIVE-SHORT-FLOAT 6
 LEAST-POSITIVE-SINGLE-FLOAT 6
- LENGTH 12
 LET 16
 LET* 16
 LISP-IMPLEMENTATION-TYPE 47
 LISP-IMPLEMENTATION-VERSION 47
 LIST 8, 26, 31
 LIST-ALL-PACKAGES 42
 LIST-LENGTH 8
 LIST* 8
 LISTEN 39
 LISTP 8
 LOAD 44
 LOAD-LOGICAL-PATHNAME-TRANSLATIONS 41
 LOAD-TIME-VALUE 45
 LOCALLY 45
 LOG 3
 LOGAND 5
 LOGANDC1 5
 LOGANDC2 5
 LOGBITP 5
 LOGCOUNT 5
 LOGEQV 5
 LOGICAL-PATHNAME 31, 41
 LOGICAL-PATHNAME-TRANSLATIONS 41
 LOGIOR 5
 LOGNAND 5
 LOGNOR 5
 LOGNOT 5
 LOGORC1 5
 LOGORC2 5
 LOGTEST 5
 LOGXOR 5
 LONG-FLOAT 31, 34
 LONG-FLOAT-FLOAT-EPSILON 6
 LONG-FLOAT-NEGATIVE-EPSILON 6
 LONG-SITE-NAME 47
 LOOP 21
 LOOP-FINISH 23
 LOWER-CASE-P 6
- MACHINE-INSTANCE 47
 MACHINE-TYPE 47
 MACHINE-VERSION 47
 MACRO-FUNCTION 45
 MACROEXPAND 46
 MACROEXPAND-1 46
 MACROLET 18
 MAKE-ARRAY 10
 MAKE-BROADCAST-STREAM 39
 MAKE-CONCATENATED-STREAM 39
 MAKE-CONDITION 28
 MAKE-DISPATCH-MACRO-CHARACTER 33
 MAKE-ECHO-STREAM 39
 MAKE-HASH-TABLE 14
 MAKE-INSTANCE 24
 MAKE-INSTANCES-OBSOLETE 24
 MAKE-LIST 8
 MAKE-LOAD-FORM 45
 MAKE-LOAD-FORM-SAVING-SLOTS 45
 MAKE-METHOD 27
 MAKE-PACKAGE 42
 MAKE-PATHNAME 40
 MAKE-RANDOM-STATE 4
 MAKE-SEQUENCE 12
 MAKE-STRING 7
- MAKE-STRING-INPUT-STREAM 39
 MAKE-STRING-OUTPUT-STREAM 39
 MAKE-SYMBOL 43
 MAKE-SYNONYM-STREAM 39
 MAKE-TWO-WAY-STREAM 39
 MAKUNBOUND 16
 MAP 14
 MAP-INTO 14
 MAPC 9
 MAPCAN 9
 MAPCAR 9
 MAPCON 9
 MAPHASH 14
 MAPL 9
 MAPLIST 9
 MASK-FIELD 5
 MAX 4, 26
 MAXIMIZE 23
 MAXIMIZING 23
 MEMBER 8, 30
 MEMBER-IF 8
 MEMBER-IF-NOT 8
 MERGE 12
 MERGE-PATHNAMES 41
 METHOD 31
 METHOD-COMBINATION 31, 44
 METHOD-COMBINATION-ERROR 26
 METHOD-QUALIFIERS 26
 MIN 4, 26
 MINIMIZE 23
 MINIMIZING 23
 MINUSP 3
 MISMATCH 12
 MOD 4, 30
 MOST-NEGATIVE-DOUBLE-FLOAT 6
 MOST-NEGATIVE-FIXNUM 6
 MOST-NEGATIVE-LONG-FLOAT 6
 MOST-NEGATIVE-SHORT-FLOAT 6
 MOST-NEGATIVE-SINGLE-FLOAT 6
 MOST-POSITIVE-DOUBLE-FLOAT 6
 MOST-POSITIVE-FIXNUM 6
 MOST-POSITIVE-LONG-FLOAT 6
 MOST-POSITIVE-SHORT-FLOAT 6
 MOST-POSITIVE-SINGLE-FLOAT 6
 MUFFLE-WARNING 29
 MULTIPLE-VALUE-BIND 16
 MULTIPLE-VALUE-CALL 17
 MULTIPLE-VALUE-LIST 17
 MULTIPLE-VALUE-PROG1 20
 MULTIPLE-VALUE-SETQ 16
 MULTIPLE-VALUES-LIMIT 18
- NAME-CHAR 7
 NAMED 21
 NAMESTRING 41
 NBUFLAST 9
 NCONC 9, 23, 26
 NCONCING 23
 NEVER 23
 NEWLINE 6
 NEXT-METHOD-P 25
 NIL 2, 44
 NINTERSECTION 10
 NINTH 8
 NO-APPLICABLE-METHOD 26
 NO-NEXT-METHOD 26
 NOT 15, 30, 34
 NOTANY 12
 NOTEVERY 12
 NOTINLINE 47
 NRECONC 9
 NREVERSE 12
 NSET-DIFFERENCE 10
 NSET-EXCLUSIVE-OR 10
 NSTRING-CAPITALIZE 7
 NSTRING-DOWNCASE 7
 NSTRING-UPCASE 7
 NSUBLIS 10
 NSUBST 10
 NSUBST-IF 10
 NSUBST-IF-NOT 10
 NSUBSTITUTE 13
 NSUBSTITUTE-IF 13
 NSUBSTITUTE-IF-NOT 13
 NTH 8
 NTH-VALUE 17

- NTHCDR 8
 NULL 8, 31
 NUMBER 31
 NUMBERP 3
 NUMERATOR 4
 NUNION 10
- ODDP 3
 OF 21, 23
 OF-TYPE 21
 ON 21
 OPEN 39
 OPEN-STREAM-P 32
 OPTIMIZE 47
 OR 20, 26, 30, 34
 OTHERWISE 20, 30
 OUTPUT-STREAM-P 32
- PACKAGE 31
 PACKAGE-ERROR 31
 PACKAGE-ERROR-PACKAGE 29
 PACKAGE-NAME 42
 PACKAGE-NICKNAMES 42
 PACKAGE-SHADOWING-SYMBOLS 43
 PACKAGE-USE-LIST 42
 PACKAGE-USED-BY-LIST 42
 PACKAGEP 42
 PAIRLIS 9
 PARSE-ERROR 31
 PARSE-INTEGER 8
 PARSE-NAMESTRING 41
 PATHNAME 31, 41
 PATHNAME-DEVICE 40
 PATHNAME-DIRECTORY 40
 PATHNAME-HOST 40
 PATHNAME-MATCH-P 32
 PATHNAME-NAME 40
 PATHNAME-TYPE 40
 PATHNAME-VERSION 40
 PATHNAMEP 32
 PEEK-CHAR 32
 PHASE 4
 PI 3
 PLUSP 3
 POP 9
 POSITION 13
 POSITION-IF 13
 POSITION-IF-NOT 13
 PPRINT 34
 PPRINT-DISPATCH 36
 PPRINT-EXIT-IF-LIST-EXHAUSTED 35
 PPRINT-FILL 35
 PPRINT-INDENT 35
 PPRINT-LINEAR 35
 PPRINT-LOGICAL-BLOCK 35
 PPRINT-NEWLINE 36
 PPRINT-POP 35
 PPRINT-TAB 35
 PPRINT-TABULAR 35
 PRESENT-SYMBOL 23
 PRESENT-SYMBOLS 23
 PRIN1 34
 PRIN1-TO-STRING 34
 PRINC 34
 PRINC-TO-STRING 34
 PRINT 34
 PRINT-NOT-READABLE 31
 PRINT-NOT-READABLE-OBJECT 29
 PRINT-OBJECT 34
 PRINT-UNREADABLE-OBJECT 34
 PROBE-FILE 41
 PROCLAIM 46
 PROG 20
 PROG1 20
 PROG2 20
 PROG* 20
 PROG# 20
 PROG# 20, 26
 PROGRAM-ERROR 31
 PROG# 16
 PROVIDE 43
 PSETF 16
 PSETQ 16
 PUSH 9
 PUSHNEW 9
- QUOTE 33, 45
- RANDOM 4
 RANDOM-STATE 31
 RANDOM-STATE-P 3
 RASSOC 9
 RASSOC-IF 9
 RASSOC-IF-NOT 9
 RATIO 31, 34
 RATIONAL 4, 31
 RATIONALIZE 4
 RATIONALP 3
 READ 32
 READ-BYTE 32
 READ-CHAR 32
- READ-CHAR-NO-HANG 32
 READ-DELIMITED-LIST 32
 READ-FROM-STRING 32
 READ-LINE 32
 READ-PRESERVING-WHITESPACE 32
 READ-SEQUENCE 33
 READER-ERROR 31
 READTABLE 31
 READTABLE-CASE 33
 READTABLEP 32
 REAL 31
 REALP 3
 REALPART 4
 REDUCE 14
 REINITIALIZE-INSTANCE 24
 REM 4
 REMF 16
 REMHASH 14
 REMOVE 13
 REMOVE-DUPLICATES 13
 REMOVE-IF 13
 REMOVE-IF-NOT 13
 REMOVE-METHOD 26
 REMPROP 16
 RENAME-FILE 41
 RENAME-PACKAGE 42
 REPEAT 23
 REPLACE 14
 REQUIRE 43
 REST 8
 RESTART 31
 RESTART-BIND 29
 RESTART-CASE 28
 RESTART-NAME 29
 RETURN 20, 23
 RETURN-FROM 20
 REVAPPEND 9
 REVERSE 12
 ROOM 46
 ROTATEF 16
 ROUND 4
 ROW-MAJOR-AREF 10
 RPLACA 9
 RPLACD 9
- SAFETY 47
 SATISFIES 30
 SBIT 11
 SCALE-FLOAT 6
 SCHAR 8
 SEARCH 13
 SECOND 8
 SEQUENCE 31
 SERIOUS-CONDITION 31
 SET 16
 SET-DIFFERENCE 10
 SET-DISPATCH-MACRO-CHARACTER 33
 SET-EXCLUSIVE-OR 10
 SET-MACRO-CHARACTER 33
 SET-PPRINT-DISPATCH 36
 SET-SYNTAX-FROM-CHAR 33
 SETF 16, 44
 SETQ 16
 SEVENTH 8
 SHADOW 43
 SHADOWING-IMPORT 43
 SHARED-INITIALIZE 25
 SHIFTF 16
 SHORT-FLOAT 31, 34
 SHORT-FLOAT-EPSILON 6
 SHORT-FLOAT-NEGATIVE-EPSILON 6
 SHORT-SITE-NAME 47
 SIGNAL 28
 SIGNED-BYTE 31
 SIGNUM 4
 SIMPLE-ARRAY 31
 SIMPLE-BASE-STRING 31
 SIMPLE-BIT-VECTOR 31
 SIMPLE-BIT-VECTOR-P 10
 SIMPLE-CONDITION 31
 SIMPLE-CONDITION-FORMAT-ARGUMENTS 29
 SIMPLE-CONDITION-FORMAT-CONTROL 29
 SIMPLE-ERROR 31
 SIMPLE-STRING 31
 SIMPLE-STRING-P 7
 SIMPLE-TYPE-ERROR 31
 SIMPLE-VECTOR 31
 SIMPLE-VECTOR-P 10
 SIMPLE-WARNING 31
 SIN 3
 SINGLE-FLOAT 31, 34
 SINGLE-FLOAT-EPSILON 6
- SINGLE-FLOAT-NEGATIVE-EPSILON 6
 SINH 3
 SIXTH 8
 SLEEP 20
 SLOT-BOUND#P 24
 SLOT-EXISTS-P 24
 SLOT-MAKUNBOUND 24
 SLOT-MISSING 25
 SLOT-UNBOUND 25
 SLOT-VALUE 24
 SOFTWARE-TYPE 47
 SOFTWARE-VERSION 47
 SOME 12
 SORT 12
 SPACE 6, 47
 SPECIAL 47
 SPECIAL-OPERATOR-P 44
 SPEED 47
 SQRT 3
 STABLE-SORT 12
 STANDARD 26
 STANDARD-CHAR 6, 31
 STANDARD-CHAR-P 6
 STANDARD-CLASS 31
 STANDARD-GENERIC-FUNCTION 31
 STANDARD-METHOD 31
 STANDARD-OBJECT 31
 STEP 46
 STORAGE-CONDITION 31
 STORE-VALUE 29
 STREAM 31
 STREAM-ELEMENT-TYPE 30
 STREAM-ERROR 31
 STREAM-ERROR-STREAM 29
 STREAM-EXTERNAL-FORMAT 40
 STREAMP 32
 STRING 7, 31
 STRING-CAPITALIZE 7
 STRING-DOWNCASE 7
 STRING-EQUAL 7
 STRING-GREATERP 7
 STRING-LEFT-TRIM 7
 STRING-LESSP 7
 STRING-NOT-EQUAL 7
 STRING-NOT-GREATERP 7
 STRING-NOT-LESSP 7
 STRING-RIGHT-TRIM 7
 STRING-STREAM 31
 STRING-TRIM 7
 STRING-UPCASE 7
 STRING/= 7
 STRING< 7
 STRING<= 7
 STRING= 7
 STRING> 7
 STRING>= 7
 STRINGP 7
 STRUCTURE 44
 STRUCTURE-CLASS 31
 STRUCTURE-OBJECT 31
 STYLE-WARNING 31
 SUBLIS 10
 SUBSEQ 12
 SUBSETP 8
 SUBST 10
 SUBST-IF 10
 SUBST-IF-NOT 10
 SUBSTITUTE 13
 SUBSTITUTE-IF 13
 SUBSTITUTE-IF-NOT 13
 SUBTYPEP 30
 SUM 23
 SUMMING 23
 SVREF 11
 SXHASH 14
 SYMBOL 23, 31, 43
 SYMBOL-FUNCTION 44
 SYMBOL-MACROLET 18
 SYMBOL-NAME 44
 SYMBOL-PACKAGE 44
 SYMBOL-PLIST 44
 SYMBOL-VALUE 44
 SYMBOLP 42
 SYMBOLS 23
 SYNONYM-STREAM 31
 SYNONYM-STREAM-SYMBOL 39
- T 2, 31, 44
 TAGBODY 20
 TAILP 8
 TAN 3
 TANH 3
 TENTH 8
 TERPRI 34
 THE 21, 30
 THEN 21
 THEREIS 23
 THIRD 8
- THROW 20
 TIME 46
 TO 21
 TRACE 46
 TRANSLATE-LOGICAL-PATHNAME 41
 TRANSLATE-PATHNAME 41
 TREE-EQUAL 10
 TRUENAME 41
 TRUNCATE 4
 TWO-WAY-STREAM 31
 TWO-WAY-STREAM-INPUT-STREAM 39
 TWO-WAY-STREAM-OUTPUT-STREAM 39
 TYPE 44, 46
 TYPE-ERROR 31
 TYPE-ERROR-DATUM 29
 TYPE-ERROR-EXPECTED-TYPE 29
 TYPE-OF 30
 TYPECASE 30
 TYPEP 30
- UNBOUND-SLOT 31
 UNBOUND-SLOT-INSTANCE 29
 UNBOUND-VARIABLE 31
 UNDEFINED-FUNCTION 31
 UNEXPORT 43
 UNINTERN 43
 UNION 10
 UNLESS 19, 23
 UNREAD-CHAR 32
 UNSIGNED-BYTE 31
 UNTIL 23
 UNTRACE 46
 UNUSE-PACKAGE 42
 UNWIND-PROTECT 20
 UPDATE-INSTANCE-FOR-DIFFERENT-CLASS 24
 UPDATE-INSTANCE-FOR-REDEFINED-CLASS 25
 UPFROM 21
 UPGRADED-ARRAY-ELEMENT-TYPE 30
 UPGRADED-COMPLEX-PART-TYPE 6
 UPPER-CASE-P 6
 UPTO 21
 USE-PACKAGE 42
 USE-VALUE 29
 USER-HOMEDIR-PATHNAME 41
 USING 21, 23
- V 38
 VALUES 17, 30
 VALUES-LIST 17
 VARIABLE 44
 VECTOR 11, 31
 VECTOR-POP 11
 VECTOR-PUSH 11
 VECTOR-PUSH-EXTEND 11
 VECTORP 10
- WARN 28
 WARNING 31
 WHEN 19, 23
 WHILE 23
 WILD-PATHNAME-P 32
 WITH 21
 WITH-ACCESSORS 24
 WITH-COMPILED-UNIT 45
 WITH-CONDITION-RESTARTS 29
 WITH-HASH-TABLE-ITERATOR 14
 WITH-INPUT-FROM-STRING 40
 WITH-OPEN-FILE 40
 WITH-OPEN-STREAM 40
 WITH-OUTPUT-TO-STRING 40
 WITH-PACKAGE-ITERATOR 43
 WITH-SIMPLE-RESTART 28
 WITH-SLOTS 24
 WITH-STANDARD-IO-SYNTAX 32
 WRITE 35
 WRITE-BYTE 35
 WRITE-CHAR 35
 WRITE-LINE 35
 WRITE-SEQUENCE 35
 WRITE-STRING 35
 WRITE-TO-STRING 35
- Y-OR-N-P 32
 YES-OR-NO-P 32
- ZEROP 3

