

Quick Reference

cl

Common

lisp

Bert Burgemeister

Contents

1	Numbers	3	9.5	Control Flow	21
1.1	Predicates	3	9.6	Iteration	22
1.2	Numeric Functns .	3	9.7	Loop Facility	22
1.3	Logic Functions .	5	10	CLOS	25
1.4	Integer Functions .	6	10.1	Classes	25
1.5	Implementation- Dependent	6	10.2	Generic Functns .	26
2	Characters	7	10.3	Method Combi- nation Types	28
3	Strings	8	11	Conditions and Errors	28
4	Conses	8	12	Types and Classes	31
4.1	Predicates	8	13	Input/Output	33
4.2	Lists	9	13.1	Predicates	33
4.3	Association Lists .	10	13.2	Reader	33
4.4	Trees	10	13.3	Character Syntax .	35
4.5	Sets	11	13.4	Printer	36
5	Arrays	11	13.5	Format	38
5.1	Predicates	11	13.6	Streams	40
5.2	Array Functions .	11	13.7	Paths and Files . .	42
5.3	Vector Functions .	12	14	Packages and Symbols	43
6	Sequences	12	14.1	Predicates	43
6.1	Seq. Predicates . .	12	14.2	Packages	44
6.2	Seq. Functions . .	13	14.3	Symbols	45
7	Hash Tables	15	14.4	Std Packages	46
8	Structures	16	15	Compiler	46
9	Control Structure	16	15.1	Predicates	46
9.1	Predicates	16	15.2	Compilation	46
9.2	Variables	17	15.3	REPL & Debug . . .	47
9.3	Functions	18	15.4	Declarations	48
9.4	Macros	19	16	External Environment	49

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 21.

$\frac{\textit{foo}; \textit{bar}; \textit{baz}}{2 \quad n}$ ▷ 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*_□^{*})

(*f** *a*_□^{*})

▷ Return $\sum a$ or $\prod a$, respectively.

(*f*- *a* *b*^{*})

(*f*/ *a* *b*^{*})

▷ Return $a - \sum b$ or $a / \prod b$, respectively. Without any *bs*, return $-a$ or $1/a$, respectively.

(*f*1+ *a*)

(*f*1- *a*)

▷ Return $a + 1$ or $a - 1$, respectively.

($\left. \begin{matrix} m \text{incf} \\ m \text{decf} \end{matrix} \right\} \widetilde{\text{place}} [delta a_{\square}])$

▷ Increment or decrement the value of *place* by *delta*. Return new value.

(*f*exp *p*)

(*f*expt *b* *p*)

▷ Return e^p or b^p , respectively.

(*f*log *a* [*b*_□])

▷ Return $\log_b a$ or, without *b*, $\ln a$.

(*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*)

(*f*tan *a*)

▷ $\sin a$, $\cos a$, or $\tan a$, respectively. (*a* in radians.)

(*f*asin *a*)

(*f*acos *a*)

▷ $\arcsin a$ or $\arccos a$, respectively, in radians.

(*f*atan *a* [*b*_□])

▷ $\arctan \frac{a}{b}$ in radians.

(*f***sinh** *a*)
(*f***cosh** *a*) ▷ sinh *a*, cosh *a*, or tanh *a*, 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*.

$\left(\begin{array}{l} \{ \textit{fround} | \textit{fround} \} \\ \{ \textit{ffloor} | \textit{ffloor} \} \\ \{ \textit{fceil} | \textit{fceil} \} \\ \{ \textit{ftruncate} | \textit{ftruncate} \} \end{array} \right) n [d_{\square}]$
▷ Return as **integer** or **float**, respectively, n/d rounded, or rounded towards $-\infty$, $+\infty$, or 0, respectively; and remainder.

$\left(\begin{array}{l} \{ \textit{fmod} \} \\ \{ \textit{frem} \} \end{array} \right) n d$
▷ 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] [NIL]])
▷ 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* \square]) ▷ *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* \square]) ▷ 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* 0.0f0])
▷ 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

cboole-1	▷ <u>int-a</u> .
cboole-2	▷ <u>int-b</u> .
cboole-c1	▷ <u>¬int-a</u> .
cboole-c2	▷ <u>¬int-b</u> .
cboole-set	▷ <u>All bits set</u> .
cboole-clr	▷ <u>All bits zero</u> .
cboole-eqv	▷ <u>int-a ≡ int-b</u> .
cboole-and	▷ <u>int-a ∧ int-b</u> .
cboole-andc1	▷ <u>¬int-a ∧ int-b</u> .
cboole-andc2	▷ <u>int-a ∧ ¬int-b</u> .
cboole-nand	▷ <u>¬(int-a ∧ int-b)</u> .
cboole-ior	▷ <u>int-a ∨ int-b</u> .
cboole-orc1	▷ <u>¬int-a ∨ int-b</u> .
cboole-orc2	▷ <u>int-a ∨ ¬int-b</u> .
cboole-xor	▷ <u>¬(int-a ≡ int-b)</u> .
cboole-nor	▷ <u>¬(int-a ∨ int-b)</u> .

(*f*lognot integer) ▷ ¬integer.

(*f*logeqv integer*)

(*f*logand integer*)

▷ Return value of exclusive-nored or anded integers, respectively. Without any integer, return -1.

(*f*logandc1 int-a int-b) ▷ ¬int-a ∧ int-b.

(*f*logandc2 int-a int-b) ▷ int-a ∧ ¬int-b.

(*f*lognand int-a int-b) ▷ ¬(int-a ∧ int-b).

(*f*logxor integer*)

(*f*logior integer*)

▷ Return value of exclusive-ored or ored integers, respectively. Without any integer, return 0.

(*f*logorc1 int-a int-b) ▷ ¬int-a ∨ int-b.

(*f*logorc2 int-a int-b) ▷ int-a ∨ ¬int-b.

(*f*lognor int-a int-b) ▷ ¬(int-a ∨ int-b).

(*f*logbitp i int) ▷ T if zero-indexed *i*th bit of *int* is set.

(*f*logtest int-a int-b)

▷ Return T if there is any bit set in *int-a* which is set in *int-b* as well.

(*f*logcount int)

▷ Number of 1 bits in *int* ≥ 0, number of 0 bits in *int* < 0.

1.4 Integer Functions

(*f***integer-length** *integer*)

▷ Number of bits necessary to represent *integer*.

(*f***ldb-test** *byte-spec integer*)

▷ Return T if any bit specified by *byte-spec* in *integer* is set.

(*f***ash** *integer count*)

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

(*f***ldb** *byte-spec integer*)

▷ Extract byte denoted by *byte-spec* from *integer*. **setfable**.

($\left. \begin{array}{l} \text{fdeposit-field} \\ \text{fdpb} \end{array} \right\} \textit{int-a byte-spec int-b}$)

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

(*f***mask-field** *byte-spec integer*)

▷ Return copy of *integer* with all bits unset but those denoted by *byte-spec*. **setfable**.

(*f***byte** *size position*)

▷ Byte specifier for a byte of *size* bits starting at a weight of $2^{\textit{position}}$.

(*f***byte-size** *byte-spec*)

(*f***byte-position** *byte-spec*)

▷ Size or position, respectively, of *byte-spec*.

1.5 Implementation-Dependent

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

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

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

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

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

▷ 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*_{NTD}])

▷ 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 11 and 12.

(*fstringp* *foo*)
(*fsimple-string-p* *foo*) ▷ T if *foo* is of indicated type.

$\left(\begin{array}{l} \text{fstring=} \\ \text{fstring-equal} \end{array} \right) \text{foo bar} \left\{ \begin{array}{l} \text{:start1 } \text{start-foo}_{\boxed{0}} \\ \text{:start2 } \text{start-bar}_{\boxed{0}} \\ \text{:end1 } \text{end-foo}_{\boxed{\text{NIL}}} \\ \text{:end2 } \text{end-bar}_{\boxed{\text{NIL}}} \end{array} \right\}$

▷ Return T if subsequences of *foo* and *bar* are equal. Obey/ignore, respectively, case.

$\left(\begin{array}{l} \text{fstring}\{/= \mid \text{-not-equal}\} \\ \text{fstring}\{> \mid \text{-greaterp}\} \\ \text{fstring}\{>= \mid \text{-not-lessp}\} \\ \text{fstring}\{< \mid \text{-lessp}\} \\ \text{fstring}\{<= \mid \text{-not-greaterp}\} \end{array} \right) \text{foo bar} \left\{ \begin{array}{l} \text{:start1 } \text{start-foo}_{\boxed{0}} \\ \text{:start2 } \text{start-bar}_{\boxed{0}} \\ \text{:end1 } \text{end-foo}_{\boxed{\text{NIL}}} \\ \text{:end2 } \text{end-bar}_{\boxed{\text{NIL}}} \end{array} \right\}$

▷ 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.

(*fmake-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*.

(*fstring* *x*)
 $\left(\begin{array}{l} \text{fstring-capitalize} \\ \text{fstring-upcase} \\ \text{fstring-downcase} \end{array} \right) x \left\{ \begin{array}{l} \text{:start } \text{start}_{\boxed{0}} \\ \text{:end } \text{end}_{\boxed{\text{NIL}}} \end{array} \right\}$

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

$\left(\begin{array}{l} \text{fnstring-capitalize} \\ \text{fnstring-upcase} \\ \text{fnstring-downcase} \end{array} \right) \widetilde{\text{string}} \left\{ \begin{array}{l} \text{:start } \text{start}_{\boxed{0}} \\ \text{:end } \text{end}_{\boxed{\text{NIL}}} \end{array} \right\}$

▷ Convert *string* into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

$\left(\begin{array}{l} \text{fstring-trim} \\ \text{fstring-left-trim} \\ \text{fstring-right-trim} \end{array} \right) \text{char-bag } \text{string}$

▷ Return string with all characters in sequence *char-bag* removed from both ends, from the beginning, or from the end, respectively.

(*fchar* *string* *i*)
(*fchar* *string* *i*)

▷ Return zero-indexed *i*th character of string ignoring/obeying, respectively, fill pointer. **setfable**.

(*fparse-integer* *string* $\left\{ \begin{array}{l} \text{:start } \text{start}_{\boxed{0}} \\ \text{:end } \text{end}_{\boxed{\text{NIL}}} \\ \text{:radix } \text{int}_{\boxed{10}} \\ \text{:junk-allowed } \text{bool}_{\boxed{\text{NIL}}} \end{array} \right\}$)

▷ Return integer parsed from *string* and index of parse end.

4 Conses

4.1 Predicates

(*fconsp* *foo*)
(*flistp* *foo*) ▷ Return T if *foo* is of indicated type.

(*fendp* *list*)
(*fnull* *foo*) ▷ Return T if *list/foo* is NIL.

- (**atom** *foo*) ▷ Return T if *foo* is not a **cons**.
- (**tailp** *foo list*) ▷ Return T if *foo* is a tail of *list*.
- (**member** *foo list* $\left\{ \begin{array}{l} \text{:test function} \neq \text{eq} \\ \text{:test-not function} \\ \text{:key function} \end{array} \right\}$)
 ▷ Return tail of *list* starting with its first element matching *foo*. Return NIL if there is no such element.
- ($\left\{ \begin{array}{l} \text{fmember-if} \\ \text{fmember-if-not} \end{array} \right\}$ *test list* [:key function])
 ▷ Return tail of *list* starting with its first element satisfying *test*. Return NIL if there is no such element.
- (**subsetp** *list-a list-b* $\left\{ \begin{array}{l} \text{:test function} \neq \text{eq} \\ \text{:test-not function} \\ \text{:key function} \end{array} \right\}$)
 ▷ Return T if *list-a* is a subset of *list-b*.

4.2 Lists

- (**cons** *foo bar*) ▷ Return new cons (*foo . bar*).
- (**list** *foo**) ▷ Return list of foos.
- (**list*** *foo+*)
 ▷ Return list of foos with last *foo* becoming cdr of last cons. Return foo if only one *foo* given.
- (**make-list** *num* [:initial-element *foo* NIL])
 ▷ New list with *num* elements set to *foo*.
- (**list-length** *list*) ▷ Length of *list*; NIL for circular *list*.
- (**car** *list*) ▷ Car of *list* or NIL if *list* is NIL. **setfable**.
- (**cdr** *list*)
 (**rest** *list*) ▷ Cdr of *list* or NIL if *list* is NIL. **setfable**.
- (**nthcdr** *n list*) ▷ Return tail of list after calling **f****cdr** *n* times.
- ($\left\{ \text{ffirst} \mid \text{fsecond} \mid \text{fthird} \mid \text{fourth} \mid \text{fifth} \mid \text{fsixth} \mid \dots \mid \text{fninth} \mid \text{ftenth} \right\}$ *list*)
 ▷ Return nth element of list if any, or NIL otherwise. **setfable**.
- (**nth** *n list*) ▷ Zero-indexed nth element of *list*. **setfable**.
- (**cXr** *list*)
 ▷ With *X* being one to four **as** and **ds** representing **f****cars** and **f****cdrs**, e.g. (**f****cadr** *bar*) is equivalent to (**f****car** (**f****cdr** *bar*)). **setfable**.
- (**last** *list* [*num* 1]) ▷ Return list of last num conses of *list*.
- ($\left\{ \begin{array}{l} \text{fbutlast } list \\ \text{fnbutlast } \widetilde{list} \end{array} \right\}$ [*num* 1]) ▷ list excluding last *num* conses.
- ($\left\{ \begin{array}{l} \text{frplaca} \\ \text{frplacd} \end{array} \right\}$ \widetilde{cons} *object*)
 ▷ Replace car, or cdr, respectively, of cons with *object*.
- (**ldiff** *list foo*)
 ▷ If *foo* is a tail of *list*, return preceding part of list. Otherwise return list.
- (**adjoin** *foo list* $\left\{ \begin{array}{l} \text{:test function} \neq \text{eq} \\ \text{:test-not function} \\ \text{:key function} \end{array} \right\}$)
 ▷ Return list if *foo* is already member of *list*. If not, return (**f****cons** *foo list*).
- (**pop** \widetilde{place})
 ▷ Set *place* to (**f****cdr** *place*), return (**f****car** *place*).

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

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

(**fappend** [*proper-list** *foo* $\underline{\text{NIL}}$])

(**fconc** [*non-circular-list** *foo* $\underline{\text{NIL}}$])

▷ Return concatenated list or, with only one argument, *foo*. *foo* can be of any type.

(**frevappend** *list* *foo*)

(**fpreconc** \widetilde{list} *foo*)

▷ Return concatenated list after reversing order in *list*.

$\left(\begin{array}{l} \text{fmapcar} \\ \text{fmaplist} \end{array} \right) function\ list^+$

▷ Return list of return values of *function* successively invoked with corresponding arguments, either cars or cdrs, respectively, from each *list*.

$\left(\begin{array}{l} \text{fmapcan} \\ \text{fmapcon} \end{array} \right) function\ \widetilde{list}^+$

▷ 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.

$\left(\begin{array}{l} \text{fmapc} \\ \text{fmapl} \end{array} \right) function\ list^+$

▷ Return first list after successively applying *function* to corresponding arguments, either cars or cdrs, respectively, from each *list*. *function* should have some side effects.

(**fcopy-list** *list*) ▷ Return copy of *list* with shared elements.

4.3 Association Lists

(**fpairlis** *keys values* [*alist* $\underline{\text{NIL}}$])

▷ Prepend to *alist* an association list made from lists *keys* and *values*.

(**facons** *key value alist*)

▷ Return *alist* with a (*key* . *value*) pair added.

$\left(\begin{array}{l} \text{fassoc} \\ \text{fassoc} \end{array} \right) foo\ alist\ \left\{ \begin{array}{l} \text{:test } test_{\neq} \\ \text{:test-not } test \\ \text{:key } function \end{array} \right\}$

$\left(\begin{array}{l} \text{fassoc-if[-not]} \\ \text{fassoc-if[-not]} \end{array} \right) test\ alist\ [\text{:key } function]$

▷ First cons whose car, or cdr, respectively, satisfies *test*.

(**fcopy-alist** *alist*) ▷ Return copy of *alist*.

4.4 Trees

(**ftree-equal** *foo bar* $\left\{ \begin{array}{l} \text{:test } test_{\neq} \\ \text{:test-not } test \end{array} \right\}$)

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

$\left(\begin{array}{l} \text{fsubst} \\ \text{fnsubst} \end{array} \right) new\ old\ tree\ \left\{ \begin{array}{l} \text{:test } function_{\neq} \\ \text{:test-not } function \\ \text{:key } function \end{array} \right\}$

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

$\left(\begin{array}{l} \text{fsubst-if[-not]} \\ \text{fnsubst-if[-not]} \end{array} \right) new\ test\ tree\ [\text{:key } function]$

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

$$\left(\begin{array}{l} \text{fsublis } \textit{association-list tree} \\ \text{fnsublis } \textit{association-list tree} \end{array} \right) \left\{ \begin{array}{l} \text{:test function } \overline{\text{\#'eq}} \\ \text{:test-not function} \\ \text{:key function} \end{array} \right\}$$

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

(**fcopy-tree** *tree*) ▷ Copy of tree with same shape and leaves.

4.5 Sets

$$\left(\begin{array}{l} \text{fintersection} \\ \text{fset-difference} \\ \text{funion} \\ \text{fset-exclusive-or} \\ \text{fnintersection} \\ \text{fnset-difference} \\ \text{fnunion} \\ \text{fnset-exclusive-or} \end{array} \right) \left\{ \begin{array}{l} a \ b \\ \tilde{a} \ b \\ \tilde{a} \ \tilde{b} \end{array} \right\} \left\{ \begin{array}{l} \text{:test function } \overline{\text{\#'eq}} \\ \text{:test-not function} \\ \text{:key function} \end{array} \right\}$$

▷ Return $a \cap b$, $a \setminus b$, $a \cup b$, or $a \Delta b$, respectively, of lists *a* and *b*.

5 Arrays

5.1 Predicates

(**farrayp** *foo*)

(**fvectorp** *foo*)

(**fsimple-vector-p** *foo*)

▷ T if *foo* is of indicated type.

(**fbit-vector-p** *foo*)

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

(**fadjustable-array-p** *array*)

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

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

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

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

5.2 Array Functions

$$\left(\begin{array}{l} \text{fmake-array } \textit{dimension-sizes} \text{ [:adjustable } \textit{bool} \text{NLL}] \\ \text{fadjust-array } \textit{array} \textit{dimension-sizes} \end{array} \right) \left\{ \begin{array}{l} \text{:element-type } \textit{type} \text{N} \\ \text{:fill-pointer } \{ \textit{num} | \textit{bool} \} \text{NLL} \\ \left\{ \begin{array}{l} \text{:initial-element } \textit{obj} \\ \text{:initial-contents } \textit{tree-or-array} \\ \text{:displaced-to } \textit{array} \text{NLL} \text{ [:displaced-index-offset } \textit{i} \text{N}] \end{array} \right\} \end{array} \right\}$$

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

(**faref** *array* [*subscripts*])

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

(**frow-major-aref** *array* *i*)

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

(**farray-row-major-index** *array* [*subscripts*])

▷ Index in row-major order of the element denoted by *subscripts*.

(**farray-dimensions** *array*)

▷ List containing the lengths of *array*'s dimensions.

(**farray-dimension** *array* *i*)

▷ Length of *i*th dimension of *array*.

(**farray-total-size** *array*)

▷ Number of elements in *array*.

(**farray-rank** *array*)

▷ Number of dimensions of *array*.

(**farray-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.

$\left(\begin{array}{l} \textit{fbit-eqv} \\ \textit{fbit-and} \\ \textit{fbit-andc1} \\ \textit{fbit-andc2} \\ \textit{fbit-nand} \\ \textit{fbit-ior} \\ \textit{fbit-orc1} \\ \textit{fbit-orc2} \\ \textit{fbit-xor} \\ \textit{fbit-nor} \end{array} \right)$	}	$\textit{bit-array-a} \textit{ bit-array-b} \textit{ [result-bit-array}_{\textit{NIL}}\textit{]}$
---	---	---

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

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

carray-dimension-limit

▷ Upper bound of an array dimension; ≥ 1024.

carray-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 ≥ *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

$\left(\begin{array}{l} \textit{fevery} \\ \textit{fnotevery} \end{array} \right) \textit{ test sequence}^+$

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

$\left(\begin{array}{l} \textit{fsome} \\ \textit{fnotany} \end{array} \right) \textit{ test sequence}^+$

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

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

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

6.2 Sequence Functions

(*f* **make-sequence** *sequence-type* *size* [**initial-element** *foo*])

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

(*f* **concatenate** *type* *sequence**)

▷ Return concatenated sequence of *type*.

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

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

(*f* **fill** $\widetilde{\text{sequence}}$ *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*.

(*f* **length** *sequence*)

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

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

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

$\left\{ \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \end{array} \right\}$ *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 number of elements in *sequence* which satisfy *test*.

(*f* **elt** *sequence* *index*)

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

(*f* **subseq** *sequence* *start* [*end* NIL])

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

$\left\{ \begin{array}{l} \text{:from-end } \text{bool}_{\text{NIL}} \\ \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \end{array} \right\}$ $\widetilde{\text{sequence}}$ *test* [**key function**])

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

(*f* **reverse** *sequence*)

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

▷ Return sequence in reverse order.

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

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

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

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

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

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

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

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

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

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

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

▷ Make copy of *sequence* without duplicates.

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

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

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

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

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

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

$$(\text{map } \text{type } \text{function } \text{sequence}^+)$$

▷ 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* $\left\{ \begin{array}{l} \text{:initial-value } \text{foo}_{\text{NIL}} \\ \text{:from-end } \text{bool}_{\text{NIL}} \\ \text{:start } \text{start}_{\text{T}} \\ \text{:end } \text{end}_{\text{NIL}} \\ \text{:key } \text{function} \end{array} \right\}$)

▷ 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 22.

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

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

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

▷ Make a hash table.

(*f*gethash *key* *hash-table* [*default*_{NIL}])

▷ 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**)

▷ 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-prefixfoo-])
      (:constructor
       [makerMAKE-foo [(ord-λ*)]])
      (:copier
       [copierCOPY-foo])
      (:include struct
       {slot
        (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-namefoo-P])
    }
    [doc] {slot
           (slot [init {type slot-type
                       :read-only bool}])})
  )

```

▷ 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 18) is given, by (*maker* *arg** **{:key value}***). In the latter case, *args* and *:keys* correspond to the positional and keyword parameters defined in *ord-λ* whose *vars* in turn correspond to *slots*. **:print-object**/**:print-function** generate a *gprint-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*eq1 *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*eq1, or are equivalent **pathnames**, or are **conses** with *f*equal cars and cdrs, or are **strings** or **bit-vectors** with *f*eq1 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.

- (**fconstantp** *foo* [*environment*_{NIL}])
 ▷ T if *foo* is a constant form.
- (**functionp** *foo*) ▷ T if *foo* is of type **function**.
- (**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

- ($\left\{ \begin{array}{l} m\text{defconstant} \\ m\text{defparameter} \end{array} \right\}$ \widehat{foo} *form* [*doc*])
 ▷ Assign value of *form* to global constant/dynamic variable *foo*.
- (**mdefvar** \widehat{foo} [*form* [*doc*]])
 ▷ Unless bound already, assign value of *form* to dynamic variable *foo*.
- ($\left\{ \begin{array}{l} m\text{setf} \\ mpsetf \end{array} \right\}$ $\{place\ form\}^*$)
 ▷ Set *places* to primary values of *forms*. Return values of last *form*/NIL; work sequentially/in parallel, respectively.
- ($\left\{ \begin{array}{l} s\text{setq} \\ mpsetq \end{array} \right\}$ $\{symbol\ form\}^*$)
 ▷ Set *symbols* to primary values of *forms*. Return value of last *form*/NIL; work sequentially/in parallel, respectively.
- (**fset** \widetilde{symbol} *foo*) ▷ Set *symbol*'s value cell to *foo*. Deprecated.
- (**mmultiple-value-setq** *vars* *form*)
 ▷ Set elements of *vars* to the values of *form*. Return *form*'s primary value.
- (**mshiftf** \widetilde{place}^+ *foo*)
 ▷ Store value of *foo* in rightmost *place* shifting values of *places* left, returning first *place*.
- (**mrotatef** \widetilde{place}^*)
 ▷ Rotate values of *places* left, old first becoming new last *place*'s value. Return NIL.
- (**fmakunbound** \widetilde{foo}) ▷ Delete special variable *foo* if any.
- (**fget** *symbol* *key* [*default*_{NIL}])
 (**fgetf** *place* *key* [*default*_{NIL}])
 ▷ First entry *key* from property list stored in *symbol*/in *place*, respectively, or *default* if there is no *key*. **setfable**.
- (**fget-properties** *property-list* *keys*)
 ▷ Return *key* and *value* of first entry from *property-list* matching a key from $\underset{2}{keys}$, and tail of *property-list* starting with that key. Return NIL, NIL, and NIL if there was no matching key in *property-list*.
- (**fremprop** \widetilde{symbol} *key*)
 (**mremf** *place* *key*)
 ▷ Remove first entry *key* from property list stored in *symbol*/in *place*, respectively. Return T if *key* was there, or NIL otherwise.
- (**sprogv** *symbols* *values* $form^P$)
 ▷ Evaluate *forms* with locally established dynamic bindings of *symbols* to *values* or NIL. Return values of *forms*.
- ($\left\{ \begin{array}{l} s\text{let} \\ s\text{let}^* \end{array} \right\}$ $\left(\left\{ \begin{array}{l} name \\ (name\ [value_{\text{NIL}}]) \end{array} \right\}^* \right) (\text{declare } \widehat{decl}^*)^* form^P$)
 ▷ Evaluate *forms* with *names* lexically bound (in parallel or sequentially, respectively) to *values*. Return values of *forms*.

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

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

(*m***destructuring-bind** *destruct-λ bar* (**declare** \widehat{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

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

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

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

▷ 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*.

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

▷ 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*.

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

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

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

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

(*f***funcall** *function arg*^{*}) ▷ Values of *function* called with *args*.

(*s***multiple-value-call** *function form*^{*})

▷ Call *function* with all the values of each *form* as its arguments. Return values returned by *function*.

(*f***values-list** *list*) ▷ Return elements of *list*.

(*f***values** *foo*^{*})

▷ Return as multiple values the primary values of the *foos*. **setfable**.

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

(*m***nth-value** *n form*)

▷ Zero-indexed *n*th return value of *form*.

(*f***complement** *function*)

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

- (*f***constantly** *foo*)
 ▷ Function of any number of arguments returning *foo*.
- (*f***identity** *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***definition** $\left\{ \begin{array}{l} \textit{foo} \\ \text{(setf } \textit{foo}) \end{array} \right\}$)
 ▷ Definition of global function *foo*. **setfable**.
- (*f***makunbound** *foo*)
 ▷ Remove global function or macro definition *foo*.
- c**call-arguments-limit
clambda-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

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

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

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

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

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

or

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

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.

- ($\left\{ \begin{array}{l} \textit{mdefmacro} \\ \textit{fdefine-compiler-macro} \end{array} \right\} \left\{ \begin{array}{l} \textit{foo} \\ \text{(setf } \textit{foo}) \end{array} \right\} (\textit{macro-}\lambda^*) (\widehat{\text{declare } \textit{decl}^*})^*$
 $[\widehat{\text{doc}}] \textit{form}^{\text{P}^*}$)
 ▷ 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** $\widehat{\text{local-decl}^*}$)^{*} $\widehat{\text{doc}}$)
 $\textit{macro-form}^{\text{P}^*}$)^{*}) (**declare** $\widehat{\text{decl}^*}$)^{*} $\textit{form}^{\text{P}^*}$)
 ▷ 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** $\widehat{\text{decl}}^*$)^{P*} *form*^{P*})
 ▷ Evaluate forms with locally defined symbol macros *foo*.

(**m**defsetf $\widehat{\text{function}}$
 $\left\{ \begin{array}{l} \widehat{\text{updater}} \ [\widehat{\text{doc}}] \\ (\text{setf-}\lambda^*) \ (s\text{-var}^*) \ (\text{declare} \ \widehat{\text{decl}}^*) \ [\widehat{\text{doc}}] \ \text{form}^{\text{P}*} \end{array} \right\}$
 where defsetf lambda list (*setf-λ*^{*}) has the form
 $(\text{var}^* \ [\&\text{optional} \ \left\{ \begin{array}{l} \text{var} \\ (\text{var} \ [\text{init}_{\text{NIL}}] \ [\text{supplied-p}])] \end{array} \right\}^*]$
 $[\&\text{rest} \ \text{var}] \ [\&\text{key} \ \left\{ \begin{array}{l} \text{var} \\ (\text{:key} \ \text{var}) \end{array} \right\} \ [\text{init}_{\text{NIL}}] \ [\text{supplied-p}]] \right\}^*$
 $[\&\text{allow-other-keys}] \ [\&\text{environment} \ \text{var}]$
 ▷ 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 **sblock** named *function*.

(**m**define-setf-expander *function* (*macro-λ*^{*}) (**declare** $\widehat{\text{decl}}^*$)^{P*} [*doc*]
 $\text{form}^{\text{P}*}$)
 ▷ 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 **fget-setf-expansion** where the elements of macro lambda list *macro-λ*^{*} are bound to corresponding *args*. *forms* are enclosed in an implicit **sblock** 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**
 $\left\{ \begin{array}{l} \text{var} \\ (\text{var} \ [\text{init}_{\text{NIL}}] \ [\text{supplied-p}])] \end{array} \right\}^*$ [**&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.

λlambda-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 **slet***.

9.5 Control Flow

(**sif** *test* *then* [*else*_{NIL}])

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

(**mcond** (*test then*^{P*}_[*test*])*)

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

($\left\{ \begin{array}{l} m\text{when} \\ m\text{unless} \end{array} \right\}$ *test foo*^{P*})

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

(**mcase** *test* ($\left\{ \begin{array}{l} \widehat{key} \\ key \end{array} \right\}$ *foo*^{P*})* [$\left\{ \begin{array}{l} \text{otherwise} \\ T \end{array} \right\}$ *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*.

($\left\{ \begin{array}{l} m\text{ecase} \\ m\text{ccase} \end{array} \right\}$ *test* ($\left\{ \begin{array}{l} \widehat{key} \\ key \end{array} \right\}$ *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*.

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

▷ Evaluate **s***tagbody*-like body with *names* lexically bound (in parallel or sequentially, respectively) to *values*. Return NIL or explicitly mreturned values. Implicitly, the whole form is a **s***block* 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 **s***return-from*.

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

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

▷ Have nearest enclosing **s***block* 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 **s***go*. Return NIL.

(**s***go* \widehat{tag})

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

- (**s**catch *tag form*^{P*})
 ▷ Evaluate *forms* and return their values unless interrupted by **s**throw.
- (**s**throw *tag form*)
 ▷ Have the nearest dynamically enclosing **s**catch with a tag *τ* **eq** *tag* return with the values of *form*.
- (**f**sleep *n*) ▷ Wait *n* seconds; return NIL.

9.6 Iteration

- (**m**do **m**do*) ($\left\{ \begin{array}{l} \text{var} \\ (\text{var } [start \ [step]]) \end{array} \right\}^*$) (*stop result*^{P*}) (**declare** \widehat{decl}^*)
 $\left\{ \begin{array}{l} \widehat{tag} \\ \text{form} \end{array} \right\}^*$)
 ▷ Evaluate **s**tbody-like body with *vars* successively bound according to the values of the corresponding *start* and *step* forms. *vars* are bound in parallel/sequentially, respectively. Stop iteration when *stop* is T. Return values of result^{*}. Implicitly, the whole form is a **s**block named NIL.
- (**m**dotimes (*var i* [*result*_{NIL}]) (**declare** \widehat{decl}^*)^{*} $\left\{ \widehat{tag} \mid \text{form} \right\}^*$)
 ▷ Evaluate **s**tbody-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 **s**block named NIL.
- (**m**dolist (*var list* [*result*_{NIL}]) (**declare** \widehat{decl}^*)^{*} $\left\{ \widehat{tag} \mid \text{form} \right\}^*$)
 ▷ Evaluate **s**tbody-like body with *var* successively bound to the elements of *list*. Upon evaluation of *result*, *var* is NIL. Implicitly, the whole form is a **s**block named NIL.

9.7 Loop Facility

- (**m**loop *form*^{*})
 ▷ **Simple Loop**. If *forms* do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit **s**block named NIL.
- (**m**loop *clause*^{*})
 ▷ **Loop Facility**. For Loop Facility keywords see below and Figure 1.
- named** *n*_{NIL} ▷ Give **m**loop's implicit **s**block a name.
- with** $\left\{ \begin{array}{l} \text{var-s} \\ (\text{var-s}^*) \end{array} \right\} [d\text{-type}] [= \text{foo}]^+$
 $\left\{ \begin{array}{l} \text{var-p} \\ (\text{var-p}^*) \end{array} \right\} [d\text{-type}] [= \text{bar}]^*$
 where destructuring type specifier *d-type* has the form
 $\left\{ \begin{array}{l} \text{fixnum} \mid \text{float} \mid \text{T} \mid \text{NIL} \mid \left\{ \begin{array}{l} \text{of-type} \\ (\text{type}^*) \end{array} \right\} \end{array} \right\}$
 ▷ Initialize (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel.
- $\left\{ \left\{ \begin{array}{l} \text{for} \mid \text{as} \end{array} \right\} \left\{ \begin{array}{l} \text{var-s} \\ (\text{var-s}^*) \end{array} \right\} [d\text{-type}]^+ \left\{ \begin{array}{l} \text{and} \\ (\text{var-p}^*) \end{array} \right\} [d\text{-type}]^* \right\}$
 ▷ 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**.
- upfrom** | **from** | **downfrom** } *start*
 ▷ Start stepping with *start*
- upto** | **downto** | **to** | **below** | **above** } *form*
 ▷ Specify *form* as the end value for stepping.
- in** | **on** } *list*
 ▷ Bind *var* to successive elements/tails, respectively, of *list*.
- by** { *step*₁ | *function*_{##cdr} }
 ▷ Specify the (positive) decrement or increment or the *function* of one argument returning the next part of the list.

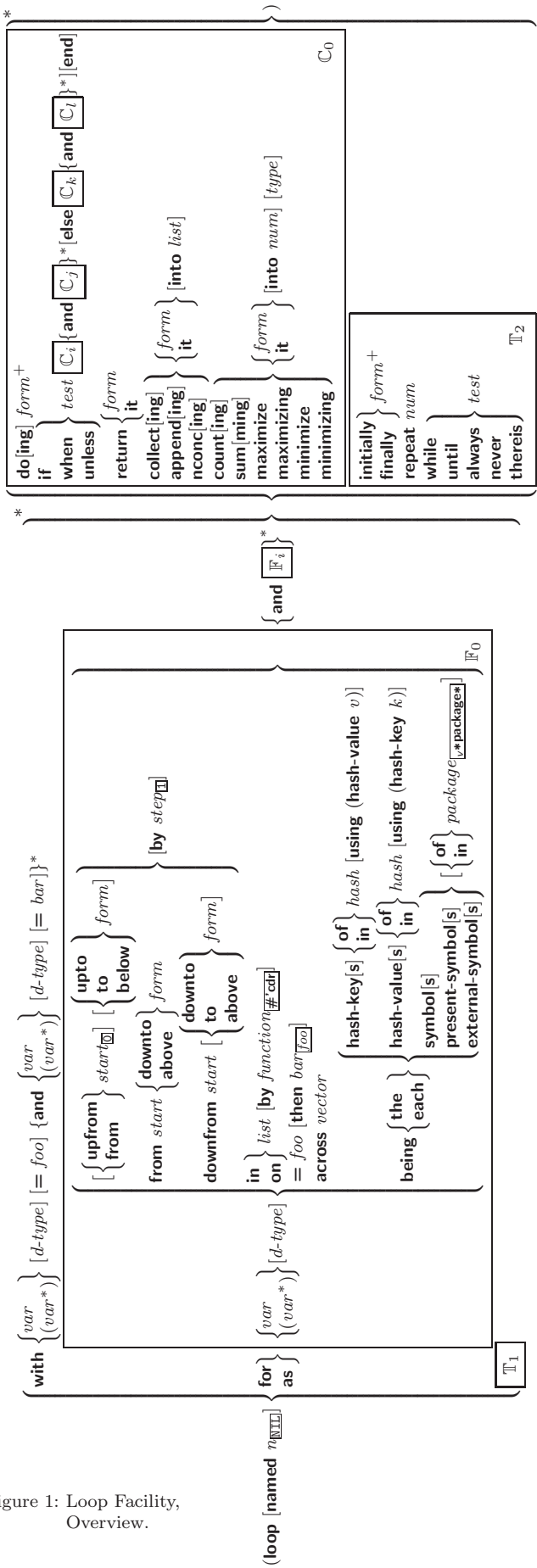


Figure 1: Loop Facility, Overview.

= *foo* [**then** *bar*_{*foo*}]

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

across *vector*

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

being {**the**|**each**}

▷ Iterate over a hash table or a package.

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

▷ Bind *var* successively to the keys of *hash-table*;
bind *value* to corresponding values.

{**hash-value**|**hash-values**} {**of**|**in**} *hash-table* [**using**
(**hash-key** *key*)]

▷ Bind *var* successively to the values of
hash-table; bind *key* to corresponding keys.

{**symbol**|**symbols**|**present-symbol**|**present-symbols**|
external-symbol|**external-symbols**} [{**of**|**in**}
*package*_{*package**}]

▷ Bind *var* successively to the accessible symbols,
or the present symbols, or the external symbols
respectively, of *package*.

{**do**|**doing**} *form*⁺

▷ Evaluate *forms* in every iteration.

{**if**|**when**|**unless**} *test* *i-clause* {**and** *j-clause*}* [**else** *k-clause*
{**and** *l-clause*}*] [**end**]

▷ If *test* returns T, T, or NIL, respectively, evaluate
i-clause and *j-clauses*; otherwise, evaluate *k-clause* and
l-clauses.

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 ter-
mination.

{**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 evalu-
ated once.

{**while**|**until**} *test*

▷ Continue iteration until *test* returns NIL or T, respec-
tively.

- {always|never} test**
 ▷ Terminate *mloop* returning NIL and skipping any **finally** parts as soon as *test* is NIL or T, respectively. Otherwise continue *mloop* with its default return value set to T.
- thereis test**
 ▷ Terminate *mloop* when *test* is T and return value of *test*, skipping any **finally** parts. Otherwise continue *mloop* with its default return value set to NIL.
- (*mloop-finish*)**
 ▷ Terminate *mloop* immediately executing any **finally** clauses and returning any accumulated results.

10 CLOS

10.1 Classes

- (*fslot-exists-p* foo bar)** ▷ T if *foo* has a slot *bar*.
- (*fslot-boundp* instance slot)** ▷ T if *slot* in *instance* is bound.

(*mdefclass* foo (superclass* standard-object)
 { (slot { { :reader reader}*
 { :writer { writer
 (setf writer) } }*
 { :accessor accessor}*
 :allocation { :instance
 :class } instance }) })
 { (:default-initargs { name value }*)
 (:documentation class-doc)
 (:metaclass name standard-class) })

▷ Define or modify class *foo* as a subclass of *superclasses*. Transform existing instances, if any, by *gmake-instances-obsolete*. In a new instance *i* of *foo*, a *slot*'s value defaults to *form* unless set via *:initarg-name*; it is readable via (*reader i*) or (*accessor i*), and writable via (*writer value i*) or (**setf** (*accessor i*) *value*). *slots* with **:allocation :class** are shared by all instances of class *foo*.

- (*ffind-class* symbol [*errorp*_□ [*environment*]])**
 ▷ Return class named *symbol*. **setfable**.
- (*gmake-instance* class { :initarg value }* other-keyarg*)**
 ▷ Make new instance of class.
- (*greinitialize-instance* instance { :initarg value }* other-keyarg*)**
 ▷ Change local slots of instance according to *initargs* by means of *gshared-initialize*.
- (*fslot-value* foo slot)** ▷ Return value of slot in foo. **setfable**.
- (*fslot-makunbound* instance slot)**
 ▷ Make *slot* in instance unbound.

(*mwith-slots* ({ slot ({ var slot })* }) instance (**declare decl*)
*form**)**
 ▷ Return values of forms after evaluating them in a lexical environment with slots of *instance* visible as **setfable slots** or *vars*/with *accessors* of *instance* visible as **setfable vars**.

- (*gclass-name* class)**
((setf *gclass-name*) new-name class) ▷ Get/set name of class.
- (*fclass-of* foo)** ▷ Class *foo* is a direct instance of.

(**gchange-class** $\widetilde{instance}$ *new-class* $\{ :initarg\ value\}^*$ *other-keyarg**)
 ▷ Change class of instance to *new-class*. Retain the status of any slots that are common between *instance*'s original class and *new-class*. Initialize any newly added slots with the *values* of the corresponding *initargs* if any, or with the values of their **:initform** forms if not.

(**gmake-instances-obsolete** *class*)
 ▷ Update all existing instances of *class* using **gupdate-instance-for-redefined-class**.

(**ginitialize-instance** *instance*
gupdate-instance-for-different-class *previous current*
 $\{ :initarg\ value\}^*$ *other-keyarg**)
 ▷ Set slots on behalf of **gmake-instance**/of **gchange-class** by means of **gshared-initialize**.

(**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* $\left\{ \begin{array}{l} \textit{initform-slots} \\ \text{T} \end{array} \right\}$ $\{ :initarg-slot\ value\}^*$ *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** $\left\{ \begin{array}{l} \textit{foo} \\ (\text{setf } \textit{foo}) \end{array} \right\}$ (*required-var** [**&optional** $\left\{ \begin{array}{l} \textit{var} \\ (\textit{var}) \end{array} \right\}^*$]
 [**&rest** *var*] [**&key** $\left\{ \begin{array}{l} \textit{var} \\ (\textit{var} | (:key\ \textit{var})) \end{array} \right\}^*$ [**&allow-other-keys**]])
 $\left. \left\{ \begin{array}{l} (:argument-precedence-order\ \textit{required-var}^+) \\ (\text{declare}\ (\text{optimize}\ \textit{method-selection-optimization})^+) \\ (:documentation\ \textit{string}) \\ (:generic-function-class\ \textit{gf-class}\ \text{standard-generic-function}) \\ (:method-class\ \textit{method-class}\ \text{standard-method}) \\ (:method-combination\ \textit{c-type}\ \text{standard}\ \textit{c-arg}^*) \\ (:method\ \textit{defmethod-args})^* \end{array} \right\} \right)$)

▷ Define or modify generic function *foo*. Remove any methods previously defined by **defgeneric**. *gf-class* and the lambda parameters *required-var** and *var** must be compatible with existing methods. *defmethod-args* resemble those of **mdefmethod**. For *c-type* see section 10.3.

(**fensure-generic-function** $\left\{ \begin{array}{l} \textit{foo} \\ (\text{setf } \textit{foo}) \end{array} \right\}$)

$$\left. \begin{array}{l} \text{:argument-precedence-order } \textit{required-var}^+ \\ \text{:declare (optimize } \textit{method-selection-optimization})} \\ \text{:documentation } \textit{string} \\ \text{:generic-function-class } \textit{gf-class} \\ \text{:method-class } \textit{method-class} \\ \text{:method-combination } \textit{c-type c-arg}^* \\ \text{:lambda-list } \textit{lambda-list} \\ \text{:environment } \textit{environment} \end{array} \right\}$$

▷ 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.

$$\begin{array}{l} (\textit{mdefmethod} \left\{ \begin{array}{l} \textit{foo} \\ (\textit{setf } \textit{foo}) \end{array} \right\} \left[\begin{array}{l} \text{:before} \\ \text{:after} \\ \text{:around} \end{array} \right] \left\{ \begin{array}{l} \text{primary method} \\ \textit{qualifier}^* \end{array} \right\} \\ \left(\left\{ \begin{array}{l} \textit{var} \\ (\textit{spec-var} \left\{ \begin{array}{l} \textit{class} \\ (\textit{eql } \textit{bar}) \end{array} \right\}) \end{array} \right\} \right)^* \text{ [&optional} \\ \left\{ \begin{array}{l} \textit{var} \\ (\textit{var} [\textit{init} [\textit{supplied-p}]]) \end{array} \right\} \text{ [&rest } \textit{var} \text{ [&key} \\ \left\{ \begin{array}{l} \textit{var} \\ (\left\{ \begin{array}{l} \textit{var} \\ (\text{:key } \textit{var}) \end{array} \right\} [\textit{init} [\textit{supplied-p}]]) \end{array} \right\} \text{ [&allow-other-keys]]} \\ \text{ [&aux} \left\{ \begin{array}{l} \textit{var} \\ (\textit{var} [\textit{init}]) \end{array} \right\} \right\} \left\{ \left(\begin{array}{l} \text{declare } \widehat{\textit{decl}}^* \\ \text{doc} \end{array} \right)^* \text{ form}^{\text{P}^*} \end{array} \right.$$

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

$$\left\{ \begin{array}{l} \textit{gadd-method} \\ \textit{gremove-method} \end{array} \right\} \textit{generic-function method}$$

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

$$(\textit{gfind-method} \textit{generic-function} \textit{qualifiers} \textit{specializers} [\textit{error} \text{□}])$$

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

$$(\textit{gcompute-applicable-methods} \textit{generic-function} \textit{args})$$

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

$$(\textit{fcall-next-method} \textit{arg}^* \text{current args})$$

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

$$(\textit{gno-applicable-method} \textit{generic-function} \textit{arg}^*)$$

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

$$\left\{ \begin{array}{l} \textit{finvalid-method-error} \textit{method} \\ \textit{fmethod-combination-error} \end{array} \right\} \textit{control} \textit{arg}^*$$

▷ Signal **error** on applicable method with invalid qualifiers, or on method combination. For *control* and *args* see **format**, page 38.

$$(\textit{gno-next-method} \textit{generic-function} \textit{method} \textit{arg}^*)$$

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

$$(\textit{gfunction-keywords} \textit{method})$$

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

$$(\textit{gmethod-qualifiers} \textit{method}) \quad \triangleright \text{ List of } \underline{\text{qualifiers}}$$
 of *method*.

10.3 Method Combination Types

standard

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

(m define-method-combination *c-type*

$$\left\{ \begin{array}{l} \text{:documentation } \widehat{string} \\ \text{:identity-with-one-argument } \text{bool}_{\text{NIL}} \\ \text{:operator } \text{operator}_{\text{[c-type]}} \end{array} \right\}$$

▷ **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, f **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 $\left[\begin{array}{l} \text{:most-specific-first} \\ \text{:most-specific-last} \end{array} \right]_{\text{[:most-specific-first]}}$ (specified as *c-arg* in m **defgeneric**). Using *c-type* as the *qualifier* in m **defmethod** makes the method primary.

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

$$\left\{ \begin{array}{l} * \\ \text{(qualifier* [*]) } \\ \text{predicate} \end{array} \right\}$$

$$\left\{ \begin{array}{l} \text{:description } \text{control} \\ \text{:order } \left\{ \begin{array}{l} \text{:most-specific-first} \\ \text{:most-specific-last} \end{array} \right\}_{\text{[:most-specific-first]}} \\ \text{:required } \text{bool} \end{array} \right\}^*)$$

$$\left\{ \begin{array}{l} \text{(:arguments } \text{method-combination-}\lambda^*) \\ \text{(:generic-function } \text{symbol}) \\ \text{(declare } \widehat{decl}^*) \\ \widehat{doc} \end{array} \right\} \text{body}^*$$

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

(m call-method

$$\left\{ \begin{array}{l} \widehat{method} \\ \text{(mmake-method } \widehat{form}) \end{array} \right\} \left[\left(\left\{ \begin{array}{l} \widehat{next-method} \\ \text{(mmake-method } \widehat{form}) \end{array} \right\}^* \right) \right]$$

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

11 Conditions and Errors

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

(*mdefine-condition* *foo* (*parent-type** condition)

$$\left(\left(\left(\left(\left(\left(\begin{array}{l} \text{:reader } \text{reader}^* \\ \text{:writer } \left\{ \begin{array}{l} \text{writer} \\ \text{(setf writer)} \end{array} \right\}^* \\ \text{:accessor } \text{accessor}^* \\ \text{:allocation } \left\{ \begin{array}{l} \text{:instance} \\ \text{:class} \end{array} \right\} \text{instance} \\ \text{:initarg } \text{initarg-name}^* \\ \text{:initform } \text{form} \\ \text{:type } \text{type} \\ \text{:documentation } \text{slot-doc} \end{array} \right\} \right) \right) \right) \right) \right) \right)^* \\ \left(\left(\begin{array}{l} \text{:default-initargs } \{ \text{name value} \}^* \\ \text{:documentation } \text{condition-doc} \\ \text{:report } \left\{ \begin{array}{l} \text{string} \\ \text{report-function} \end{array} \right\} \end{array} \right) \right) \right)$$

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

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

▷ Return new instance of *condition-type*.

$$\left(\left(\begin{array}{l} \text{fsignal} \\ \text{fwarn} \\ \text{ferror} \end{array} \right) \left(\begin{array}{l} \text{condition} \\ \text{condition-type } \{ \text{:initarg-name value} \}^* \\ \text{control arg}^* \end{array} \right) \right)$$

▷ 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 38), **simple-condition**, **simple-warning**, or **simple-error**, respectively. From *f***signal** and *f***warn**, return NIL.

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

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

(*mignore-errors* *form*^{P*})

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

(*finvoke-debugger* *condition*)

▷ Invoke debugger with *condition*.

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

▷ 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 38), **error**. When using the debugger's continue option, *places* can be altered before re-evaluation of *test*. Return NIL.

(*mhandler-case* *foo* (*type* ([*var*]) (**declare** $\widehat{\text{decl}}^*$)^{P*} *condition-form*^{P*})^{*} [**:no-error** (*ord-λ**) (**declare** $\widehat{\text{decl}}^*$)^{P*} *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 18 for (*ord-λ**).

(*mhandler-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*.

(**mwith-simple-restart** ($\left\{ \begin{array}{l} \text{restart} \\ \text{NIL} \end{array} \right\}$ *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 38) and return NIL and T.

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

(**declare** $\widehat{\text{decl}}^*$)^{*} *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 18 for *ord-λ**.

(**mrestart-bind** ($\left(\left\{ \begin{array}{l} \text{restart} \\ \text{NIL} \end{array} \right\} \text{restart-function} \right. \\ \left. \left\{ \begin{array}{l} \text{:interactive-function } \text{arg-function} \\ \text{:report-function } \text{report-function} \\ \text{:test-function } \text{test-function} \end{array} \right\} \right)^*$) *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.

($\left\{ \begin{array}{l} \text{f find-restart} \\ \text{f compute-restarts } \text{name} \end{array} \right\}$ [*condition*])

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

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

($\left\{ \begin{array}{l} \text{f abort} \\ \text{f muffle-warning} \\ \text{f continue} \\ \text{f store-value } \text{value} \\ \text{f use-value } \text{value} \end{array} \right\}$ [*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.

(**mwith-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*.
- v**break-on-signals*_{NIL}
 ▷ 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*}*etypecase**}* *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*.

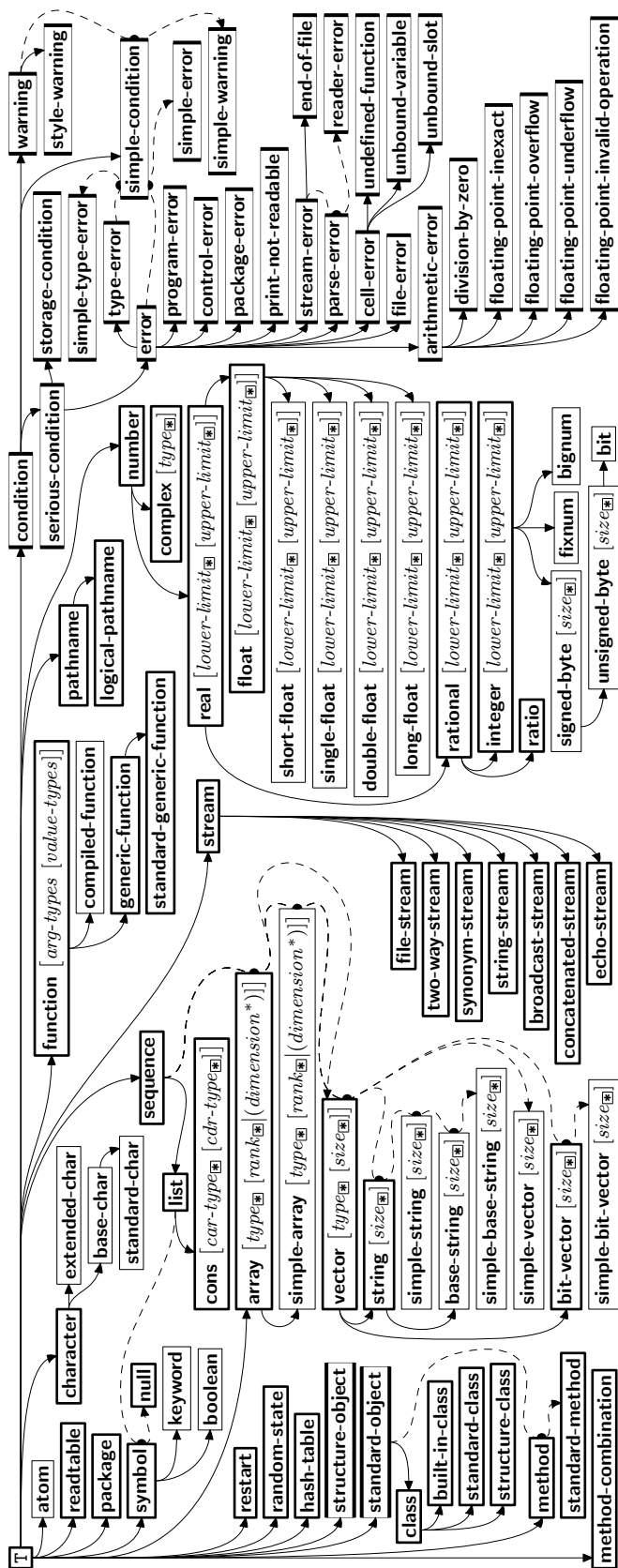


Figure 2: Precedence Order of System Classes (□), Classes (▣), Types (▤), and Condition Types (▥).

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

- (*m*def^Ptype *foo* (*macro-λ**) (**declare** \widehat{decl}^*)* [\widehat{doc}] *form*^P*)
 ▷ Define type *foo* which when referenced as (*foo* \widehat{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 19 but with default value of * instead of NIL. *forms* are enclosed in an implicit **block** named *foo*.
- (**eql** *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**_T) ▷ Type specifier for intersection of *types*.
- (**or** *type**_{NIL}) ▷ Type specifier for union of *types*.
- (**values** *type** [**&optional** *type** [**&rest** *other-args*]])
 ▷ Type specifier for multiple values.
- * ▷ As a type argument (cf. Figure 2): no restriction.

13 Input/Output

13.1 Predicates

- (*f*stream^Pp *foo*)
 (*f*pathname^P *foo*) ▷ T if *foo* is of indicated type.
 (*f*readtable^P *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 38, *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) [\widetilde{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.

- (**fread-delimited-list** *char* [\widetilde{stream} $\underline{v*standard-input*}$ [*recursive* \underline{NIL}]])
 ▷ Continue reading until encountering *char*. Return list of objects read. Signal error if no *char* is found in stream.
- (**fread-char** [\widetilde{stream} $\underline{v*standard-input*}$ [*eof-err* \underline{T}] [*eof-val* \underline{NIL}] [*recursive* \underline{NIL}]])
 ▷ Return next character from *stream*.
- (**fread-char-no-hang** [\widetilde{stream} $\underline{v*standard-input*}$ [*eof-error* \underline{T}] [*eof-val* \underline{NIL}] [*recursive* \underline{NIL}]])
 ▷ Next character from *stream* or NIL if none is available.
- (**fpeek-char** [*mode* \underline{NIL}] [\widetilde{stream} $\underline{v*standard-input*}$ [*eof-error* \underline{T}] [*eof-val* \underline{NIL}] [*recursive* \underline{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.
- (**funread-char** *character* [\widetilde{stream} $\underline{v*standard-input*}$])
 ▷ Put last **fread-char**ed *character* back into *stream*; return NIL.
- (**fread-byte** \widetilde{stream} [*eof-err* \underline{T}] [*eof-val* \underline{NIL}])
 ▷ Read next byte from binary *stream*.
- (**fread-line** [\widetilde{stream} $\underline{v*standard-input*}$ [*eof-err* \underline{T}] [*eof-val* \underline{NIL}] [*recursive* \underline{NIL}]])
 ▷ Return a line of text from *stream* and \underline{T} if line has been ended by end of file.
- (**fread-sequence** *sequence* \widetilde{stream} [*:start* *start* $\underline{0}$][*:end* *end* \underline{NIL}])
 ▷ Replace elements of *sequence* between *start* and *end* with elements from binary or character *stream*. Return index of *sequence*'s first unmodified element.
- (**freadtable-case** *readtable*) \underline{upcase}
 ▷ Case sensitivity attribute (one of **:upcase**, **:downcase**, **:preserve**, **:invert**) of *readtable*. **settable**.
- (**fcopy-readtable** [*from-readtable* $\underline{v*readtable*}$ [*to-readtable* \underline{NIL}]])
 ▷ Return copy of from-readtable.
- (**fset-syntax-from-char** *to-char* *from-char* [*to-readtable* $\underline{v*readtable*}$ [*from-readtable* $\underline{standard\ readtable}$]])
 ▷ Copy syntax of *from-char* to *to-readtable*. Return T.
- v*readtable*** ▷ Current readtable.
- v*read-base*** $\underline{10}$ ▷ Radix for reading **integers** and **ratios**.
- v*read-default-float-format*** $\underline{single-float}$
 ▷ Floating point format to use when not indicated in the number read.
- v*read-suppress*** \underline{NIL}
 ▷ If T, reader is syntactically more tolerant.
- (**fset-macro-character** *char* *function* [*non-term-p* \underline{NIL}] [\widetilde{rt} $\underline{v*readtable*}$])
 ▷ Make *char* a macro character associated with *function* of stream and *char*. Return T.
- (**fget-macro-character** *char* [*rt* $\underline{v*readtable*}$])
 ▷ Reader macro function associated with *char*, and \underline{T} if *char* is a non-terminating macro character.
- (**fmake-dispatch-macro-character** *char* [*non-term-p* \underline{NIL}] [*rt* $\underline{v*readtable*}$])
 ▷ Make *char* a dispatching macro character. Return T.

- (*f***set-dispatch-macro-character** *char sub-char function*
 $[rt_{\text{v}*readtable*}]$)
 ▷ 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* $[rt_{\text{v}*readtable*}]$)
 ▷ 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*.
- #Bn; #On; n.; #Xn; #rRn**
 ▷ Integer of radix 2, 8, 10, 16, or *r*; $2 \leq r \leq 36$.
- n/d* ▷ The **ratio** $\frac{n}{d}$.
- $\{[m].n[\{S|F|D|L|E\}x_{\text{E0}}]m[.n]\{S|F|D|L|E\}x\}$
 ▷ *m.n* · 10^{*x*} as **short-float**, **single-float**, **double-float**, **long-float**, or the type from ***read-default-float-format***.
- #C(a b)** ▷ (*fcomplex a b*), the complex number *a* + *bi*.
- #'foo** ▷ (*sfunction foo*); the function named *foo*.
- #nAsequence** ▷ *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*.
- #Pstring** ▷ A pathname.
- #:foo** ▷ Uninterned symbol *foo*.
- #.form** ▷ Read-time value of *form*.
- v*read-eval*□** ▷ If NIL, a **reader-error** is signalled at **#.**
- #integer= foo** ▷ Give *foo* the label *integer*.
- #integer#** ▷ Object labelled *integer*.
- #<** ▷ Have the reader signal **reader-error**.

#+feature *when-feature*

#-feature *unless-feature*

▷ Means *when-feature* if *feature* is **T**; means *unless-feature* if *feature* is **NIL**. *feature* is a symbol from **v*features***, or (**{and |or} feature***), or (**(not feature)**).

v*features*

▷ List of symbols denoting implementation-dependent features.

|c*|; \c

▷ Treat arbitrary character(s) *c* as alphabetic preserving case.

13.4 Printer

$\left(\begin{array}{l} \text{f} \text{prin1} \\ \text{f} \text{print} \\ \text{f} \text{pprint} \\ \text{f} \text{princ} \end{array} \right) \text{foo } [\widetilde{\text{stream}}_{\text{v*standard-output*}}])$

▷ Print *foo* to *stream* **freadably**, **freadably** between a newline and a space, **freadably** after a newline, or **human-readably** without any extra characters, respectively. **fprin1**, **fprint** and **fprinc** return *foo*.

(fprin1-to-string foo)

(fprinc-to-string foo)

▷ Print *foo* to *string* **freadably** or **human-readably**, respectively.

(gprint-object object stream)

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

(mprint-unreadable-object (foo stream {:type** *bool*_{**NIL**} **:identity** *bool*_{**NIL**}}) *form*^{P*})**

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

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

▷ Output a newline to *stream*. Return **NIL**.

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

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

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

▷ Output *char* to *stream*.

$\left(\begin{array}{l} \text{f} \text{write-string} \\ \text{f} \text{write-line} \end{array} \right) \text{string } [\widetilde{\text{stream}}_{\text{v*standard-output*}} [\left\{ \begin{array}{l} \text{:start } \text{start}_{\text{0}} \\ \text{:end } \text{end}_{\text{NIL}} \end{array} \right\}]])$

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

(fwrite-byte byte stream)

▷ Write *byte* to binary *stream*.

(fwrite-sequence sequence stream {:start** *start*_{**0**} **:end** *end*_{**NIL**}})**

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

$(\left. \begin{array}{l} \text{_fwrite} \\ \text{_fwrite-to-string} \end{array} \right\} \textit{foo} \left. \begin{array}{l} \text{_array } \textit{bool} \\ \text{_base } \textit{radix} \\ \text{_case } \left\{ \begin{array}{l} \text{_upcase} \\ \text{_downcase} \\ \text{_capitalize} \end{array} \right\} \\ \text{_circle } \textit{bool} \\ \text{_escape } \textit{bool} \\ \text{_gensym } \textit{bool} \\ \text{_length } \{ \textit{int} | \text{NIL} \} \\ \text{_level } \{ \textit{int} | \text{NIL} \} \\ \text{_lines } \{ \textit{int} | \text{NIL} \} \\ \text{_miser-width } \{ \textit{int} | \text{NIL} \} \\ \text{_pprint-dispatch } \textit{dispatch-table} \\ \text{_pretty } \textit{bool} \\ \text{_radix } \textit{bool} \\ \text{_readably } \textit{bool} \\ \text{_right-margin } \{ \textit{int} | \text{NIL} \} \\ \text{_stream } \textit{stream} \boxed{\text{_v*standard-output*}} \end{array} \right\})$

▷ 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 **_fwrite** only.)

$(\text{_pprint-fill } \textit{stream} \textit{foo} [\textit{parenthesis} \boxed{\text{_m}} [\textit{noop}]])$
 $(\text{_pprint-tabular } \textit{stream} \textit{foo} [\textit{parenthesis} \boxed{\text{_m}} [\textit{noop} [\textit{n} \boxed{\text{_16}}]])])$
 $(\text{_pprint-linear } \textit{stream} \textit{foo} [\textit{parenthesis} \boxed{\text{_m}} [\textit{noop}]])$

▷ 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 **_format** directive `~//`.

$(\text{_mpprint-logical-block } (\textit{stream} \textit{list} \left\{ \left\{ \begin{array}{l} \text{_prefix } \textit{string} \\ \text{_per-line-prefix } \textit{string} \end{array} \right\} \right\} \left. \begin{array}{l} \\ \\ \text{_suffix } \textit{string} \boxed{\text{_m}} \end{array} \right\}))$

$(\text{_declare } \widehat{\textit{decl}}^*)^* \textit{form}^{\text{Pk}}$

▷ 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 **_fwrite**. Return NIL.

(_mpprint-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*.

$(\text{_pprint-tab} \left\{ \begin{array}{l} \text{_line} \\ \text{_line-relative} \\ \text{_section} \\ \text{_section-relative} \end{array} \right\} \textit{c} \textit{i} [\textit{stream} \boxed{\text{_v*standard-output*}}])$

▷ Move cursor forward to column number $c + ki$, $k \geq 0$ being as small as possible.

$(\text{_pprint-indent} \left\{ \begin{array}{l} \text{_block} \\ \text{_current} \end{array} \right\} \textit{n} [\textit{stream} \boxed{\text{_v*standard-output*}}])$

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

$(\text{_mpprint-exit-if-list-exhausted})$

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

$(\text{_pprint-newline} \left\{ \begin{array}{l} \text{_linear} \\ \text{_fill} \\ \text{_miser} \\ \text{_mandatory} \end{array} \right\} [\textit{stream} \boxed{\text{_v*standard-output*}}])$

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

_v*print-array* ▷ If T, print arrays **_readably**.

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

- v*print-case***_[upcase]
 ▷ Print symbol names all uppercase (**:upcase**), all lowercase (**:downcase**), capitalized (**:capitalize**).
- v*print-circle***_[NIL]
 ▷ If T, avoid indefinite recursion while printing circular structure.
- v*print-escape***_[]
 ▷ 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*]_[]
 [*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** *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*]_[] [, [*col-inc*]_[] [, [*min-pad*]_[] [, '*pad-char*']_[]]]
 [:] [**@**] {**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 @, add *pad-chars* on the left rather than on the right.
- ~ [*radix*]_[] [, [*width*] [, '*pad-char*']_[] [, '*comma-char*']_[]
 [, *comma-interval*]_[]]] [:] [**@**] **R**
 ▷ **Radix**. (With one or more prefix arguments.) Print argument as number; with :, group digits *comma-interval* each; with @, always prepend a sign.

- {~R|~:R|~@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] [, [comma-char] [, [comma-interval]]] [:] [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] [, [overflow-char] [, [pad-char]]]] [C] F
- ▷ **Fixed-Format Floating-Point**. With C, always prepend a sign.
- ~ [width] [, [dec-digits] [, [exp-digits] [, [scale-factor] [, [overflow-char] [, [pad-char] [, [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] [, [int-digits] [, [width] [, [pad-char]]]] [:] [C] S
- ▷ **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] % ▷ **Newline**. Print *n* newlines.
- ~ [n] &
- ▷ **Fresh-Line**. Print *n* - 1 newlines if output stream is at the beginning of a line, or *n* newlines otherwise.
- {~|~:|~@|~@:}
- ▷ **Conditional Newline**. Print a newline like `pprint-newline` with argument *:linear*, *:fill*, *:miser*, or *:mandatory*, respectively.
- {~:~<|~@~<|~<~<}
- ▷ **Ignored Newline**. Ignore newline, or whitespace following newline, or both, respectively.
- ~ [n] | ▷ **Page**. Print *n* page separators.
- ~ [n] ~ ▷ **Tilde**. Print *n* tildes.
- ~ [min-col] [, [col-inc] [, [min-pad] [, [pad-char]]]] [:] [C] < [nl-text ~[spare] [,width]]::: {text ~;}* text ~>
- ▷ **Justification**. Justify text produced by *texts* in a field of at least *min-col* columns. With :, right justify; with C, left justify. If this would leave less than *spare* characters on the current line, output *nl-text* first.
- ~ [:] [C] < { [prefix] ~; } [per-line-prefix ~@;] body [~; suffix] ~: [C] >
- ▷ **Logical Block**. Act like `pprint-logical-block` using *body* as *f* **format** control string on the elements of the list argument or, with C, 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.

- $\{\sim [n_{\text{0}}] \text{ i} | \sim [n_{\text{0}}] \text{ :i}\}$
 ▷ **Indent.** Set indentation to n relative to leftmost/to current position.
- $\sim [c_{\text{0}}] [,i_{\text{0}}] [:] [\text{0}] \text{ T}$
 ▷ **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 0 , move to column number $c_0 + c + ki$ where c_0 is the current position.
- $\{\sim [m_{\text{0}}] * | \sim [m_{\text{0}}] \text{ :*} | \sim [n_{\text{0}}] \text{ 0*}\}$
 ▷ **Go-To.** Jump m arguments forward, or backward, or to argument n .
- $\sim [limit] [:] [\text{0}] \{ text \sim \}$
 ▷ **Iteration.** Use $text$ repeatedly, up to $limit$, as control string for the elements of the list argument or (with 0) for the remaining arguments. With $:$ or 0 , list elements or remaining arguments should be lists of which a new one is used at each iteration step.
- $\sim [x [y [,z]]] \wedge$
 ▷ **Escape Upward.** Leave immediately $\sim < \sim >$, $\sim < \sim : >$, $\sim \{ \sim \}$, $\sim ?$, 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.
- $\sim [i] [:] [\text{0}] [[\{text \sim;\}^* text] [\sim;; default] \sim]$
 ▷ **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 0 , do nothing for an argument value of NIL. Use the only $text$ and leave the argument to be read again if it is T.
- $\{\sim ? | \sim \text{0} ?\}$
 ▷ **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.
- $\sim [prefix \{,prefix\}^*] [:] [\text{0}] / [package [:] :cl-user:] function /$
 ▷ **Call Function.** Call all-uppercase $package::function$ with the arguments stream, format-argument, colon-p, at-sign-p and $prefixes$ for printing format-argument.
- $\sim [:] [\text{0}] \text{ W}$
 ▷ **Write.** Print argument of any type obeying every printer control variable. With $:$, pretty-print. With 0 , print without limits on length or depth.
- $\{\text{V} | \#\}$
 ▷ In place of the comma-separated prefix parameters: use next argument or number of remaining unprocessed arguments, respectively.

13.6 Streams

- (f **open path**) {
- | | | | | | | | | | | | |
|--|---|--------------|--------------------|------------|--------------------|---|---|------------|-----|--|--|
| :direction | <table border="0" style="border-collapse: collapse;"> <tr> <td style="border-right: 1px solid black; padding-right: 5px;">:input</td> <td style="padding-right: 5px;">:output</td> </tr> <tr> <td style="border-right: 1px solid black; padding-right: 5px;">:io</td> <td style="padding-right: 5px;">:probe</td> </tr> </table> | :input | :output | :io | :probe | :input | | | | | |
| :input | :output | | | | | | | | | | |
| :io | :probe | | | | | | | | | | |
| :element-type | <table border="0" style="border-collapse: collapse;"> <tr> <td style="border-right: 1px solid black; padding-right: 5px;">:type</td> <td style="padding-right: 5px;">:default</td> </tr> </table> | :type | :default | :character | | | | | | | |
| :type | :default | | | | | | | | | | |
| :if-exists | <table border="0" style="border-collapse: collapse;"> <tr> <td style="border-right: 1px solid black; padding-right: 5px;">:new-version</td> <td style="padding-right: 5px;">:error</td> <td style="padding-right: 5px;">:rename</td> <td style="padding-right: 5px;">:rename-and-delete</td> </tr> <tr> <td style="border-right: 1px solid black; padding-right: 5px;">:overwrite</td> <td style="padding-right: 5px;">:append</td> <td style="padding-right: 5px;">:supersede</td> <td style="padding-right: 5px;">NIL</td> </tr> </table> | :new-version | :error | :rename | :rename-and-delete | :overwrite | :append | :supersede | NIL | <table border="0" style="border-collapse: collapse;"> <tr> <td style="border: 1px solid black; padding: 2px;">:new-version if <i>path</i> specifies :newest; NIL otherwise</td> </tr> </table> | :new-version if <i>path</i> specifies :newest; NIL otherwise |
| :new-version | :error | :rename | :rename-and-delete | | | | | | | | |
| :overwrite | :append | :supersede | NIL | | | | | | | | |
| :new-version if <i>path</i> specifies :newest; NIL otherwise | | | | | | | | | | | |
| :if-does-not-exist | <table border="0" style="border-collapse: collapse;"> <tr> <td style="border-right: 1px solid black; padding-right: 5px;">:error</td> <td style="padding-right: 5px;">:create</td> </tr> <tr> <td style="border-right: 1px solid black; padding-right: 5px;">NIL</td> <td style="padding-right: 5px;">NIL</td> </tr> </table> | :error | :create | NIL | NIL | <table border="0" style="border-collapse: collapse;"> <tr> <td style="border: 1px solid black; padding: 2px;">NIL for :direction :probe; {:create} :error otherwise</td> </tr> </table> | NIL for :direction :probe; {:create} :error otherwise | | | | |
| :error | :create | | | | | | | | | | |
| NIL | NIL | | | | | | | | | | |
| NIL for :direction :probe; {:create} :error otherwise | | | | | | | | | | | |
| :external-format | <i>format</i> | :default | | | | | | | | | |
- ▷ 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*₀ [*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
*f*force-output
*f*finish-output
} [*stream*_{v*standard-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**)^{P*} *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**)^{P*} *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**)^{P*} *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**)
 ▷ 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.

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

√*standard-input*

√*standard-output*

√*error-output*

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

√*debug-io*

√*query-io*

▷ Bidirectional streams for debugging and user interaction.

13.7 Pathnames and Files

(*f*make-pathname

{	:host { <i>host</i> NIL :unspecific}	}
{	:device { <i>device</i> NIL :unspecific}	}
{	:directory {	}
{	{ <i>directory</i> :wild NIL :unspecific}	}
{	{ <i>file-name</i> :wild NIL :unspecific}	}
{	:name { <i>file-name</i> :wild NIL :unspecific}	}
{	:type { <i>file-type</i> :wild NIL :unspecific}	}
{	:version { <i>newest</i> <i>version</i> :wild NIL :unspecific}	}
{	:defaults <i>path</i> _{host from √*default-pathname-defaults*}	}
{	:case {: <i>local</i> : <i>common</i> : <i>local</i> }	}

▷ 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 } *path-or-stream* [:case {:*local*
*f*pathname-name }:*common*|:*local*])
*f*pathname-type }

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

(*f*parse-namestring *foo* [*host*
 [*default-pathname*_{√*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*_{√*default-pathname-defaults*}
 [*default-version*_{:newest}]])
 ▷ Return pathname made by filling in components missing in *path-or-stream* from *default-path-or-stream*.

√*default-pathname-defaults*

▷ Pathname to use if one is needed and none supplied.

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

- (**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*.
- (**namestring** *path-or-stream*)
 (**file-namestring** *path-or-stream*)
 (**directory-namestring** *path-or-stream*)
 (**host-namestring** *path-or-stream*)
 ▷ Return string representing full pathname; name, type, and version; directory name; or host name, respectively, of *path-or-stream*.
- (**translate-pathname** *path-or-stream wildcard-path-a wildcard-path-b*)
 ▷ Translate the path of *path-or-stream* from *wildcard-path-a* into *wildcard-path-b*. Return new path.
- (**pathname** *path-or-stream*) ▷ Pathname of *path-or-stream*.
- (**logical-pathname** *logical-path-or-stream*)
 ▷ Logical pathname of *logical-path-or-stream*. Logical pathnames are represented as all-uppercase
 "[*host*[:];]{{dir}*⁺};}*{name}*⁺[.{type}*⁺]{LISP}
 [.{version}*⁺newestNEWEST]]".
- (**logical-pathname-translations** *logical-host*)
 ▷ List of (from-wildcard to-wildcard) translations for *logical-host*. **setfable**.
- (**load-logical-pathname-translations** *logical-host*)
 ▷ Load *logical-host*'s translations. Return NIL if already loaded; return T if successful.
- (**translate-logical-pathname** *path-or-stream*)
 ▷ Physical pathname corresponding to (possibly logical) pathname of *path-or-stream*.
- (**probe-file** *file*)
 (**truename** *file*)
 ▷ Canonical name of *file*. If *file* does not exist, return NIL/signal **file-error**, respectively.
- (**file-write-date** *file*) ▷ Time at which *file* was last written.
- (**file-author** *file*) ▷ Return name of file owner.
- (**file-length** *stream*) ▷ Return length of stream.
- (**rename-file** *foo bar*)
 ▷ Rename file *foo* to *bar*. Unspecified components of path *bar* default to those of *foo*. Return new pathname, old physical file name, and new physical file name.₂
- (**delete-file** *file*) ▷ Delete *file*. Return T.
- (**directory** *path*) ▷ List of pathnames matching *path*.
- (**ensure-directories-exist** *path* [:**verbose** *bool*])
 ▷ Create parts of *path* if necessary. Second return value is T₂ if something has been created.

14 Packages and Symbols

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

14.1 Predicates

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

14.2 Packages

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

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

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

`(mdefpackage foo` $\left. \begin{array}{l} (:nicknames\ nick^*)^* \\ (:documentation\ string) \\ (:intern\ interned-symbol^*)^* \\ (:use\ used-package^*)^* \\ (:import-from\ pkg\ imported-symbol^*)^* \\ (:shadowing-import-from\ pkg\ shd-symbol^*)^* \\ (:shadow\ shd-symbol^*)^* \\ (:export\ exported-symbol^*)^* \\ (:size\ int) \end{array} \right\}$ `)`

▷ 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.

`(fmake-package foo` $\left\{ \begin{array}{l} :nicknames\ (nick^*)\ \underline{NIL} \\ :use\ (used-package^*) \end{array} \right\}$ `)`

▷ Create package foo.

`(frename-package package new-name [new-nicknames NIL])`

▷ Rename *package*. Return renamed package.

`(min-package \widehat{foo})` ▷ Make package foo current.

$\left\{ \begin{array}{l} f\text{use-package} \\ f\text{unuse-package} \end{array} \right\}$ *other-packages* [*package* ν^* package*]

▷ Make exported symbols of *other-packages* available in *package*, or remove them from *package*, respectively. Return T.

`(fpackage-use-list package)`

`(fpackage-used-by-list package)`

▷ List of other packages used by/using *package*.

`(fdelete-package $\widetilde{package}$)`

▷ Delete *package*. Return T if successful.

ν^* package* common-lisp-user ▷ The current package.

`(flist-all-packages)` ▷ List of registered packages.

`(fpackage-name package)` ▷ Name of package.

`(fpackage-nicknames package)` ▷ Nicknames of package.

`(ffind-package name)` ▷ Package with *name* (case-sensitive).

`(ffind-all-symbols foo)`

▷ List of symbols foo from all registered packages.

$\left\{ \begin{array}{l} f\text{intern} \\ f\text{find-symbol} \end{array} \right\}$ *foo* [*package* ν^* 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).

`(funintern symbol [package ν^* package*])`

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

$\left\{ \begin{array}{l} f\text{import} \\ f\text{shadowing-import} \end{array} \right\}$ *symbols* [*package* ν^* package*]

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

`(fshadow symbols [package ν^* package*])`

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

(*f*package-shadowing-symbols *package*)

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

(*f*export *symbols* [*package*_{v*package*}])

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

(*f*unexport *symbols* [*package*_{v*package*}])

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

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

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

(*m*with-package-iterator (*foo packages* [:internal|:external|:inherited])

(declare \widehat{decl}^*) * *form*^{Pk})

▷ 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.

(*f*require *module* [*paths*_{NIL}])

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

(*f*provide *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*).

(*f*make-symbol *name*)

▷ Make fresh, uninterned symbol *name*.

(*f*gensym [*s*_g])

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

(*f*gentemp [*prefix*_g [*package*_{v*package*}]])

▷ Intern fresh symbol in package. Deprecated.

(*f*copy-symbol *symbol* [*props*_{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**.

$\left(\begin{array}{l} g\text{documentation} \\ (\text{setf } g\text{documentation}) \text{ new-doc} \end{array} \right) foo \left\{ \begin{array}{l} \text{'variable|'function} \\ \text{'compiler-macro} \\ \text{'method-combination} \\ \text{'structure|'type|'setf|T} \end{array} \right\}$

▷ 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**; **v*terminal-io***.

cnil_{c()}

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

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

(**fcompiled-function-p** *foo*)

▷ T if *foo* is of type **compiled-function**.

15.2 Compilation

(**fcompile** $\left\{ \begin{array}{l} \text{NIL } \textit{definition} \\ \textit{name} \\ \text{(setf } \textit{name}) \end{array} \right\}$ [*definition*])

▷ Return compiled function or replace *name*'s function definition with the compiled function. Return T in case of **warnings** or **errors**, and T in case of **warnings** or **errors** excluding **style-warnings**.

(**fcompile-file** *file* $\left\{ \begin{array}{l} \text{:output-file } \textit{out-path} \\ \text{:verbose } \textit{bool} \text{ } \boxed{\text{v*compile-verbose*}} \\ \text{:print } \textit{bool} \text{ } \boxed{\text{v*compile-print*}} \\ \text{:external-format } \textit{file-format} \text{ } \boxed{\text{:default}} \end{array} \right\}$)

▷ 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**.

(**fcompile-file-pathname** *file* [**:output-file** *path*] [*other-keyargs*])

▷ Pathname *fcompile-file* writes to if invoked with the same arguments.

(**fload** *path* $\left\{ \begin{array}{l} \text{:verbose } \textit{bool} \text{ } \boxed{\text{v*load-verbose*}} \\ \text{:print } \textit{bool} \text{ } \boxed{\text{v*load-print*}} \\ \text{:if-does-not-exist } \textit{bool} \text{ } \boxed{\text{T}} \\ \text{:external-format } \textit{file-format} \text{ } \boxed{\text{:default}} \end{array} \right\}$)

▷ Load source file or compiled file into Lisp environment. Return T if successful.

$\left. \begin{array}{l} \text{v*compile-file} \\ \text{v*load} \end{array} \right\} \left\{ \begin{array}{l} \text{pathname*} \boxed{\text{NIL}} \\ \text{truename*} \boxed{\text{NIL}} \end{array} \right\}$

▷ Input file used by *fcompile-file*/by *fload*.

$\left. \begin{array}{l} \text{v*compile} \\ \text{v*load} \end{array} \right\} \left\{ \begin{array}{l} \text{print*} \\ \text{verbose*} \end{array} \right\}$

▷ Defaults used by *fcompile-file*/by *fload*.

(**s**eval-when ({ {**:compile-toplevel**|**compile**}
 {**:load-toplevel**|**load**}
 {**:execute**|**eval**} }) *form*^{P*})

▷ 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*^{P*})

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

(**m**with-compilation-unit ([**:override** *bool*_{**NIL**}]) *form*^{P*})

▷ 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.

{**f**macroexpand-1
fmacroexpand } *form* [*environment*_{**NIL**}])

▷ Return macro expansion, once or entirely, respectively, of *form* and **T** if *form* was a macro form. Return form and **NIL** otherwise.²

v*macroexpand-hook*

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

(*m***trace** {*function*
{(**setf** *function*)}}*)

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

(*m***untrace** {*function*
{(**setf** *function*)}}*)

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

v*trace-output*

▷ Output stream *m***trace** and *m***time** send their output to.

(*m***step** *form*)

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

(*f***break** [*control arg**)

▷ Jump directly into debugger; return NIL. See page 38, *f***format**, for *control* and *args*.

(*m***time** *form*)

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

(*f***inspect** *foo*) ▷ Interactively give information about *foo*.

(*f***describe** *foo* [*stream* **v*standard-output***])

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

(*g***describe-object** *foo* [*stream*])

▷ Send information about *foo* to *stream*. Called by *f***describe**.

(*f***disassemble** *function*)

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

(*f***room** [{NIL|:default|T}|[:default]])

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

15.4 Declarations

(*f***proclaim** *decl*)

(*m***declaim** *decl**)

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

(**declare** *decl**)

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

(**declaration** *foo**)

▷ Make *foos* names of declarations.

(**dynamic-extent** *variable** (**function** *function*)*)

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

(**[type]** *type variable**)

(**ftype** *type function**)

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

(**{ignorable}** **{ignore}** {*var*
{(**function** *function*)}}*)

▷ 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 { compilation-speed | (compilation-speed n_{3})
 debug | (debug n_{3})
 safety | (safety n_{3})
 space | (space n_{3})
 speed | (speed n_{3}) })

▷ 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.

({ *f*lisp-implementation } { type
*f*software } { version }
 { *f*machine })

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

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

Index

- " 35
' 35
(35
) 46
) 35
* 3, 32, 33, 43, 47
** 43, 47
*** 47
*BREAK-
 ON-SIGNALS* 31
*COMPILE-FILE-
 PATHNAME* 46
*COMPILE-FILE-
 TRUENAME* 46
COMPILE-PRINT 46
COMPILE-VERBOSE 46
DEBUG-IO 42
DEBUGGER-HOOK 31
*DEFAULT-
 PATHNAME-
 DEFAULTS* 42
ERROR-OUTPUT 42
FEATURES 36
GENSYM-COUNTER 45
LOAD-PATHNAME 46
LOAD-PRINT 46
LOAD-TRUENAME 46
LOAD-VERBOSE 46
*MACROEXPAND-
 HOOK* 48
MODULES 45
PACKAGE 44
PRINT-ARRAY 37
PRINT-BASE 37
PRINT-CASE 38
PRINT-CIRCLE 38
PRINT-ESCAPE 38
PRINT-GENSYM 38
PRINT-LENGTH 38
PRINT-LEVEL 38
PRINT-LINES 38
*PRINT-
 MISER-WIDTH* 38
*PRINT-PPRINT-
 DISPATCH* 38
PRINT-PRETTY 38
PRINT-RADIX 38
PRINT-READABLY 38
*PRINT-
 RIGHT-MARGIN* 38
QUERY-IO 42
RANDOM-STATE 4
READ-BASE 34
*READ-DEFAULT-
 FLOAT-FORMAT* 34
READ-EVAL 35
READ-SUPPRESS 34
READTABLE 34
STANDARD-INPUT 42
*STANDARD-
 OUTPUT* 42
TERMINAL-IO 42
TRACE-OUTPUT 48
+ 3, 28, 47
++ 47
+++ 47
. 35
.. 35
.@ 35
- 3, 47
. 35
/ 3, 35, 47
// 47
/// 47
/= 3
: 44
:: 44
:ALLOW-OTHER-KEYS 20
; 35
< 3
<= 3
= 3, 22, 24
> 3
>= 3
\ 36
40
#\ 35
#' 35
#(35
#* 35
#+ 36
#- 36
#. 35
#: 35
#< 35
#= 35
#A 35
#B 35
#C(35
#O 35
#P 35
#R 35
#S(35
#X 35
35
#| |# 35
&ALLOW-
 OTHER-KEYS 20
&AUX 20
&BODY 20
&ENVIRONMENT 20
&KEY 20
&OPTIONAL 20
&REST 20
&WHOLE 20
~(~) 39
~* 40
~/ / 40
~< ~:> 39
~< ~> 39
~? 40
~A 38
~B 39
~C 39
~D 39
~E 39
~F 39
~G 39
~I 40
~O 39
~P 39
~R 38
~S 38
~T 40
~W 40
~X 39
~[~] 40
~\$ 39
~% 39
~& 39
~^ 40
~_ 39
~| 39
~{ ~} 40
~. 39
~< 39
~ 35
| | 36
1+ 3
1- 3
ABORT 30
ABOVE 22
ABS 4
ACONS 10
ACOS 3
ACOSH 4
ACROSS 24
ADD-METHOD 27
ADJOIN 9
ADJUST-ARRAY 11
ADJUSTABLE-
 ARRAY-P 11
ALLOCATE-INSTANCE 26
ALPHA-CHAR-P 7
ALPHANUMERICP 7
ALWAYS 25
AND 21, 22, 24, 28, 33, 36
APPEND 10, 24, 28
APPENDING 24
APPLY 18
APROPPOS 47
APROPPOS-LIST 47
AREF 11
ARITHMETIC-ERROR 32
ARITHMETIC-ERROR-
 OPERANDS 30
ARITHMETIC-ERROR-
 OPERATION 30
ARRAY 32
ARRAY-DIMENSION 11
ARRAY-DIMENSION-
 LIMIT 12
ARRAY-DIMENSIONS 11
ARRAY-
 DISPLACEMENT 11
ARRAY-
 ELEMENT-TYPE 31
ARRAY-HAS-
 FILL-POINTER-P 11
ARRAY-IN-BOUNDS-P 11
ARRAY-RANK 11
ARRAY-RANK-LIMIT 12
ARRAY-ROW-
 MAJOR-INDEX 11
ARRAY-TOTAL-SIZE 11
ARRAY-TOTAL-
 SIZE-LIMIT 12
ARRAYP 11
AS 22
ASH 6
ASIN 3
ASINH 4
ASSERT 29
ASSOC 10
ASSOC-IF 10
ASSOC-IF-NOT 10
ATAN 3
ATANH 4
ATOM 9, 32
BASE-CHAR 32
BASE-STRING 32
BEING 24
BELOW 22
BIGNUM 32
BIT 12, 32
BIT-AND 12
BIT-ANDC1 12
BIT-ANDC2 12
BIT-EQV 12
BIT-IOR 12
BIT-NAND 12
BIT-NOR 12
BIT-NOT 12
BIT-ORC1 12
BIT-ORC2 12
BIT-VECTOR 32
BIT-VECTOR-P 11
BIT-XOR 12
BLOCK 21
BOOLE 5
BOOLE-1 5
BOOLE-2 5
BOOLE-AND 5
BOOLE-ANDC1 5
BOOLE-ANDC2 5
BOOLE-C1 5
BOOLE-C2 5
BOOLE-CLR 5
BOOLE-EQV 5
BOOLE-IOR 5
BOOLE-NAND 5
BOOLE-NOR 5
BOOLE-ORC1 5
BOOLE-ORC2 5
BOOLE-SET 5
BOOLE-XOR 5
BOOLEAN 32
BOTH-CASE-P 7
BOUNDP 16
BREAK 48
BROADCAST-STREAM 32
BROADCAST-
 STREAM-STREAMS 41
BUILT-IN-CLASS 32
BUTLAST 9
BY 22
BYTE 6
BYTE-POSITION 6
BYTE-SIZE 6
CAAR 9
CADR 9
CALL-ARGUMENTS-
 LIMIT 19
CALL-METHOD 28
CALL-NEXT-METHOD 27
CAR 9
CASE 21
CATCH 22
CCASE 21
CDAR 9
CDDR 9
CDR 9
CEILING 4
CELL-ERROR 32
CELL-ERROR-NAME 31
CERROR 29
CHANGE-CLASS 26
CHAR 8
CHAR-CODE 7
CHAR-CODE-LIMIT 7
CHAR-DOWNCASE 7
CHAR-EQUAL 7
CHAR-GREATERP 7
CHAR-INT 7
CHAR-LESSP 7
CHAR-NAME 7
CHAR-NOT-EQUAL 7
CHAR-NOT-GREATERP 7
CHAR-NOT-LESSP 7
CHAR-UPCASE 7
CHAR/= 7
CHAR< 7
CHAR<= 7
CHAR= 7
CHAR> 7
CHAR>= 7
CHARACTER 7, 32, 35
CHARACTERP 7
CHECK-TYPE 31
CIS 4
CL 46
CL-USER 46
CLASS 32
CLASS-NAME 25
CLASS-OF 25
CLEAR-INPUT 41
CLEAR-OUTPUT 41
CLOSE 41
CLQR 1
CLRHASH 15
CODE-CHAR 7
COERCE 31
COLLECT 24
COLLECTING 24
COMMON-LISP 46
COMMON-LISP-USER 46
COMPILATION-SPEED 49
COMPILE 46
COMPILE-FILE 46
COMPILE-
 FILE-PATHNAME 46
COMPILED-FUNCTION 32
COMPILED-
 FUNCTION-P 46
COMPILER-MACRO 45
COMPILER-MACRO-
 FUNCTION 47
COMPLEMENT 18
COMPLEX 4, 32, 35
COMPLEXP 3
COMPUTE-
 APPLICABLE-
 METHODS 27
COMPUTE-RESTARTS 30
CONCATENATE 13
CONCATENATED-
 STREAM 32
CONCATENATED-
 STREAM-STREAMS 41
COND 21
CONDITION 32
CONJUGATE 4
CONS 9, 32
CONSP 8
CONSTANTLY 19
CONSTANTP 17
CONTINUE 30
CONTROL-ERROR 32
COPY-ALIST 10
COPY-LIST 10
COPY-PPRINT-
 DISPATCH 38
COPY-READTABLE 34
COPY-SEQ 15
COPY-STRUCTURE 16
COPY-SYMBOL 45
COPY-TREE 11
COS 3
COSH 4
COUNT 13, 24
COUNT-IF 13
COUNT-IF-NOT 13
COUNTING 24
CTYPECASE 31
DEBUG 49
DEC 3
DECLAIM 48
DECLARATION 48
DECLARE 48
DECODE-FLOAT 6
DECODE-UNIVERSAL-
 TIME 49
DEFCCLASS 25
DEFCONSTANT 17
DEFGENERIC 26
DEFINE-COMPILER-
 MACRO 19
DEFINE-CONDITION 29
DEFINE-METHOD-
 COMBINATION 28
DEFINE-
 MODIFY-MACRO 20
DEFINE-
 SETF-EXPANDER 20
DEFINE-
 SYMBOL-MACRO 19
DEFMACRO 19
DEFMETHOD 27
DEFPACKAGE 44
DEFPARAMETER 17
DEFSETF 20
DEFSTRUCT 16
DEFTYPE 33
DEFUN 18
DEFVAR 17
DELETE 14
DELETE-DUPLICATES 14
DELETE-FILE 43
DELETE-IF 14
DELETE-IF-NOT 14
DELETE-PACKAGE 44
DENOMINATOR 4
DEPOSIT-FIELD 6
DESCRIBE 48
DESCRIBE-OBJECT 48
DESTRUCTURING-
 BIND 18
DIGIT-CHAR 7

- NAME-CHAR 7
NAMED 22
NAMESTRING 43
NBUTLAST 9
NCONC 10, 24, 28
NCONCING 24
NEVER 25
NEWLINE 7
NEXT-METHOD-P 26
NIL 2, 46
NINTERSECTION 11
NINTH 9
NO-APPLICABLE-METHOD 27
NO-NEXT-METHOD 27
NOT 16, 33, 36
NOTANY 12
NOTEVERY 12
NOTINLINE 48
NRECONC 10
NREVERSE 13
NSET-DIFFERENCE 11
NSET-EXCLUSIVE-OR 11
NSTRING-CAPITALIZE 8
NSTRING-DOWNCASE 8
NSTRING-UPCASE 8
NSUBLIS 11
NSUBST 10
NSUBST-IF 10
NSUBST-IF-NOT 10
NSUBSTITUTE 14
NSUBSTITUTE-IF 14
NSUBSTITUTE-IF-NOT 14
NTH 9
NTH-VALUE 18
NTHCDR 9
NULL 8, 32
NUMBER 32
NUMBERP 3
NUMERATOR 4
NUNION 11

ODDP 3
OF 24
OF-TYPE 22
ON 22
OPEN 40
OPEN-STREAM-P 33
OPTIMIZE 49
OR 21, 28, 33, 36
OTHERWISE 21, 31
OUTPUT-STREAM-P 33

PACKAGE 32
PACKAGE-ERROR 32
PACKAGE-ERROR-PACKAGE 31
PACKAGE-NAME 44
PACKAGE-NICKNAMES 44
PACKAGE-SHADOWING-SYMBOLS 45
PACKAGE-USE-LIST 44
PACKAGE-USED-BY-LIST 44
PACKAGEP 43
PAIRLIS 10
PARSE-ERROR 32
PARSE-INTEGER 8
PARSE-NAMESTRING 42
PATHNAME 32, 43
PATHNAME-DEVICE 42
PATHNAME-DIRECTORY 42
PATHNAME-HOST 42
PATHNAME-MATCH-P 33
PATHNAME-NAME 42
PATHNAME-TYPE 42
PATHNAME-VERSION 42
PATHNAMEP 33
PEEK-CHAR 34
PHASE 4
PI 3
PLUSP 3
POP 9
POSITION 13
POSITION-IF 14
POSITION-IF-NOT 14
PPRINT 36
PPRINT-DISPATCH 38
PPRINT-EXIT-IF-LIST-EXHAUSTED 37
PPRINT-FILL 37
PPRINT-INDENT 37
PPRINT-LINEAR 37
PPRINT-LOGICAL-BLOCK 37
PPRINT-NEWLINE 37
PPRINT-POP 37
PPRINT-TAB 37
PPRINT-TABULAR 37
PRESENT-SYMBOL 24
PRESENT-SYMBOLS 24
PRIN1 36
PRIN1-TO-STRING 36
PRINC 36
PRINC-TO-STRING 36
PRINT 36
PRINT-
NOT-READABLE 32
PRINT-NOT-
READABLE-OBJECT 31
PRINT-OBJECT 36
PRINT-UNREADABLE-OBJECT 36
PROBE-FILE 43
PROCLAIM 48
PROG 21
PROG1 21
PROG2 21
PROG* 21
PROGN 21, 28
PROGRAM-ERROR 32
PROGV 17
PROVIDE 45
PSETF 17
PSETQ 17
PUSH 10
PUSHNEW 10

QUOTE 35, 47

RANDOM 4
RANDOM-STATE 32
RANDOM-STATE-P 3
RASSOC 10
RASSOC-IF 10
RASSOC-IF-NOT 10
RATIO 32, 35
RATIONAL 4, 32
RATIONALIZE 4
RATIONALP 3
READ 33
READ-BYTE 34
READ-CHAR 34
READ-CHAR-NO-HANG 34
READ-
DELIMITED-LIST 34
READ-FROM-STRING 33
READ-LINE 34
READ-PRESERVING-WHITESPACE 33
READ-SEQUENCE 34
READER-ERROR 32
READTABLE 32
READTABLE-CASE 34
READTABLEP 33
REAL 32
REALP 3
REALPART 4
REDUCE 15
REINITIALIZE-
INSTANCE 25
REM 4
REMF 17
REMHASH 15
REMOVE 14
REMOVE-DUPLICATES 14
REMOVE-IF 14
REMOVE-IF-NOT 14
REMOVE-METHOD 27
REMPROP 17
RENAME-FILE 43
RENAME-PACKAGE 44
REPEAT 24
REPLACE 14
REQUIRE 45
REST 9
RESTART 32
RESTART-BIND 30
RESTART-CASE 30
RESTART-NAME 30
RETURN 21, 24
RETURN-FROM 21
REVAPPEND 10
REVERSE 13
ROOM 48
ROTATEF 17
ROUND 4
ROW-MAJOR-AREF 11
RPLACA 9
RPLACD 9

SAFETY 49
SATISFIES 33
SBIT 12
SCALE-FLOAT 6
SCHAR 8
SEARCH 14
SECOND 9
SEQUENCE 32
SERIOUS-CONDITION 32
SET 17
SET-DIFFERENCE 11
SET-
DISPATCH-MACRO-
CHARACTER 35
SET-EXCLUSIVE-OR 11
SET-MACRO-
CHARACTER 34
SET-PPRINT-
DISPATCH 38
SET-SYNTAX-
FROM-CHAR 34
SETF 17, 45
SETQ 17
SEVENTH 9
SHADOW 44
SHADOWING-IMPORT 44
SHARED-INITIALIZE 26
SHIFTF 17
SHORT-FLOAT 32, 35
SHORT-
FLOAT-EPSILON 6
SHORT-FLOAT-
NEGATIVE-EPSILON 6
SHORT-SITE-NAME 49
SIGNAL 29
SIGNED-BYTE 32
SIGNUM 4
SIMPLE-ARRAY 32
SIMPLE-BASE-STRING 32
SIMPLE-BIT-VECTOR 32
SIMPLE-
BIT-VECTOR-P 11
SIMPLE-CONDITION 32
SIMPLE-CONDITION-
FORMAT-
ARGUMENTS 31
SIMPLE-CONDITION-
FORMAT-CONTROL 31
SIMPLE-ERROR 32
SIMPLE-STRING 32
SIMPLE-STRING-P 8
SIMPLE-TYPE-ERROR 32
SIMPLE-VECTOR 32
SIMPLE-VECTOR-P 11
SIMPLE-WARNING 32
SIN 3
SINGLE-FLOAT 32, 35
SINGLE-
FLOAT-EPSILON 6
SINGLE-FLOAT-
NEGATIVE-EPSILON 6
SINH 4
SIXTH 9
SLEEP 22
SLOT-BOUND 25
SLOT-EXISTS-P 25
SLOT-MAKUNBOUND 25
SLOT-MISSING 26
SLOT-UNBOUND 26
SLOT-VALUE 25
SOFTWARE-TYPE 49
SOFTWARE-VERSION 49
SOME 12
SORT 13
SPACE 7, 49
SPECIAL 49
SPECIAL-OPERATOR-P 46
SPEED 49
SQRT 3
STABLE-SORT 13
STANDARD 28
STANDARD-CHAR 7, 32
STANDARD-CHAR-P 7
STANDARD-CLASS 32
STANDARD-GENERIC-FUNCTION 32
STANDARD-METHOD 32
STANDARD-OBJECT 32
STEP 48
STORAGE-CONDITION 32
STORE-VALUE 30
STREAM 32
STREAM-
ELEMENT-TYPE 31
STREAM-ERROR 32
STREAM-
ERROR-STREAM 31
STREAM-EXTERNAL-
FORMAT 42
STREAMP 33
STRING 8, 32
STRING-CAPITALIZE 8
STRING-DOWNCASE 8
STRING-EQUAL 8
STRING-GREATERP 8
STRING-LEFT-TRIM 8
STRING-LESSP 8
STRING-NOT-EQUAL 8
STRING-
NOT-GREATERP 8
STRING-NOT-LESSP 8
STRING-RIGHT-TRIM 8
STRING-STREAM 32
STRING-TRIM 8
STRING-UPCASE 8
STRING/= 8
STRING< 8
STRING<= 8
STRING= 8
STRING> 8
STRING>= 8
STRINGP 8
STRUCTURE 45
STRUCTURE-CLASS 32
STRUCTURE-OBJECT 32
STYLE-WARNING 32
SUBLIS 11
SUBSEQ 13
SUBSETP 9
SUBST 10
SUBST-IF 10
SUBST-IF-NOT 10
SUBSTITUTE 14
SUBSTITUTE-IF 14
SUBSTITUTE-IF-NOT 14
SUBTYPEP 31
SUM 24
SUMMING 24
SVREF 12
SXHASH 15
SYMBOL 24, 32, 45
SYMBOL-FUNCTION 45
SYMBOL-MACROLET 20
SYMBOL-NAME 45
SYMBOL-PACKAGE 45
SYMBOL-PLIST 45
SYMBOL-VALUE 45
SYMBOLP 43
SYMBOLS 24
SYNONYM-STREAM 32
SYNONYM-STREAM-SYMBOL 41

T 2, 32, 46
TAGBODY 21
TAILP 9
TAN 3
TANH 4
TENTH 9
TERPRI 36
THE 24, 31
THEN 24
THEREIS 25
THIRD 9
THROW 22
TIME 48
TO 22
TRACE 48
TRANSLATE-LOGICAL-PATHNAME 43
TRANSLATE-PATHNAME 43
TREE-EQUAL 10
TRUENAME 43
TRUNCATE 4
TWO-WAY-STREAM 32
TWO-WAY-STREAM-
INPUT-STREAM 41
TWO-WAY-STREAM-
OUTPUT-STREAM 41
TYPE 45, 48
TYPE-ERROR 32
TYPE-ERROR-DATUM 31
TYPE-ERROR-
EXPECTED-TYPE 31
TYPE-OF 31
TYPECASE 31
TYPEP 31

UNBOUND-SLOT 32
UNBOUND-
SLOT-INSTANCE 31
UNBOUND-VARIABLE 32
UNDEFINED-FUNCTION 32
UNEXPORT 45
UNINTERN 44
UNION 11
UNLESS 21, 24
UNREAD-CHAR 34
UNSIGN-BYTE 32
UNTIL 24
UNTRACE 48
UNUSE-PACKAGE 44
UNWIND-PROTECT 21
UPDATE-INSTANCE-
FOR-DIFFERENT-
CLASS 26
UPDATE-INSTANCE-
FOR-REDEFINED-
CLASS 26
UPFROM 22
UPGRADED-ARRAY-
ELEMENT-TYPE 31
UPGRADED-
COMPLEX-
PART-TYPE 6
UPPER-CASE-P 7
UPTO 22
USE-PACKAGE 44
USE-VALUE 30

USER-HOMEDIR-PATHNAME 42
USING 24

V 40
VALUES 18, 33
VALUES-LIST 18
VARIABLE 45
VECTOR 12, 32
VECTOR-POP 12
VECTOR-PUSH 12
VECTOR-PUSH-EXTEND 12
VECTORP 11

WARN 29
WARNING 32
WHEN 21, 24
WHILE 24
WILD-PATHNAME-P 33
WITH 22
WITH-ACCESSORS 25
WITH-COMPIATION-UNIT 47
WITH-CONDITION-RESTARTS 30
WITH-HASH-TABLE-ITERATOR 15
WITH-INPUT-
FROM-STRING 41
WITH-OPEN-FILE 41
WITH-OPEN-STREAM 41
WITH-OUTPUT-TO-STRING 42
WITH-PACKAGE-ITERATOR 45
WITH-SIMPLE-RESTART 30
WITH-SLOTS 25
WITH-STANDARD-IO-SYNTAX 33
WRITE 37

WRITE-BYTE 36
WRITE-CHAR 36
WRITE-LINE 36
WRITE-SEQUENCE 36
WRITE-STRING 36
WRITE-TO-STRING 37

Y-OR-N-P 33
YES-OR-NO-P 33

ZEROP 3

