

- (*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*.
- $$\left\{ \begin{array}{l} \{f\text{round}\} \{f\text{round}\} \\ \{f\text{floor}\} \{f\text{floor}\} \\ \{f\text{ceiling}\} \{f\text{ceiling}\} \\ \{f\text{truncate}\} \{f\text{truncate}\} \end{array} \right\} n \ [d_{\square}]$$
 ▷ Return as **integer** or **float**, respectively, n/d rounded, or rounded towards $-\infty$, $+\infty$, or 0, respectively; and remainder.
- $$\left\{ \begin{array}{l} f\text{mod} \\ f\text{rem} \end{array} \right\} n \ d$$
 ▷ 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} [*NTL*]])
 ▷ 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* \square .*OFF*])
 ▷ Convert *real* into float with type of *prototype*.

1.3 Logic Functions

Negative integers are used in two's complement representation.

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

Quick Reference

Common lisp

Bert Burgemeister

Contents

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

Typographic Conventions

name; *f***name**; *g***name**; *m***name**; *s***name**; *v****name***; *c***name**
 ▷ Symbol defined in Common Lisp; *esp.* function, generic function, macro, special operator, variable, constant.

them ▷ Placeholder for actual code.
me ▷ Literal text.
 $[foo_{\overline{bar}}]$ ▷ 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.

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

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

\widehat{foo} ▷ Argument *foo* is not evaluated.
 \widehat{bar} ▷ Argument *bar* is possibly modified.
 foo^R ▷ *foo** is evaluated as in *sprogn*; see page 20.
 $\frac{foo}{2}; \frac{bar}{2}; \frac{baz}{n}$ ▷ Primary, secondary, and *n*th return value.
T; **NIL** ▷ **t**, or truth in general; and **nil** or **()**.

1 Numbers

1.1 Predicates

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

$(f > number^+)$
 $(f \geq number^+)$
 $(f < number^+)$
 $(f \leq number^+)$
 ▷ Return **T** if *numbers* are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

$(f \text{minusp } a)$
 $(f \text{zerop } a)$ ▷ **T** if $a < 0$, $a = 0$, or $a > 0$, respectively.
 $(f \text{plusp } a)$

$(f \text{evenp } int)$ ▷ **T** if *int* is even or odd, respectively.
 $(f \text{oddp } int)$

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

1.2 Numeric Functions

$(f + a_{\overline{n}}^*)$ ▷ Return $\sum a$ or $\prod a$, respectively.
 $(f * a_{\overline{n}}^*)$

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

$(f 1+ a)$ ▷ Return $a + 1$ or $a - 1$, respectively.
 $(f 1- a)$

$\left\{ \begin{matrix} m \text{incf} \\ m \text{decf} \end{matrix} \right\} \widetilde{place} [\delta_{\overline{n}}]$
 ▷ Increment or decrement the value of *place* by *delta*. Return new value.

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

$(f \text{log } a \ [b_{\overline{n}}])$ ▷ Return $\log_b a$ or, without *b*, $\ln a$.

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

$(f \text{lcm } integer^*_{\overline{n}})$
 $(f \text{gcd } integer^*_{\overline{n}})$
 ▷ Least common multiple or greatest common denominator, respectively, of *integers*. (**gcd**) returns 0.

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

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

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

$(f \text{atan } a \ [b_{\overline{n}}])$ ▷ $\arctan \frac{a}{b}$ in radians.

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

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

(*f*parse-integer *string* $\left\{ \begin{array}{l} \text{:start } \text{start}_{\text{[0]}} \\ \text{:end } \text{end}_{\text{[NIL]}} \\ \text{:radix } \text{int}_{\text{[10]}} \\ \text{:junk-allowed } \text{bool}_{\text{[NIL]}} \end{array} \right\}$)
 ▷ Return integer parsed from *string* and index of parse end.

4 Conses

4.1 Predicates

(*f*consp *foo*)
 (*f*listp *foo*)
 ▷ Return T if *foo* is of indicated type.

(*f*endp *list*)
 (*f*null *foo*)
 ▷ Return T if *list/foo* is NIL.

(*f*atom *foo*)
 ▷ Return T if *foo* is not a **cons**.

(*f*tailp *foo* *list*)
 ▷ Return T if *foo* is a tail of *list*.

(*f*member *foo* *list* $\left\{ \begin{array}{l} \text{:test } \text{function}_{\text{[#\text{eq}]}} \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\}$)
 ▷ 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* $\left\{ \begin{array}{l} \text{:key } \text{function} \end{array} \right\}$
 ▷ Return tail of list starting with its first element satisfying *test*. Return NIL if there is no such element.

(*f*subsetp *list-a* *list-b* $\left\{ \begin{array}{l} \text{:test } \text{function}_{\text{[#\text{eq}]}} \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\}$)
 ▷ Return T if *list-a* is a subset of *list-b*.

4.2 Lists

(*f*cons *foo* *bar*)
 ▷ Return new cons (*foo* . *bar*).

(*f*list *foo**)
 ▷ Return list of foos.

(*f*list* *foo**)
 ▷ Return list of foos with last *foo* becoming cdr of last cons. Return foo if only one *foo* given.

(*f*make-list *num* $\left\{ \begin{array}{l} \text{:initial-element } \text{foo}_{\text{[NIL]}} \end{array} \right\}$)
 ▷ New list with *num* elements set to *foo*.

(*f*list-length *list*)
 ▷ Length of *list*; NIL for circular *list*.

(*f*car *list*)
 ▷ Car of *list* or NIL if *list* is NIL. **setfable**.

(*f*cdr *list*)
 (*f*rest *list*)
 ▷ Cdr of *list* or NIL if *list* is NIL. **setfable**.

(*f*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**.

(*f*nth *n* *list*)
 ▷ Zero-indexed nth element of *list*. **setfable**.

(*f*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**.

(*f*last *list* [*num*])
 ▷ Return list of last num conses of *list*.

cboole-eqv ▷ $\text{int-}a \equiv \text{int-}b$.
 cboole-and ▷ $\text{int-}a \wedge \text{int-}b$.
 cboole-andc1 ▷ $\neg \text{int-}a \wedge \text{int-}b$.
 cboole-andc2 ▷ $\text{int-}a \wedge \neg \text{int-}b$.
 cboole-nand ▷ $\neg(\text{int-}a \wedge \text{int-}b)$.
 cboole-ior ▷ $\text{int-}a \vee \text{int-}b$.
 cboole-orc1 ▷ $\neg \text{int-}a \vee \text{int-}b$.
 cboole-orc2 ▷ $\text{int-}a \vee \neg \text{int-}b$.
 cboole-xor ▷ $\neg(\text{int-}a \equiv \text{int-}b)$.
 cboole-nor ▷ $\neg(\text{int-}a \vee \text{int-}b)$.

(*f*lognot *integer*)
 ▷ $\neg \text{integer}$.

(*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*)
 (*f*logandc2 *int-a* *int-b*)
 (*f*lognand *int-a* *int-b*)
 ▷ $\neg \text{int-}a \wedge \text{int-}b$.
 ▷ $\text{int-}a \wedge \neg \text{int-}b$.
 ▷ $\neg(\text{int-}a \wedge \text{int-}b)$.

(*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*)
 (*f*logorc2 *int-a* *int-b*)
 (*f*lognor *int-a* *int-b*)
 ▷ $\neg \text{int-}a \vee \text{int-}b$.
 ▷ $\text{int-}a \vee \neg \text{int-}b$.
 ▷ $\neg(\text{int-}a \vee \text{int-}b)$.

(*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**.

$\left\{ \begin{array}{l} \text{fdeposit-field} \\ \text{fdpb} \end{array} \right\}$ *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^{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.

$(\text{fdecode-float } n)$
 $(\text{finteger-decode-float } n)$
 ▷ Return significand, exponent, and sign of float n .

$(\text{fscale-float } n \ [i])$ ▷ With n 's radix b , return nb^i .

$(\text{ffloat-radix } n)$
 $(\text{ffloat-digits } n)$
 $(\text{ffloat-precision } n)$
 ▷ Radix, number of digits in that radix, or precision in that radix, respectively, of float n .

$(\text{fupgraded-complex-part-type } foo \ [environment_{\text{NIL}}])$
 ▷ Type of most specialized **complex** number able to hold parts of type foo .

2 Characters

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

$(\text{fcharacterp } foo)$
 $(\text{fstandard-char-p } char)$
 ▷ T if argument is of indicated type.

$(\text{fgraphic-char-p } character)$
 $(\text{falpha-char-p } character)$
 $(\text{falphanumericp } character)$
 ▷ T if $character$ is visible, alphabetic, or alphanumeric, respectively.

$(\text{fupper-case-p } character)$
 $(\text{flower-case-p } character)$
 $(\text{fboth-case-p } character)$
 ▷ Return T if $character$ is uppercase, lowercase, or able to be in another case, respectively.

$(\text{fdigit-char-p } character \ [radix_{\text{10}}])$
 ▷ Return its weight if $character$ is a digit, or NIL otherwise.

$(\text{fchar= } character^+)$
 $(\text{fchar/= } character^+)$
 ▷ Return T if all $characters$, or none, respectively, are equal.

$(\text{fchar-equal } character^+)$
 $(\text{fchar-not-equal } character^+)$
 ▷ Return T if all $characters$, or none, respectively, are equal ignoring case.

$(\text{fchar> } character^+)$
 $(\text{fchar>= } character^+)$
 $(\text{fchar< } character^+)$
 $(\text{fchar<= } character^+)$
 ▷ Return T if $characters$ are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

$(\text{fchar-greaterp } character^+)$
 $(\text{fchar-not-lessp } character^+)$
 $(\text{fchar-lessp } character^+)$
 $(\text{fchar-not-greaterp } character^+)$
 ▷ Return T if $characters$ are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.

$(\text{fchar-upcase } character)$
 $(\text{fchar-downcase } character)$
 ▷ Return corresponding uppercase/lowercase character, respectively.

$(\text{fdigit-char } i \ [radix_{\text{10}}])$ ▷ Character representing digit i .

$(\text{fchar-name } char)$ ▷ $char$'s name if any, or NIL.

$(\text{fname-char } foo)$ ▷ Character named foo if any, or NIL.

$(\text{fchar-int } character)$ ▷ Code of $character$.
 $(\text{fchar-code } character)$

$(\text{fcode-char } code)$ ▷ Character with $code$.

cchar-code-limit ▷ Upper bound of $(\text{fchar-code } char)$; ≥ 96 .

$(\text{fcharacter } c)$ ▷ Return #\c.

3 Strings

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

$(\text{fstringp } foo)$
 $(\text{fsimple-string-p } foo)$ ▷ T if foo is of indicated type.

$\left\{ \begin{array}{l} \text{fstring=} \\ \text{fstring-equal} \end{array} \right\} foo \ bar \left\{ \begin{array}{l} \text{:start1 } start\text{-}foo_{\text{0}} \\ \text{:start2 } start\text{-}bar_{\text{0}} \\ \text{:end1 } end\text{-}foo_{\text{NIL}} \\ \text{:end2 } end\text{-}bar_{\text{NIL}} \end{array} \right\}$
 ▷ Return T if subsequences of foo and bar are equal. Obey/ignore, respectively, case.

$\left\{ \begin{array}{l} \text{fstring}\{/= \text{ } | \text{-not-equal}\} \\ \text{fstring}\{> \text{ } | \text{-greaterp}\} \\ \text{fstring}\{>= \text{ } | \text{-not-lessp}\} \\ \text{fstring}\{< \text{ } | \text{-lessp}\} \\ \text{fstring}\{<= \text{ } | \text{-not-greaterp}\} \end{array} \right\} foo \ bar \left\{ \begin{array}{l} \text{:start1 } start\text{-}foo_{\text{0}} \\ \text{:start2 } start\text{-}bar_{\text{0}} \\ \text{:end1 } end\text{-}foo_{\text{NIL}} \\ \text{:end2 } end\text{-}bar_{\text{NIL}} \end{array} \right\}$
 ▷ 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.

$(\text{fmake-string } size \left\{ \begin{array}{l} \text{:initial-element } char \\ \text{:element-type } type_{\text{character}} \end{array} \right\})$
 ▷ Return string of length $size$.

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

$\left\{ \begin{array}{l} \text{fnstring-capitalize} \\ \text{fnstring-upcase} \\ \text{fnstring-downcase} \end{array} \right\} \widetilde{string} \left\{ \begin{array}{l} \text{:start } start_{\text{0}} \\ \text{:end } end_{\text{NIL}} \end{array} \right\}$
 ▷ Convert $string$ into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

$\left\{ \begin{array}{l} \text{fstring-trim} \\ \text{fstring-left-trim} \\ \text{fstring-right-trim} \end{array} \right\} char\text{-}bag \ string$
 ▷ Return string with all characters in sequence $char\text{-}bag$ removed from both ends, from the beginning, or from the end, respectively.

6 Sequences

6.1 Sequence Predicates

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

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

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

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

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

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

6.2 Sequence Functions

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

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

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

▷ Return concatenated sequence of *type*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$(\text{f nreverse } \text{sequence})$ ▷ Return sequence in reverse order.

$\left\{ \begin{array}{l} \text{f butlast } \text{list} \\ \text{f nbutlast } \widetilde{\text{list}} \end{array} \right\} [\text{num}_{\text{0}}])$ ▷ *list* excluding last *num* conses.

$\left\{ \begin{array}{l} \text{f rplaca} \\ \text{f rplacd} \end{array} \right\} \widetilde{\text{cons } \text{object}}$

▷ Replace *car*, or *cdr*, respectively, of *cons* with *object*.

$(\text{f ldiff } \text{list } \text{foo})$

▷ If *foo* is a tail of *list*, return preceding part of *list*. Otherwise return *list*.

$(\text{f adjoin } \text{foo } \text{list } \left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{:test function}_{\text{#eq}} \\ \text{:test-not function} \end{array} \right\} \\ \text{:key function} \end{array} \right\})$

▷ Return *list* if *foo* is already member of *list*. If not, return $(\text{f cons } \text{foo } \text{list})$.

$(\text{f pop } \widetilde{\text{place}})$

▷ Set *place* to $(\text{f cdr } \text{place})$, return $(\text{f car } \text{place})$.

$(\text{f push } \text{foo } \widetilde{\text{place}})$

▷ Set *place* to $(\text{f cons } \text{foo } \text{place})$.

$(\text{f pushnew } \text{foo } \widetilde{\text{place}} \left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{:test function}_{\text{#eq}} \\ \text{:test-not function} \end{array} \right\} \\ \text{:key function} \end{array} \right\})$

▷ Set *place* to $(\text{f adjoin } \text{foo } \text{place})$.

$(\text{f append } [\text{proper-list}^* \text{foo}_{\text{NIL}}])$

$(\text{f nconc } [\text{non-circular-list}^* \text{foo}_{\text{NIL}}])$

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

$(\text{f revappend } \text{list } \text{foo})$

$(\text{f nreconc } \text{list } \text{foo})$

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

$\left\{ \begin{array}{l} \text{f mapcar} \\ \text{f maplist} \end{array} \right\} \text{ function } \text{list}^+$

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

$\left\{ \begin{array}{l} \text{f mapcan} \\ \text{f mapcon} \end{array} \right\} \text{ function } \widetilde{\text{list}^+}$

▷ 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.

$\left\{ \begin{array}{l} \text{f mapc} \\ \text{f mapl} \end{array} \right\} \text{ function } \text{list}^+$

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

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

4.3 Association Lists

$(\text{f pairlis } \text{keys } \text{values } [\text{alist}_{\text{NIL}}])$

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

$(\text{f acons } \text{key } \text{value } \text{alist})$

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

$\left\{ \begin{array}{l} \text{f assoc} \\ \text{f rassoc} \end{array} \right\} \text{ foo } \text{alist } \left\{ \begin{array}{l} \left\{ \begin{array}{l} \text{:test } \text{test}_{\text{#eq}} \\ \text{:test-not } \text{test} \end{array} \right\} \\ \text{:key function} \end{array} \right\})$

$\left\{ \begin{array}{l} \text{f assoc-if[-not]} \\ \text{f rassoc-if[-not]} \end{array} \right\} \text{ test } \text{alist } [\text{:key function}]$

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

$(\text{f copy-alist } \text{alist})$

▷ Return copy of *alist*.

4.4 Trees

- (*f*tree-equal *foo bar* {*:test* *test*_{#'eq}
:test-not *test*})
- ▷ Return T if trees *foo* and *bar* have same shape and leaves satisfying *test*.
- {*f*subst *new old tree* } {*:test* *function*_{#'eq}
*f*nsubst *new old tree* } {*:test-not* *function*
:key *function* }
- ▷ Make copy of *tree* with each subtree or leaf matching *old* replaced by *new*.
- {*f*subst-if[-not] *new test tree* } [:key *function*]
{*f*nsubst-if[-not] *new test tree* }
- ▷ Make copy of *tree* with each subtree or leaf satisfying *test* replaced by *new*.
- {*f*sublis *association-list tree* } {*:test* *function*_{#'eq}
*f*nsublis *association-list tree* } {*:test-not* *function*
:key *function* }
- ▷ Make copy of *tree* with each subtree or leaf matching a key in *association-list* replaced by that key's value.
- (*f*copy-tree *tree*)
- ▷ Copy of *tree* with same shape and leaves.

4.5 Sets

- {*f*intersection
*f*set-difference
*f*union
*f*set-exclusive-or
*f*nintersection
*f*nset-difference
*f*nunion
*f*nset-exclusive-or}
- a b* *~a b* *~a ~b*
- {*:test* *function*_{#'eq}
:test-not *function*
:key *function* }
- ▷ Return $a \cap b$, $a \setminus b$, $a \cup b$, or $a \triangle b$, respectively, of lists *a* and *b*.

5 Arrays

5.1 Predicates

- (*f*arrayp *foo*)
(*f*vectorp *foo*)
(*f*simple-vector-p *foo*) ▷ T if *foo* is of indicated type.
(*f*bit-vector-p *foo*)
(*f*simple-bit-vector-p *foo*)
- (*f*adjustable-array-p *array*)
(*f*array-has-fill-pointer-p *array*)
▷ T if *array* is adjustable/has a fill pointer, respectively.
- (*f*array-in-bounds-p *array* [*subscripts*])
▷ Return T if *subscripts* are in *array*'s bounds.

5.2 Array Functions

- {*f*make-array *dimension-sizes* [:adjustable *bool*_{NIL}]
*f*adjust-array *array* *dimension-sizes* }
- {*:element-type* *type*_□
:fill-pointer {*num*_{bool}}_{NIL}
:initial-element *obj*
:initial-contents *tree-or-array*
:displaced-to *array*_{NIL} [:displaced-index-offset *i*_□]
- ▷ Return fresh, or readjust, respectively, vector or array.
- (*f*aref *array* [*subscripts*])
▷ Return array element pointed to by *subscripts*. **settable**.
- (*f*row-major-aref *array* *i*)
▷ Return *i*th element of *array* in row-major order. **settable**.

- (*f*array-row-major-index *array* [*subscripts*])
▷ Index in row-major order of the element denoted by *subscripts*.
- (*f*array-dimensions *array*)
▷ List containing the lengths of *array*'s dimensions.
- (*f*array-dimension *array* *i*)
▷ Length of *i*th dimension of *array*.
- (*f*array-total-size *array*) ▷ Number of elements in *array*.
- (*f*array-rank *array*) ▷ Number of dimensions of *array*.
- (*f*array-displacement *array*) ▷ Target array and offset₂.
- (*f*bit *bit-array* [*subscripts*])
(*f*sbit *simple-bit-array* [*subscripts*])
▷ Return element of *bit-array* or of *simple-bit-array*. **settable**.
- (*f*bit-not *bit-array* [*result-bit-array*_{NIL}])
▷ Return result of bitwise negation of *bit-array*. If *result-bit-array* is T, put result in *bit-array*; if it is NIL, make a new array for result.

- {*f*bit-eqv
*f*bit-and
*f*bit-andc1
*f*bit-andc2
*f*bit-nand
*f*bit-ior
*f*bit-orc1
*f*bit-orc2
*f*bit-xor
*f*bit-nor}
- bit-array-a bit-array-b* [*result-bit-array*_{NIL}])
- ▷ Return result of bitwise logical operations (cf. operations of **fboole**, page 4) on *bit-array-a* and *bit-array-b*. If *result-bit-array* is T, put result in *bit-array-a*; if it is NIL, make a new array for result.

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

array-dimension-limit
▷ Upper bound of an array dimension; ≥ 1024 .

array-total-size-limit ▷ Upper bound of array size; ≥ 1024 .

5.3 Vector Functions

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

- (*f*vector *foo**) ▷ Return fresh simple vector of *foos*.
- (*f*svref *vector* *i*) ▷ Element *i* of simple *vector*. **settable**.
- (*f*vector-push *foo* *vector*)
▷ Return NIL if *vector*'s fill pointer equals size of *vector*. Otherwise replace element of *vector* pointed to by fill pointer with *foo*; then increment fill pointer.
- (*f*vector-push-extend *foo* *vector* [*num*])
▷ Replace element of *vector* pointed to by fill pointer with *foo*, then increment fill pointer. Extend *vector*'s size by \geq *num* if necessary.
- (*f*vector-pop *vector*)
▷ Return element of *vector* its fillpointer points to after decrementation.
- (*f*fill-pointer *vector*) ▷ Fill pointer of *vector*. **settable**.

(*f*fboundp {foo})
 ▷ T if *foo* is a global function or macro.

9.2 Variables

{*m*defconstant
*m*defparameter} *foo* *form* [*doc*])
 ▷ Assign value of *form* to global constant/dynamic variable *foo*.

(*m*defvar *foo* [*form* [*doc*]])
 ▷ Unless bound already, assign value of *form* to dynamic variable *foo*.

{*m*setf
*m*psetf} {*place form*}*
 ▷ Set *places* to primary values of *forms*. Return values of last *form*/NIL; work sequentially/in parallel, respectively.

{*s*setq
*m*psetq} {*symbol form*}*
 ▷ Set *symbols* to primary values of *forms*. Return value of last *form*/NIL; work sequentially/in parallel, respectively.

(*f*set *symbol* *foo*)
 ▷ Set *symbol*'s value cell to *foo*. Deprecated.

(*m*multiple-value-setq *vars form*)
 ▷ Set elements of *vars* to the values of *form*. Return *form*'s primary value.

(*m*shiftf *place*⁺ *foo*)
 ▷ Store value of *foo* in rightmost *place* shifting values of *places* left, returning first *place*.

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

(*f*makunbound *foo*) ▷ Delete special variable *foo* if any.

(*f*get *symbol* *key* [*default*_{NIL}])
 (*f*getf *place* *key* [*default*_{NIL}])
 ▷ First entry *key* from property list stored in *symbol*/in *place*, respectively, or *default* if there is no *key*. setfable.

(*f*get-properties *property-list* *keys*)
 ▷ Return key and value of first entry from *property-list* matching a key from *keys*, and tail of *property-list* starting with that key. Return NIL, NIL, and NIL if there was no matching key in *property-list*.

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

(*s*progv *symbols* *values* *form*^P)
 ▷ Evaluate *forms* with locally established dynamic bindings of *symbols* to *values* or NIL. Return values of *forms*.

{*s*let
*s*let*} ({*name*
(*name* [*value*_{NIL}])})* (declare *decl**)* *form*^P)
 ▷ Evaluate *forms* with *names* lexically bound (in parallel or sequentially, respectively) to *values*. Return values of *forms*.

(*m*multiple-value-bind (*var*^{*}) *values-form* (declare *decl**)*
body-form^P)
 ▷ Evaluate *body-forms* with *vars* lexically bound to the return values of *values-form*. Return values of *body-forms*.

{*f*find
*f*position} *foo* *sequence* {*from-end* *bool*_{NIL}
:*test* *function*_{#'eq}
:*test-not* *test*
:*start* *start*₀
:*end* *end*_{NIL}
:*key* *function*}

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

{*f*find-if
*f*find-if-not
*f*position-if
*f*position-if-not} *test* *sequence* {*from-end* *bool*_{NIL}
:*start* *start*₀
:*end* *end*_{NIL}
:*key* *function*}

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

(*f*search *sequence-a* *sequence-b*) {*from-end* *bool*_{NIL}
:*test* *function*_{#'eq}
:*test-not* *function*
:*start1* *start-a*₀
:*start2* *start-b*₀
:*end1* *end-a*_{NIL}
:*end2* *end-b*_{NIL}
:*key* *function*}

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

{*f*remove *foo* *sequence*
*f*delete *foo* *sequence*} {*from-end* *bool*_{NIL}
:*test* *function*_{#'eq}
:*test-not* *function*
:*start* *start*₀
:*end* *end*_{NIL}
:*key* *function*
:*count* *count*_{NIL}}

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

{*f*remove-if
*f*remove-if-not
*f*delete-if
*f*delete-if-not} *test* *sequence* {*from-end* *bool*_{NIL}
:*start* *start*₀
:*end* *end*_{NIL}
:*key* *function*
:*count* *count*_{NIL}}

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

{*f*remove-duplicates *sequence*
*f*delete-duplicates *sequence*} {*from-end* *bool*_{NIL}
:*test* *function*_{#'eq}
:*test-not* *function*
:*start* *start*₀
:*end* *end*_{NIL}
:*key* *function*}

▷ Make copy of *sequence* without duplicates.

{*f*substitute *new* *old* *sequence*
*f*nsubstitute *new* *old* *sequence*} {*from-end* *bool*_{NIL}
:*test* *function*_{#'eq}
:*test-not* *function*
:*start* *start*₀
:*end* *end*_{NIL}
:*key* *function*
:*count* *count*_{NIL}}

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

{*f*substitute-if
*f*substitute-if-not
*f*nsubstitute-if
*f*nsubstitute-if-not} *new* *test* *sequence* {*from-end* *bool*_{NIL}
:*start* *start*₀
:*end* *end*_{NIL}
:*key* *function*
:*count* *count*_{NIL}}

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

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

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

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

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

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

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

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

▷ 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 21.

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

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

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

▷ 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* **clrhsh** *hash-table*) ▷ Empty *hash-table*.

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

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

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

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

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

▷ Test function used in *hash-table*.

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

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

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

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

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

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

8 Structures

(*m* **defstruct**

foo $\left\{ \begin{array}{l} \text{:conc-name} \\ \text{:conc-name } [\text{slot-prefix}_{\boxed{\text{foo-}}}] \\ \text{:constructor} \\ \text{:constructor } [\text{maker}_{\boxed{\text{MAKE-foo}}} [(\text{ord-}\lambda^*)]] \\ \text{:copier} \\ \text{:copier } [\text{copier}_{\boxed{\text{COPY-foo}}}]] \\ \text{:include } \text{struct} \left\{ \begin{array}{l} \text{slot} \\ \text{:type } \text{sl-type} \\ \text{:read-only } \hat{b} \end{array} \right\} \\ \text{:type } \left\{ \begin{array}{l} \text{list} \\ \text{vector} \\ \text{vector } \text{type} \end{array} \right\} \\ \text{:named } \left\{ \begin{array}{l} \text{:named} \\ \text{:initial-offset } \hat{n} \end{array} \right\} \\ \text{:print-object } [\text{o-printer}] \\ \text{:print-function } [\text{f-printer}] \\ \text{:predicate} \\ \text{:predicate } [\text{p-name}_{\boxed{\text{foo-P}}}]] \end{array} \right\}$

doc $\left\{ \begin{array}{l} \text{slot} \\ \text{:type } \text{slot-type} \\ \text{:read-only } \text{bool} \end{array} \right\}$

▷ Define structure *foo* together with functions *MAKE-foo*, *COPY-foo* and *foo-P*; and **setfable** accessors *foo-slot*. Instances are of class *foo* or, if **defstruct** option **:type** is given, of the specified type. They can be created by (*MAKE-foo* *{:slot value}*^{*}) or, if *ord-λ* (see page 17) is given, by (*maker* *arg*^{*} *{:key value}*^{*}). In the latter case, *args* and *:keys* correspond to the positional and keyword parameters defined in *ord-λ* whose *vars* in turn correspond to *slots*. **:print-object**/**:print-function** generate a *g* **print-object** method for an instance *bar* of *foo* calling (*o-printer* *bar* *stream*) or (*f-printer* *bar* *stream* *print-level*), respectively. If **:type** without **:named** is given, no *foo-P* is created.

(*f* **copy-structure** *structure*)

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

9 Control Structure

9.1 Predicates

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

(*f* **eql** *foo* *bar*)

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

(*f* **equal** *foo* *bar*)

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

(*f* **equalp** *foo* *bar*)

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

(*f* **not** *foo*)

▷ *T* if *foo* is NIL; *NIL* otherwise.

(*f* **boundp** *symbol*)

▷ *T* if *symbol* is a special variable.

(*f* **constantp** *foo* [*environment*_{NIL}])

▷ *T* if *foo* is a constant form.

(*f* **functionp** *foo*)

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

(*m*case test ($\widehat{\left\{ \begin{smallmatrix} \text{key}^* \\ \text{key} \end{smallmatrix} \right\}}$) foo^{P^*})* [$\left\{ \begin{smallmatrix} \text{otherwise} \\ \text{T} \end{smallmatrix} \right\}$ 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 .

($\left\{ \begin{smallmatrix} \text{mcase} \\ \text{mccase} \end{smallmatrix} \right\}$ test ($\widehat{\left\{ \begin{smallmatrix} \text{key}^* \\ \text{key} \end{smallmatrix} \right\}}$) 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 $\text{form}^* \text{NIL}$)

▷ Evaluate *forms* from left to right. Immediately return NIL if one *form*'s value is NIL. Return values of last *form* otherwise.

(*m*or $\text{form}^* \text{NIL}$)

▷ 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 $\text{form}^* \text{NIL}$)

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

(*s*multiple-value-prog1 form-r form^*)
(*m*prog1 form-r form^*)
(*m*prog2 $\text{form-a form-r form}^*$)

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

($\left\{ \begin{smallmatrix} \text{mprog} \\ \text{mprog*} \end{smallmatrix} \right\}$ ($\left\{ \begin{smallmatrix} \text{name} \\ \text{(name [value NIL])} \end{smallmatrix} \right\}^*$) (declare $\widehat{\text{decl}^*}$)* ($\widehat{\text{tag}} \left\{ \begin{smallmatrix} \text{form} \end{smallmatrix} \right\}^*$)

▷ 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 ($\widehat{\text{tag}} \left\{ \begin{smallmatrix} \text{form} \end{smallmatrix} \right\}^*$)

▷ 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 $\widehat{\text{tag}}$)

▷ 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 *f*eq *tag* return with the values of *form*.

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

(*m*destructuring-bind *destruct-λ* *bar* (declare $\widehat{\text{decl}^*}$)* form^{P^*})

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

9.3 Functions

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

(*var** [**&optional** (*var* [*init* NIL] [*supplied-p*])])* [**&rest** *var*]

[**&key** (*var* (*key* *var*)) [*init* NIL] [*supplied-p*])]*

[**&allow-other-keys**] [**&aux** (*var* [*init* NIL])])*].

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

($\left\{ \begin{smallmatrix} \text{mdefun} \\ \text{mlambda} \end{smallmatrix} \right\}$ (*foo* (*ord-λ**) (**setf** *foo*) (*new-value ord-λ**)*) (declare $\widehat{\text{decl}^*}$)* [*doc*])

form^{P*}

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

($\left\{ \begin{smallmatrix} \text{slet} \\ \text{slabels} \end{smallmatrix} \right\}$ ((*foo* (*ord-λ**) (**setf** *foo*) (*new-value ord-λ**)*) (declare $\widehat{\text{local-decl}^*}$)* [*doc*] *local-form*^{P*}*) (declare $\widehat{\text{decl}^*}$)* form^{P^*})

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

(*s*function (*foo* (*m*lambda form^*)))

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

(*f*apply (*function* (**setf** *function*)) *arg** *args*)

▷ Values of *function* called with *args* and the list elements of *args*. *setf*able if *function* is one of *f*aref, *f*bit, and *f*sbit.

(*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*. *setf*able.

(*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** $\left\{ \begin{smallmatrix} \text{foo} \\ (\text{setf } \text{foo}) \end{smallmatrix} \right\}$)

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

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

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

c **call-arguments-limit**

c **lambda-parameters-limit**

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

c **multiple-values-limit**

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

9.4 Macros

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

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

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

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

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

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

or

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

$$\left\{ \begin{smallmatrix} \text{var} \\ (\text{var } [\text{init}_{\text{NIL}} [\text{supplied-}p]]) \end{smallmatrix} \right\}^*] [E] . \text{rest-var}).$$

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.

$\left\{ \begin{smallmatrix} \text{mdefmacro} \\ \text{fdefine-compiler-macro} \end{smallmatrix} \right\} \left\{ \begin{smallmatrix} \text{foo} \\ (\text{setf } \text{foo}) \end{smallmatrix} \right\} (\text{macro-}\lambda^*) (\text{declare}$

- $\widehat{\text{decl}}^*)^* [\widehat{\text{doc}}] \text{form}^{\text{P}_k})$
- ▷ 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** $\widehat{\text{local-decl}}^*)^* [\widehat{\text{doc}}$

- $\text{macro-form}^{\text{P}_k})^*) (\text{declare } \widehat{\text{decl}}^*)^* \text{form}^{\text{P}_k})$
- ▷ 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** $\widehat{\text{decl}}^*)^* \text{form}^{\text{P}_k})$

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

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

- $\left\{ \begin{smallmatrix} \text{updater } [\widehat{\text{doc}}] \\ (\text{setf-}\lambda^*) (s\text{-var}^*) (\text{declare } \widehat{\text{decl}}^*)^* [\widehat{\text{doc}}] \text{form}^{\text{P}_k} \end{smallmatrix} \right\}$
- where defsetf lambda list (*setf-λ**) has the form (*var**

$[\&\text{optional} \left\{ \begin{smallmatrix} \text{var} \\ (\text{var } [\text{init}_{\text{NIL}} [\text{supplied-}p]]) \end{smallmatrix} \right\}^*] [\&\text{rest } \text{var}]$

$[\&\text{key} \left\{ \left(\begin{smallmatrix} \text{var} \\ (\text{:key } \text{var}) \end{smallmatrix} \right) [\text{init}_{\text{NIL}} [\text{supplied-}p]] \right\}^*]$

$[\&\text{allow-other-keys}] [\&\text{environment } \text{var}]$)

- ▷ 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** $\widehat{\text{decl}}^*)^* [\widehat{\text{doc}}$

- $\text{form}^{\text{P}_k})$
- ▷ 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**

- $\left\{ \begin{smallmatrix} \text{var} \\ (\text{var } [\text{init}_{\text{NIL}} [\text{supplied-}p]]) \end{smallmatrix} \right\}^*] [\&\text{rest } \text{var}]) \text{function } [\widehat{\text{doc}}])$
- ▷ Define macro *foo* able to modify a place. On invocation of (*foo place arg**), the value of *function* applied to *place* and *args* will be stored into *place* and returned.

c **lambda-list-keywords**

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

&whole *var*

- ▷ Bind *var* to the entire macro call form.

&optional *var**

- ▷ Bind *vars* to corresponding arguments if any.

{&rest|&body} *var*

- ▷ Bind *var* to a list of remaining arguments.

&key *var**

- ▷ Bind *vars* to corresponding keyword arguments.

&allow-other-keys

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

&environment *var*

- ▷ Bind *var* to the lexical compilation environment.

&aux *var**

- ▷ Bind *vars* as in **slet***.

9.5 Control Flow

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

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

(*m* **cond** (*test* then_{test}^{P_k})*)

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

$\left\{ \begin{smallmatrix} \text{mwhen} \\ \text{munless} \end{smallmatrix} \right\} \text{test } \text{foo}^{\text{P}_k})$

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

10 CLOS

10.1 Classes

(*f*slot-exists-p *foo bar*) ▷ T if *foo* has a slot *bar*.

(*f*slot-boundp *instance slot*) ▷ T if *slot* in *instance* is bound.

(*m*defclass *foo* (*superclass** standard-object)

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

▷ Define or modify class *foo* as a subclass of *superclasses*. Transform existing instances, if any, by *g*make-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*.

(*f*find-class *symbol* [*errorp*_{NT}] [*environment*])

▷ Return class named *symbol*. **setfable**.

(*g*make-instance *class* *{:initarg value}* other-keyarg**)

▷ Make new instance of *class*.

(*g*reinitialize-instance *instance* *{:initarg value}* other-keyarg**)

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

(*f*slot-value *foo slot*) ▷ Return value of *slot* in *foo*. **setfable**.

(*f*slot-makunbound *instance slot*)

▷ Make *slot* in *instance* unbound.

$\left\{ \begin{array}{l} \text{mwith-slots } (\{ \widehat{\text{slot}} | (\widehat{\text{var}} \text{ slot})^* \})^* \\ \text{mwith-accessors } ((\widehat{\text{var}} \text{ accessor})^*)^* \end{array} \right\} \text{ instance } (\text{declare } \widehat{\text{decl}})^*$

*form_P**

▷ 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*.

(*g*class-name *class*)

((*setf* *g*class-name) *new-name class*) ▷ Get/set name of *class*.

(*f*class-of *foo*) ▷ Class *foo* is a direct instance of.

(*g*change-class *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.

(*g*make-instances-obsolete *class*)

▷ Update all existing instances of *class* using *g*update-instance-for-redefined-class.

$\left\{ \begin{array}{l} \text{ginitialize-instance } \text{instance} \\ \text{gupdate-instance-for-different-class } \text{previous current} \end{array} \right\}$

{:initarg value} other-keyarg**)

▷ Set slots on behalf of *g*make-instance/of *g*change-class by means of *g*shared-initialize.

9.6 Iteration

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

▷ Evaluate *stagbody*-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 **sblock** named NIL.

(*m*dotimes (*var i* [*result*_{NT}]) (*declare* *decl**)* *{tag|form}**)

▷ Evaluate *stagbody*-like body with *var* successively bound to integers from 0 to *i* - 1. Upon evaluation of *result*, *var* is *i*. Implicitly, the whole form is a **sblock** named NIL.

(*m*dolist (*var list* [*result*_{NT}]) (*declare* *decl**)* *{tag|form}**)

▷ Evaluate *stagbody*-like body with *var* successively bound to the elements of *list*. Upon evaluation of *result*, *var* is NIL. Implicitly, the whole form is a **sblock** named NIL.

9.7 Loop Facility

(*m*loop *form**)

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

(*m*loop *clause**)

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

named *n*_{NT} ▷ Give *m*loop's implicit **sblock** a name.

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

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

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

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

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

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

▷ 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 $\{ \text{step}_{\text{NT}} | \text{function}_{\text{NT}} \}$

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

= *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.

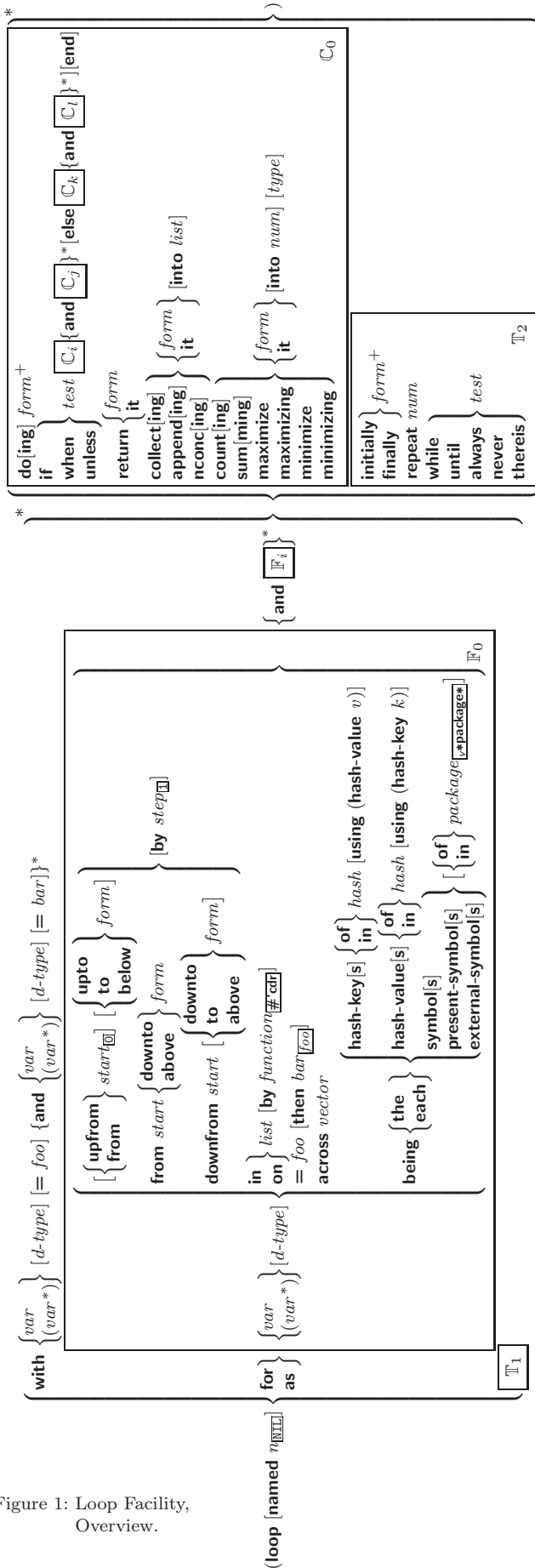


Figure 1: Loop Facility, Overview.

{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 $_{var}package^*$]
 ▷ Bind *var* successively to the accessible symbols, or the present symbols, or the external symbols respectively, of *package*.

{do|doing} $form^+$
 ▷ Evaluate *forms* in every iteration.

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

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

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

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

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

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

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

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

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

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

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

{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.

(*f*make-condition *condition-type* {:*initarg-name value*}*)
 ▷ Return new instance of *condition-type*.

$\left\{ \begin{array}{l} \text{fsignal} \\ \text{fwarn} \\ \text{ferror} \end{array} \right\} \left\{ \begin{array}{l} \text{condition} \\ \text{condition-type } \{:\text{initarg-name value}\}^* \\ \text{control arg}^* \end{array} \right\}$
 ▷ Unless handled, signal as **condition**, **warning** or **error**, respectively, *condition* or a new instance of *condition-type* or, with *f*format control and args (see page 36), **simple-condition**, **simple-warning**, or **simple-error**, respectively. From *f*signal and *f*warn, return NIL.

(*f*cerror *continue-control* $\left\{ \begin{array}{l} \text{condition continue-arg}^* \\ \text{condition-type } \{:\text{initarg-name value}\}^* \\ \text{control arg}^* \end{array} \right\}$)
 ▷ Unless handled, signal as correctable **error** *condition* or a new instance of *condition-type* or, with *f*format control and args (see page 36), **simple-error**. In the debugger, use *f*format arguments *continue-control* and *continue-args* to tag the continue option. Return NIL.

(*m*ignore-errors *form*^P)
 ▷ Return values of *forms* or, in case of **errors**, NIL and the condition.

(*f*invoke-debugger *condition*)
 ▷ Invoke debugger with *condition*.

(*m*assert *test* [(*place**)
 $\left\{ \begin{array}{l} \text{condition continue-arg}^* \\ \text{condition-type } \{:\text{initarg-name value}\}^* \\ \text{control arg}^* \end{array} \right\}]]$)
 ▷ If *test*, which may depend on *places*, returns NIL, signal as correctable **error** *condition* or a new instance of *condition-type* or, with *f*format control and args (see page 36), **error**. When using the debugger's continue option, *places* can be altered before re-evaluation of *test*. Return NIL.

(*m*handler-case *foo* $(\text{type } ([\text{var}]) (\text{declare } \widehat{\text{decl}}^*)^* \text{condition-form}^{\text{P}})^*$
 $[(\text{no-error } (\text{ord-}\lambda^*) (\text{declare } \widehat{\text{decl}}^*)^* \text{form}^{\text{P}})]])$
 ▷ If, on evaluation of *foo*, a condition of *type* is signalled, evaluate matching *condition-forms* with *var* bound to the condition, and return their values. Without a condition, bind *ord-λs* to values of *foo* and return values of *forms* or, without a **no-error** clause, return values of *foo*. See page 17 for (*ord-λ**).

(*m*handler-bind ((*condition-type* *handler-function*)* *form*^P)
 ▷ Return values of *forms* after evaluating them with *condition-types* dynamically bound to their respective *handler-functions* of argument condition.

(*m*with-simple-restart ($\left\{ \begin{array}{l} \text{restart} \\ \text{NIL} \end{array} \right\}$ *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 36) and return NIL and T.

(*m*restart-case *form* (*restart* (*ord-λ**) $\left\{ \begin{array}{l} \text{:interactive arg-function} \\ \text{:report } \left\{ \begin{array}{l} \text{report-function} \\ \text{string } [\text{"restart"}] \end{array} \right\} \\ \text{:test test-function} \square \end{array} \right\}$)
 $(\text{declare } \widehat{\text{decl}}^*)^* \text{restart-form}^{\text{P}})^*$)
 ▷ Return values of *form* or, if during evaluation of *form* one of the dynamically established *restarts* is called, the values of its *restart-forms*. A *restart* is visible under *condition* if (*funcall* #'*test-function* *condition*) returns T. If presented in the debugger, *restarts* are described by *string* or by #'*report-function* (of a stream). A *restart* can be called by (*invoke-restart* *restart arg**), where *args* match *ord-λ**, or by (*invoke-restart-interactively* *restart*) where a list of the respective *args* is supplied by #'*arg-function*. See page 17 for *ord-λ**.

(*g*update-instance-for-redefined-class *new-instance* *added-slots* *discarded-slots* *discarded-slots-property-list* {:*initarg value*}* *other-keyarg**)
 ▷ On behalf of *g*make-instances-obsolete and by means of *g*shared-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.

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

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

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

(*g*slot-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

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

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

(*f*ensure-generic-function $\left\{ \begin{array}{l} \text{foo} \\ \text{(setf foo)} \end{array} \right\}$
 $\left\{ \begin{array}{l} \text{:argument-precedence-order } \text{required-var}^+ \\ \text{:declare } (\text{optimize } \text{method-selection-optimization}) \\ \text{:documentation } \text{string} \\ \text{:generic-function-class } \text{gf-class} \\ \text{:method-class } \text{method-class} \\ \text{:method-combination } \text{c-type } \text{c-arg}^* \\ \text{:lambda-list } \text{lambda-list} \\ \text{:environment } \text{environment} \end{array} \right\}$)
 ▷ Define or modify generic function *foo*. *gf-class* and *lambda-list* must be compatible with a pre-existing generic function or with existing methods, respectively. Changes to *method-class* do not propagate to existing methods. For *c-type* see section 10.3.

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

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

▷ 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

10.3 Method Combination Types

standard

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

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

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

$$(\text{mdefine-method-combination } \text{c-type} \\ \left\{ \begin{array}{l} \text{:documentation } \text{string} \\ \text{:identity-with-one-argument } \text{bool}[\text{NIL}] \\ \text{:operator } \text{operator}[\text{c-type}] \end{array} \right\})$$

▷ **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*, **fcall-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 **mdefgeneric**). Using *c-type* as the *qualifier* in **mdefmethod** makes the *method* primary.

$$(\text{mdefine-method-combination } \text{c-type} (\text{ord-}\lambda^*) ((\text{group}$$

$$\left\{ \begin{array}{l} * \\ \text{qualifier}^* [\text{ }] \\ \text{predicate} \\ \text{:description } \text{control} \\ \text{:order } \left\{ \begin{array}{l} \text{:most-specific-first} \\ \text{:most-specific-last} \end{array} \right\} [\text{most-specific-first}]^* \\ \text{:required } \text{bool} \\ \left\{ \begin{array}{l} (\text{:arguments } \text{method-combination-}\lambda^*) \\ (\text{:generic-function } \text{symbol}) \\ (\text{declare } \widehat{\text{decl}}^*)^* \\ \text{doc} \end{array} \right\} \text{body}^{\text{P}_k}) \end{array} \right\})$$

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

$$(\text{mcall-method}$$

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

▷ 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 31.

$$(\text{mdefine-condition } \text{foo} (\text{parent-type}^* \text{condition})$$

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

▷ 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.

13 Input/Output

13.1 Predicates

(*f*stream-p *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 36, *f*format, for *control* and *args*.
 (*m*with-standard-io-syntax *form*^P)
 ▷ Evaluate *forms* with standard behaviour of reader and printer. Return *values of forms*.
 {*f*read
*f*read-preserving-whitespace} [*stream*_U *standard-input** [*eof-error*_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 *position* of next character.
 (*f*read-delimited-list *char* [*stream*_U *standard-input** [*recursive*_{NIL}]])
 ▷ Continue reading until encountering *char*. Return *list* of objects read. Signal error if no *char* is found in stream.
 (*f*read-char [*stream*_U *standard-input** [*eof-error*_T [*eof-val*_{NIL} [*recursive*_{NIL}]]]])
 ▷ Return *next character* from *stream*.
 (*f*read-char-no-hang [*stream*_U *standard-input** [*eof-error*_T [*eof-val*_{NIL} [*recursive*_{NIL}]]]])
 ▷ *Next character* from *stream* or *NIL* if none is available.
 (*f*peek-char [*mode*_{NIL} [*stream*_U *standard-input** [*eof-error*_T [*eof-val*_{NIL} [*recursive*_{NIL}]]]])
 ▷ Next, or if *mode* is *T*, next non-whitespace *character*, or if *mode* is a character, *next instance* of it, from *stream* without removing it there.
 (*f*unread-char *character* [*stream*_U *standard-input**])
 ▷ Put last *f*read-chared *character* back into *stream*; return *NIL*.
 (*f*read-byte *stream* [*eof-error*_T [*eof-val*_{NIL}]])
 ▷ Read *next byte* from binary *stream*.
 (*f*read-line [*stream*_U *standard-input** [*eof-error*_T [*eof-val*_{NIL} [*recursive*_{NIL}]]]])
 ▷ Return a *line of text* from *stream* and *T* if line has been ended by end of file.

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

(*f*invoke-restart *restart* *arg**)
 (*f*invoke-restart-interactively *restart*)
 ▷ Call function associated with *restart* with arguments given or prompted for, respectively. If *restart* function returns, return *its values*.

{*f*find-restart
*f*compute-restarts *name*} [*condition*])
 ▷ Return innermost restart *name*, or a list of all restarts, respectively, out of those either associated with *condition* or un-associated at all; or, without *condition*, out of all restarts. Return *NIL* if search is unsuccessful.

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

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

(*m*with-condition-restarts *condition* *restarts* *form*^P)
 ▷ Evaluate *forms* with *restarts* dynamically associated with *condition*. Return *values of forms*.

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

(*f*cell-error-name *condition*)
 ▷ *Name of cell* which caused *condition*.

(*f*unbound-slot-instance *condition*)
 ▷ *Instance* with unbound slot which caused *condition*.

(*f*print-not-readable-object *condition*)
 ▷ The *object* not readably printable under *condition*.

(*f*package-error-package *condition*)
 (*f*file-error-pathname *condition*)
 (*f*stream-error-stream *condition*)
 ▷ *Package*, *path*, or *stream*, respectively, which caused the *condition* of indicated type.

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

(*f*simple-condition-format-control *condition*)
 (*f*simple-condition-format-arguments *condition*)
 ▷ Return *f*format *control* or *list of f*format *arguments*, respectively, of *condition*.

*break-on-signals*_{NIL}
 ▷ Condition type debugger is to be invoked on.

$$_v \text{ *debugger-hook* } \boxed{\text{NIL}}$$

- ▷ 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.

$$(\text{typep } foo \text{ type } [environment_{\text{NIL}}]) \quad \triangleright \quad \text{T if } foo \text{ is of type.}$$

(**fsubtypep** *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.

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

(**fcoerce** *object type*) ▷ Coerce *object* into *type*.

$$(\text{typecase } \widehat{foo} \ (a\text{-form}^{\text{P}_*})^* \ [(\left\{ \begin{array}{l} \text{otherwise} \\ \text{true} \end{array} \right\} \ b\text{-form}^{\text{P}_*})])$$

▷ Return values of the first $a\text{-form}^*$ whose *type* is *foo* of.
Return values of $b\text{-forms}$ if no *type* matches.

$\left\{ \begin{matrix} m \\ m \end{matrix} \text{etypecase} \right\} \text{foo } (\widehat{\text{type form}}^{\text{P}_*})^*$
 ▷ Return values of the first form^* whose *type* is *foo*
 of. Signal non-correctable/correctable **type-error** if no *type*
 matches.

$(_f \mathbf{type-of} \, foo)$ \triangleright Type of foo .

(*m* **check-type** *place* *type* [*string* $\boxed{\{a \text{ an} \} \text{ type}}$])

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

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

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

($\text{fupgraded-array-element-type } type [environment_{\text{NTT}}]$)
 \triangleright Element type of most specialized array capable of holding
 elements of *type*.

(*macro-λ**) (**declare** \widehat{decl}^*)* $\lceil \widehat{doc} \rceil form^P$)

- ▷ 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 18 but with default value of * instead of NIL. *forms* are enclosed in an implicit **block** named *foo*.

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

(**satisfies** *predicate*)
 ▷ Type specifier for all objects satisfying *predicate*.

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

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

(**and** $type^*_{\mathbb{T}}$) ▷ Type specifier for intersection of *types*.

(or $type^* \boxed{NTU}$) ▷ Type specifier for union of *types*.

(values *type** [**&optional** *type** [**&rest** *other-args*]])
 ▷ Type specifier for multiple values.

- * \triangleright As a type argument (cf. Figure 2): no restriction.

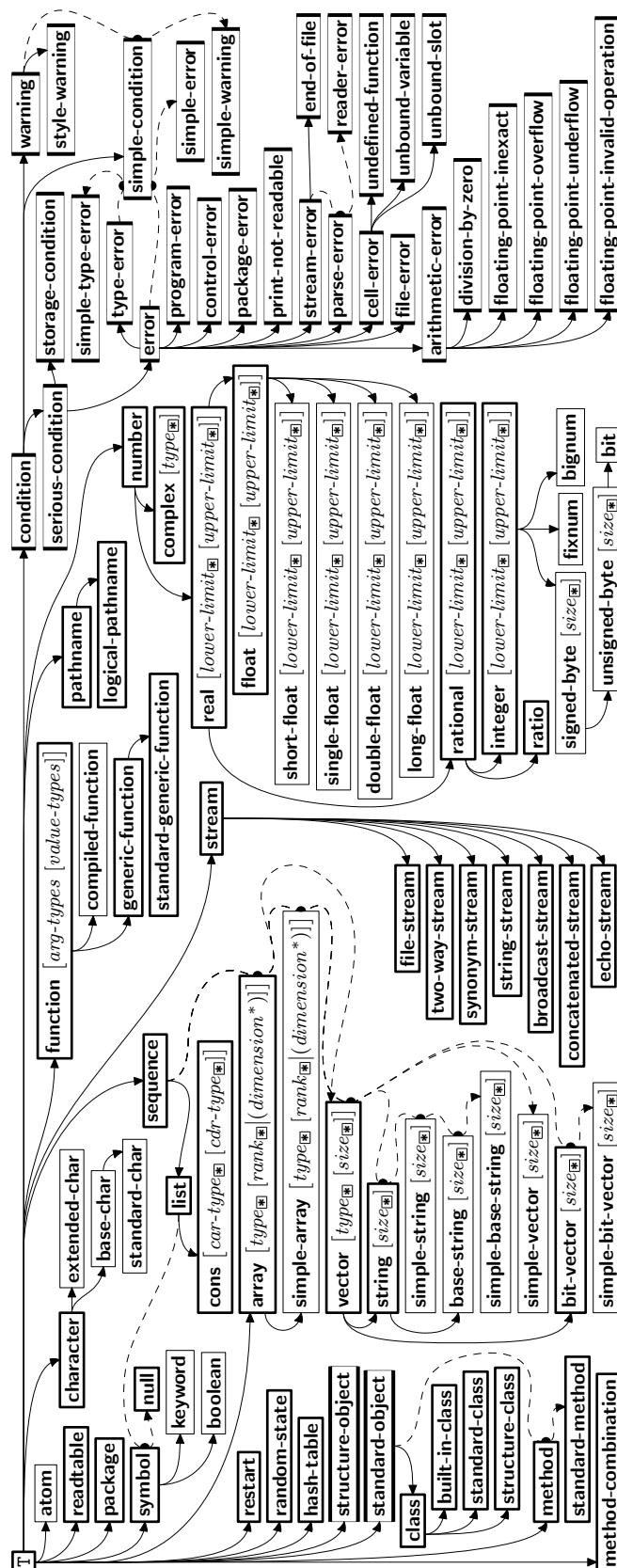


Figure 2: Precedence Order of System Classes (\square), Classes (\blacksquare), Types (\sqsubset), and Condition Types (\blacksquare). Every type is also a supertype of NIL, the empty type.

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

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

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

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

*v**print-case**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**1*

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

*v**print-gensym**1*

▷ If T, print #: before uninterned symbols.

*v**print-length**NIL*

*v**print-level**NIL*

*v**print-lines**NIL*

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

*v**print-miser-width*

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

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

*v**print-radix**NIL*

▷ If T, print rationals with a radix indicator.

*v**print-readably**NIL*

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

*v**print-right-margin**NIL*

▷ Right margin width in ems while pretty-printing.

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

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

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

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

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

▷ Return copy of *table* or, if *table* is *NIL*, initial value of *v**print-pprint-dispatch*.

*v**print-pprint-dispatch*

▷ Current pretty print dispatch table.

13.5 Format

(*m* formatter *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*.

(*f* read-sequence *sequence* *stream* [:start *start*0] [:end *end**NIL*])

▷ Replace elements of *sequence* between *start* and *end* with elements from binary or character *stream*. Return *index* of *sequence*'s first unmodified element.

(*f* readtable-case *readtable*)*upcase*

▷ Case sensitivity attribute (one of :upcase, :downcase, :preserve, :invert) of *readtable*. *settable*.

(*f* copy-readtable [*from-readtable**v**readtable*] [*to-readtable**NIL*])

▷ Return copy of *from-readtable*.

(*f* set-syntax-from-char *to-char* *from-char* [*to-readtable**v**readtable*] [*from-readtable**standard-readtable*])

▷ Copy syntax of *from-char* to *to-readtable*. Return T.

*v**readtable*

▷ Current readtable.

*v**read-base**10* ▷ Radix for reading integers and ratios.

*v**read-default-float-format**single-float*

▷ Floating point format to use when not indicated in the number read.

*v**read-suppress**NIL*

▷ If T, reader is syntactically more tolerant.

(*f* set-macro-character *char* *function* [*non-term-p**NIL*] [*rt**v**readtables*])

▷ Make *char* a macro character associated with *function* of stream and *char*. Return T.

(*f* get-macro-character *char* [*rt**v**readtable*])

▷ Reader macro function associated with *char*, and T if *char* is a non-terminating macro character.

(*f* make-dispatch-macro-character *char* [*non-term-p**NIL*] [*rt**v**readtables*])

▷ Make *char* a dispatching macro character. Return T.

(*f* set-dispatch-macro-character *char* *sub-char* *function* [*rt**v**readtables*])

▷ Make *function* of stream, *n*, *sub-char* a dispatch function of *char* followed by *n*, followed by *sub-char*. Return T.

(*f* get-dispatch-macro-character *char* *sub-char* [*rt**v**readtables*])

▷ 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*

; *continuation* ▷ Regarding line on which it appears.

(*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 ▷ (*f*character "c"), the character c.

#B*n*; #O*n*; *n*.; #X*n*; #r*Rn*

▷ Integer of radix 2, 8, 10, 16, or *r*; $2 \leq r \leq 36$.

- n/d \triangleright The **ratio** $\frac{n}{d}$.
- $\{[m].n[\{S|F|D|L|E\}x_{\text{EQ}}]|m[.n]\{S|F|D|L|E\}x\}$
 $\triangleright m.n \cdot 10^x$ as **short-float**, **single-float**, **double-float**, **long-float**, or the type from ***read-default-float-format***.
- #C(a b)** \triangleright (**fcomplex** $a\ b$), the complex number $a + bi$.
- #'foo** \triangleright (**function** foo); the function named foo .
- #nAsequence** \triangleright n -dimensional array.
- #[n](foo*)**
 \triangleright Vector of some (or n) $foos$ filled with last foo if necessary.
- #[n]*b***
 \triangleright Bit vector of some (or n) bs filled with last b if necessary.
- #S(type {slot value}*)** \triangleright Structure of $type$.
- #Pstring** \triangleright A pathname.
- #:foo** \triangleright Uninterned symbol foo .
- #.form** \triangleright Read-time value of $form$.
- √read-eval*_{EQ}** \triangleright If **NIL**, a **reader-error** is signalled at **#.**
- #integer= foo** \triangleright Give foo the label $integer$.
- #integer#** \triangleright Object labelled $integer$.
- #<** \triangleright Have the reader signal **reader-error**.
- #+feature when-feature**
#-feature unless-feature
 \triangleright Means $when-feature$ if $feature$ is **T**; means $unless-feature$ if $feature$ is **NIL**. $feature$ is a symbol from **√features***, or (**and**|**or**| $feature^*$), or (**not** $feature$).
- √features***
 \triangleright List of symbols denoting implementation-dependent features.
- |c*|; \c**
 \triangleright Treat arbitrary character(s) c as alphabetic preserving case.

13.4 Printer

- $\left\{ \begin{array}{l} \text{fprin1} \\ \text{fprint} \\ \text{fpprint} \\ \text{fprinc} \end{array} \right\} foo [stream_{\text{v}*standard-output*}]$
 \triangleright 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)
 \triangleright Print foo to string **freadably** or human-readably, respectively.
- (gprint-object object stream)**
 \triangleright Print object to $stream$. Called by the Lisp printer.
- (mprint-unreadable-object (foo stream { :type bool_{NIL} :identity bool_{NIL} }) form_{P*})**
 \triangleright Enclosed in **#<** and **>**, print foo by means of $forms$ to $stream$. Return NIL.
- (fterpri [stream_v*standard-output*])**
 \triangleright Output a newline to $stream$. Return NIL.
- (f fresh-line) [stream_v*standard-output*]**
 \triangleright Output a newline to $stream$ and return T unless $stream$ is already at the start of a line.

- (f write-char char [stream_v*standard-output*])**
 \triangleright Output char to $stream$.
- $\left\{ \begin{array}{l} \text{fwrite-string} \\ \text{fwrite-line} \end{array} \right\} string [stream_{\text{v}*standard-output*} \left\{ \begin{array}{l} \text{:start } start_{\text{EQ}} \\ \text{:end } end_{\text{NIL}} \end{array} \right\}]$
 \triangleright Write string to $stream$ without/with a trailing newline.
- (f write-byte byte stream)** \triangleright Write byte to binary $stream$.
- (f write-sequence sequence stream { :start start_{EQ} :end end_{NIL} })**
 \triangleright Write elements of sequence to binary or character $stream$.
- $\left\{ \begin{array}{l} \text{fwrite} \\ \text{fwrite-to-string} \end{array} \right\} foo \left\{ \begin{array}{l} \text{:array } bool \\ \text{:base } radix \\ \text{:case } \left\{ \begin{array}{l} \text{:upcase} \\ \text{:downcase} \\ \text{:capitalize} \end{array} \right\} \\ \text{:circle } bool \\ \text{:escape } bool \\ \text{:gensym } bool \\ \text{:length } \{int|NIL\} \\ \text{:level } \{int|NIL\} \\ \text{:lines } \{int|NIL\} \\ \text{:miser-width } \{int|NIL\} \\ \text{:pprint-dispatch } dispatch-table \\ \text{:pretty } bool \\ \text{:radix } bool \\ \text{:readably } bool \\ \text{:right-margin } \{int|NIL\} \\ \text{:stream } stream_{\text{v}*standard-output*} \end{array} \right\}$
 \triangleright 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.)
- (fpprint-fill stream foo [parenthesis_{EQ} [noop]])**
(fpprint-tabular stream foo [parenthesis_{EQ} [noop] [n_{EQ}]])
(fpprint-linear stream foo [parenthesis_{EQ} [noop]])
 \triangleright 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 **fformat** directive **~//**.
- (mpprint-logical-block (stream list { :prefix string :per-line-prefix string :suffix string_{EQ} })**
(declare decl*) form_{P*})
 \triangleright 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)**
 \triangleright Take next element off $list$. If there is no remaining tail of $list$, or **√print-length*** or **√print-circle*** indicate printing should end, send element together with an appropriate indicator to $stream$.
- (fpprint-tab { :line :line-relative :section :section-relative }) c i**
 $[stream_{\text{v}*standard-output*}]$
 \triangleright Move cursor forward to column number $c + ki$, $k \geq 0$ being as small as possible.
- (fpprint-indent { :block :current } n [stream_v*standard-output*])**
 \triangleright Specify indentation for innermost logical block relative to leftmost position/to current position. Return NIL.
- (mpprint-exit-if-list-exhausted)**
 \triangleright If $list$ is empty, terminate logical block. Return NIL otherwise.

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

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

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

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

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

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

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

*v****standard-input***
*v****standard-output***
*v****error-output***
 ▷ Standard input stream, standard output stream, or standard error output stream, respectively.

*v****debug-io***
*v****query-io***
 ▷ Bidirectional streams for debugging and user interaction.

13.7 Pathnames and Files

(*f* **make-pathname** {*:host* {*host* *NIL* *:unspecific*}
:device {*device* *NIL* *:unspecific*}
:directory { {*directory* *:wild* *NIL* *:unspecific*}
 {*:absolute* {*directory* *:wild* *NIL* *:unspecific*}*
 {*:relative* {*:wild*
:wild-inferiors
:up
:back}} }
:name {*file-name* *:wild* *NIL* *:unspecific*}
:type {*file-type* *:wild* *NIL* *:unspecific*}
:version {*:newest* *version* *:wild* *NIL* *:unspecific*}
:defaults *path*_{host from *v**default-pathname-defaults*}
:case {*:local* *:common*} *:local* })
 ▷ Construct a *logical* *pathname* if there is a logical *pathname* translation for *host*, otherwise construct a *physical* *pathname*. For *:case* *:local*, leave case of components unchanged. For *:case* *:common*, leave mixed-case components unchanged; convert all-uppercase components into local customary case; do the opposite with all-lowercase components.

{*f* *pathname-host*
f *pathname-device*
f *pathname-directory*
f *pathname-name*
f *pathname-type*
f *pathname-version* *path-or-stream* } *path-or-stream* {*:case* {*:local*
:common} *:local* })
 ▷ Return *pathname* component.

~ [*min-col*] [*:col-inc*] [*:min-pad*] [*:pad-char*])
 [:] [C] {A|S}
 ▷ **Aesthetic/Standard**. Print argument of any type for consumption by humans/by the reader, respectively. With *:*, print *NIL* as *()* rather than *nil*; with *C*, add *pad-chars* on the left rather than on the right.

~ [*radix*] [*:width*] [*:pad-char*] [*:comma-char*] [*:comma-interval*] [:] [C] R
 ▷ **Radix**. (With one or more prefix arguments.) Print argument as number; with *:*, group digits *comma-interval* each; with *C*, always prepend a sign.

{~R|~R|~OR|~OR}
 ▷ **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] \$
 ▷ **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|~CC|~CC}
 ▷ **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 *eq* 1 print nothing, otherwise print *s*; do the same for the previous argument; if argument *eq* 1 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 *@*, left justify. If this would leave less than *spare* characters on the current line, output *nl-text* first.

~ [:] [*@*] < { [*prefix* ~:] [*per-line-prefix* ~*@*;] } *body* [~; *suffix* ~:] ~: [*@*] >

▷ **Logical Block.** Act like **pprint-logical-block** using *body* as *format* control string on the elements of the list argument or, with *@*, on the remaining arguments, which are extracted by **pprint-pop**. With *:*, *prefix* and *suffix* default to (and). When closed by ~*@*>, spaces in *body* are replaced with conditional newlines.

{ ~ [*n*] | ~ [*n*] :i }

▷ **Indent.** Set indentation to *n* relative to leftmost/to current position.

~ [*c*] [*i*] [:] [*@*] T

▷ **Tabulate.** Move cursor forward to column number *c+ki*, *k* ≥ 0 being as small as possible. With *:*, calculate column numbers relative to the immediately enclosing section. With *@*, move to column number *c₀ + c + ki* where *c₀* is the current position.

{ ~ [*m*] * | ~ [*m*] :* | ~ [*n*] @* }

▷ **Go-To.** Jump *m* arguments forward, or backward, or to argument *n*.

~ [*limit*] [:] [*@*] { *text* ~ }

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

~ [*x* [*y* [*z*]]] ^

▷ **Escape Upward.** Leave immediately ~< ~>, ~< ~:~>, ~{ ~ }, ~?, or the entire *format* operation. With one to three prefixes, act only if *x* = 0, *x* = *y*, or *x* ≤ *y* ≤ *z*, respectively.

~ [*i*] [:] [*@*] [[{ *text* ~; } * *text*] [~; : default] ~]

▷ **Conditional Expression.** Use the zero-indexed argument (or *i*th if given) *text* as a *format* control subclause. With *:*, use the first *text* if the argument value is NIL, or the second *text* if it is T. With *@*, do nothing for an argument value of NIL. Use the only *text* and leave the argument to be read again if it is T.

{ ~? | ~@? }

▷ **Recursive Processing.** Process two arguments as control string and argument list, or take one argument as control string and use then the rest of the original arguments.

~ [*prefix* { , *prefix* } *] [:] [*@*] / [*package* [:] : *cl-user*] *function* /

▷ **Call Function.** Call all-uppercase *package::function* with the arguments *stream*, *format-argument*, *colon-p*, *at-sign-p* and *prefixes* for printing *format-argument*.

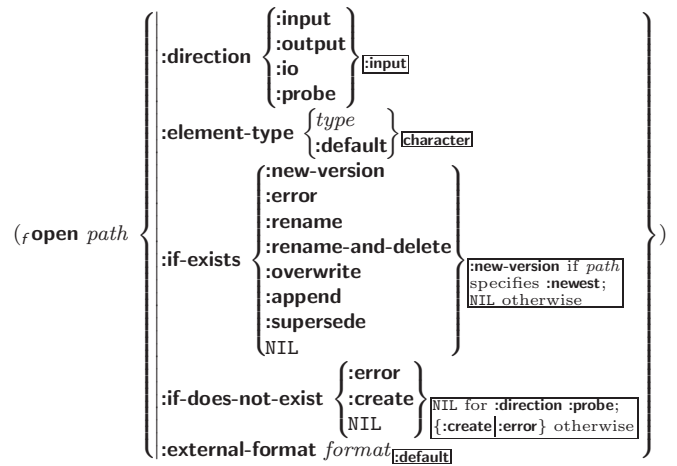
~ [:] [*@*] W

▷ **Write.** Print argument of any type obeying every printer control variable. With *:*, pretty-print. With *@*, print without limits on length or depth.

{ *V* | # }

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

13.6 Streams



(*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*] [*end*])

▷ 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*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**.

$\left\{ \begin{array}{l} \text{documentation} \\ (\text{setf } \text{documentation}) \text{ new-doc} \end{array} \right\} \text{foo} \left\{ \begin{array}{l} \text{'variable'|'function'} \\ \text{'compiler-macro'} \\ \text{'method-combination'} \\ \text{'structure'|'type'|'setf'} \end{array} \right\}$
 ▷ Get/set documentation string of *foo* of given type.

cl

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

nil|*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

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

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

15.2 Compilation

(*f*compile $\left\{ \begin{array}{l} \text{NIL definition} \\ \text{name} \\ (\text{setf name}) \end{array} \right\} [\text{definition}]$)
 ▷ 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**.

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

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

(*f*load *path* $\left\{ \begin{array}{l} \text{:verbose bool} \text{v*load-verbose*} \\ \text{:print bool} \text{v*load-print*} \\ \text{:if-does-not-exist bool} \text{v*} \\ \text{:external-format file-format} \text{v*default*} \end{array} \right\}$)
 ▷ Load source file or compiled file into Lisp environment. Return T if successful.

(*f*parse-namestring *foo* [*host* $\left[\begin{array}{l} \text{default-pathname} \text{v*default-pathname-defaults*} \\ \left\{ \begin{array}{l} \text{:start start} \text{v*} \\ \text{:end end} \text{v*} \\ \text{:junk-allowed bool} \text{v*} \end{array} \right\} \end{array} \right]]$)
 ▷ Return pathname converted from string, pathname, or stream *foo*; and position where parsing stopped.

(*f*merge-pathnames *path-or-stream* $\left[\begin{array}{l} \text{default-path-or-stream} \text{v*default-pathname-defaults*} \\ \text{default-version} \text{v*newest*} \end{array} \right]$)
 ▷ Return pathname made by filling in components missing in *path-or-stream* from *default-path-or-stream*.

v*default-pathname-defaults*

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

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

(*f*enough-namestring *path-or-stream* $\left[\begin{array}{l} \text{root-path} \text{v*default-pathname-defaults*} \end{array} \right]$)
 ▷ Return minimal path string that sufficiently describes the path of *path-or-stream* relative to *root-path*.

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

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

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

(*f*logical-pathname *logical-path-or-stream*)
 ▷ Logical pathname of *logical-path-or-stream*. Logical pathnames are represented as all-uppercase
 "[host:][;]{ $\left\{ \begin{array}{l} \text{dir} \text{v*} \end{array} \right\}^+ \}$ };{ $\left\{ \begin{array}{l} \text{name} \text{v*} \end{array} \right\}^+ \}$ }[. $\left\{ \begin{array}{l} \text{type} \text{v*} \end{array} \right\}^+ \}$]{LISP}
 [. { $\left\{ \begin{array}{l} \text{version} \text{v*} \end{array} \right\}^+ \}$ newestNEWEST}]]".

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

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

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

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

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

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

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

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

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

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

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

14 Packages and Symbols

The Loop Facility provides additional means of symbol handling; see **loop**, page 21.

14.1 Predicates

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

14.2 Packages

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

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

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

(*m*defpackage *foo*
 {
 (:nicknames *nick**)*
 (:documentation *string*)
 (:intern *interned-symbol*)*
 (:use *used-package*)*
 (:import-from *pkg* *imported-symbol*)*
 (:shadowing-import-from *pkg* *shd-symbol*)*
 (:shadow *shd-symbol*)*
 (:export *exported-symbol*)*
 (:size *int*)
 })
 ▷ Create or modify *package foo* with *interned-symbols*, symbols from *used-packages*, *imported-symbols*, and *shd-symbols*. Add *shd-symbols* to *foo*'s shadowing list.

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

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

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

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

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

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

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

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

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

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

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

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

{*f*intern
*f*find-symbol} *foo* [*package**v**package*])
 ▷ Intern or find, respectively, symbol *foo* in *package*. Second return value is one of *internal*, *external*, or *inherited* (or *NIL* if *f*intern has created a fresh symbol).

(*f*unintern *symbol* [*package**v**package*])
 ▷ Remove *symbol* from *package*, return *T* on success.

{*f*import
*f*shadowing-import} *symbols* [*package**v**package*])
 ▷ Make *symbols* internal to *package*. Return *T*. In case of a name conflict signal correctable **package-error** or shadow the old symbol, respectively.

(*f*shadow *symbols* [*package**v**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*.

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

(*m*with-package-iterator (*foo* *packages* [:internal|:external|:inherited]) (declare *decl*)* *form**)
 ▷ 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**nil*])
 ▷ Return fresh, uninterned symbol *#:sn* with *n* from *v**gensym-counter*. Increment *v**gensym-counter*.

(*f*gentemp [*prefix**nil*] [*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.

Index

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

- ν **compile-file** $\left\{ \begin{array}{l} \text{path-name}^* \underline{\text{NIL}} \\ \text{true-name}^* \underline{\text{NIL}} \end{array} \right\}$
 - ▷ Input file used by f **compile-file**/by f **load**.
- ν **compile** $\left\{ \begin{array}{l} \text{print}^* \\ \text{verbose}^* \end{array} \right\}$
 - ▷ Defaults used by f **compile-file**/by f **load**.
- $(s\text{eval-when } (\left\{ \begin{array}{l} \{ \text{compile-toplevel} \mid \text{compile} \} \\ \{ \text{load-toplevel} \mid \text{load} \} \\ \{ \text{execute} \mid \text{eval} \} \end{array} \right\}) \text{ form}^P)$
 - ▷ Return values of forms if $s\text{eval-when}$ 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.)
- $(s\text{locally } (\widehat{\text{declare } decl^*})^* \text{ form}^P)$
 - ▷ Evaluate *forms* in a lexical environment with declarations *decl* in effect. Return values of forms.
- $(m\text{with-compilation-unit } ([\text{override } \text{bool} \underline{\text{NIL}}]) \text{ form}^P)$
 - ▷ Return values of *forms*. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of *forms*.
- $(s\text{load-time-value } \text{form } [\widehat{\text{read-only}} \underline{\text{NIL}}])$
 - ▷ Evaluate *form* at compile time and treat its value as literal at run time.
- $(s\text{quote } \widehat{foo})$ ▷ Return unevaluated *foo*.
- $(g\text{make-load-form } foo [\text{environment}])$
 - ▷ 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.
- $(f\text{make-load-form-saving-slots } foo$
 $\left\{ \begin{array}{l} \text{:slot-names } slots \underline{\text{all local slots}} \\ \text{:environment } environment \end{array} \right\})$
 - ▷ 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*.
- $(f\text{macro-function } symbol [\text{environment}])$
- $(f\text{compiler-macro-function } \left\{ \begin{array}{l} name \\ \text{self } name \end{array} \right\} [\text{environment}])$
 - ▷ Return specified macro function, or compiler macro function, respectively, if any. Return NIL otherwise. **setfable**.
- $(f\text{eval } arg)$
 - ▷ Return values of value of *arg* evaluated in global environment.

15.3 REPL and Debugging

- $\nu+$ $\nu++$ $\nu+++$
- ν^* ν^{**} ν^{***}
- $\nu/$ $\nu///$ $\nu////$
 - ▷ Last, penultimate, or antepenultimate form evaluated in the REPL, or their respective primary value, or a list of their respective values.
- $\nu-$ ▷ Form currently being evaluated by the REPL.
- $(f\text{apropos } string [\text{package} \underline{\text{NIL}}])$
 - ▷ Print interned symbols containing *string*.
- $(f\text{apropos-list } string [\text{package} \underline{\text{NIL}}])$
 - ▷ List of interned symbols containing *string*.
- $(f\text{dribble } [path])$
 - ▷ Save a record of interactive session to file at *path*. Without *path*, close that file.
- $(f\text{ed } [file\text{-or-function} \underline{\text{NIL}}])$ ▷ Invoke editor if possible.

$\left\{ \begin{array}{l} \text{macroexpand-1} \\ \text{macroexpand} \end{array} \right\} \text{form } [\text{environment} \underline{\text{NIL}}])$
 ▷ Return macro expansion, once or entirely, respectively, of *form* and T if *form* was a macro form. Return form and NIL otherwise.

macroexpand-hook

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

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

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

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

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

trace-output

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

15.4 Declarations

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

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

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

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

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

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

▷ Make *foos* names of declarations.

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

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

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

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

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

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

▷ Suppress warnings about used/unused bindings.

$(\text{inline } \text{function}^*)$

$(\text{notinline } \text{function}^*)$

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

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

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

$(\text{special } \text{var}^*)$ ▷ Declare *vars* to be dynamic.

16 External Environment

$(\text{fget-internal-real-time})$

$(\text{fget-internal-run-time})$

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

internal-time-units-per-second

▷ Number of clock ticks per second.

$(\text{fencode-universal-time } \text{sec } \text{min } \text{hour } \text{date } \text{month } \text{year } [\text{zone} \underline{\text{current}}])$

$(\text{fget-universal-time})$

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

$(\text{fdecode-universal-time } \text{universal-time } [\text{time-zone} \underline{\text{current}}])$

$(\text{fget-decoded-time})$

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

$(\text{fshort-site-name})$

(flong-site-name)

▷ String representing physical location of computer.

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

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

$(\text{fmachine-instance})$ ▷ Computer name.

DOUBLE-FLOAT- NEGATIVE-EPSILON 6	FUNCTION-LAMBDA- EXPRESSION 18	LEAST-NEGATIVE- NORMALIZED- SHORT-FLOAT 6	MAKE-STRING- INPUT-STREAM 39
DOWNFROM 21	FUNCTIONP 15	LEAST-NEGATIVE- NORMALIZED- SINGLE-FLOAT 6	MAKE-STRING- OUTPUT-STREAM 39
DRIBBLE 45	GENERIC-FUNCTION 31	LEAST-NEGATIVE- SINGLE-FLOAT 6	MAKE-SYNONYM- STREAM 39
DYNAMIC-EXTENT 46	GENSYM 43	LEAST-NEGATIVE- SINGLE-FLOAT 6	MAKE-TWO- WAY-STREAM 39
EACH 21	GENTEMP 43	LEAST-POSITIVE- DOUBLE-FLOAT 6	MAKUNBOUND 16
ECASE 20	GET 16	LEAST-POSITIVE- LONG-FLOAT 6	MAP 14
ECHO-STREAM 31	GET-DECODED-TIME 47	LEAST-POSITIVE- LONG-FLOAT 6	MAP-INTO 14
ECHO-STREAM- INPUT-STREAM 39	GET- DISPATCH-MACRO- CHARACTER 33	LEAST-POSITIVE- NORMALIZED- DOUBLE-FLOAT 6	MAPC 9
ECHO-STREAM- OUTPUT-STREAM 39	GET-INTERNAL- REAL-TIME 47	LEAST-POSITIVE- NORMALIZED- LONG-FLOAT 6	MAPCAN 9
ED 45	GET-INTERNAL- RUN-TIME 47	LEAST-POSITIVE- NORMALIZED- SHORT-FLOAT 6	MAPCAR 9
EIGHTH 8	GET-MACRO- CHARACTER 33	LEAST-POSITIVE- NORMALIZED- SINGLE-FLOAT 6	MAPCON 9
ELSE 23	GET-OUTPUT- STREAM-STRING 39	LEAST-POSITIVE- NORMALIZED- SHORT-FLOAT 6	MAPHASH 14
ELT 12	GET-PROPERTIES 16	LEAST-POSITIVE- NORMALIZED- SINGLE-FLOAT 6	MAPL 9
ENCODE-UNIVERSAL- TIME 47	GET-SETF- EXPANSION 19	LEAST-POSITIVE- SHORT-FLOAT 6	MAPLIST 9
END 23	GET-UNIVERSAL- TIME 47	LEAST-POSITIVE- SINGLE-FLOAT 6	MASK-FIELD 5
END-OF-FILE 31	GETF 16	LENGTH 12	MAX 4, 26
ENDP 8	GETHASH 14	LET 16	MAXIMIZE 23
ENOUGH-	GO 20	LET* 16	MAXIMIZING 23
NAMESTRING 41	GRAPHIC-CHAR-P 6	LISP-	MEMBER 8, 30
ENSURE-	HANDLER-BIND 28	IMPLEMENTATION- TYPE 47	MEMBER-IF 8
DIRECTORIES- EXIST 42	HANDLER-CASE 28	LISP-	MEMBER-IF-NOT 8
ENSURE-GENERIC- FUNCTION 25	HASH-KEY 21, 23	IMPLEMENTATION- VERSION 47	MERGE 12
EQ 15	HASH-KEYS 21	LIST 8, 26, 31	MERGE-PATHNAMES 41
EQL 15, 30	HASH-TABLE 31	LIST-ALL-PACKAGES 42	METHOD 31
EQUAL 15	HASH-TABLE-COUNT 14	LIST-LENGTH 8	METHOD- COMBINATION 31, 44
EQUALP 15	HASH-TABLE-P 14	LIST* 8	METHOD- COMBINATION- ERROR 26
ERROR 28, 31	HASH-TABLE- REHASH-SIZE 14	LISTEN 39	METHOD- QUALIFIERS 26
ETYPESCASE 30	HASH- TABLE-REHASH- THRESHOLD 14	LISTP 8	MIN 4, 26
EVAL 45	HASH-TABLE-SIZE 14	LOAD 44	MINIMIZE 23
EVAL-WHEN 45	HASH-TABLE-TEST 14	LOAD-LOGICAL- PATHNAME- TRANSLATIONS 41	MINIMIZING 23
EVENP 3	HASH-VALUES 23	LOAD-TIME-VALUE 45	MINUSP 3
EVERY 12	HOST-NAMESTRING 41	LOCALLY 45	MISMATCH 12
EXP 3	IDENTITY 17	LOG 3	MOD 4, 30
EXPORT 43	IF 19, 23	LOGAND 5	MOST-NEGATIVE- DOUBLE-FLOAT 6
EXPT 3	IGNORABLE 46	LOGANDC1 5	MOST-NEGATIVE- FIXNUM 6
EXTENDED-CHAR 31	IGNORE 46	LOGANDC2 5	MOST-NEGATIVE- LONG-FLOAT 6
EXTERNAL-SYMBOL 23	IGNORE-ERRORS 28	LOGBITP 5	MOST-NEGATIVE- SHORT-FLOAT 6
EXTERNAL-SYMBOLS 23	IMAGPART 4	LOGCOUNT 5	MOST-NEGATIVE- SINGLE-FLOAT 6
FBOUNDP 16	IMPORT 43	LOGEQV 5	MOST-POSITIVE- DOUBLE-FLOAT 6
FCEILING 4	IN 21, 23	LOGICAL-PATHNAME 31, 41	MOST-POSITIVE- DOUBLE-FLOAT 6
FDEFINITION 18	IN-PACKAGE 42	LOGICAL-PATHNAME- TRANSLATIONS 41	MOST-POSITIVE- SHORT-FLOAT 6
FFLOOR 4	INCF 3	LOGIOR 5	MOST-POSITIVE- SINGLE-FLOAT 6
FIFTH 8	INITIALIZE-INSTANCE 24	LOGNOT 5	MUFFLE-WARNING 29
FILE-AUTHOR 41	INITIALLY 23	LOGORC1 5	MULTIPLE- VALUE-BIND 16
FILE-ERROR 31	INLINE 47	LOGORC2 5	MULTIPLE- VALUE-CALL 17
FILE-ERROR- PATHNAME 29	INPUT-STREAM-P 32	LOGXOR 5	MULTIPLE- VALUE-LIST 17
FILE-LENGTH 41	INSPECT 46	LOGTEST 5	MULTIPLE- VALUE-PROG1 20
FILE-NAMESTRING 41	INTEGER 31	LONG-FLOAT 31, 34	MULTIPLE- VALUE-SETQ 16
FILE-POSITION 39	INTEGER- DECODE-FLOAT 6	LONG- FLOAT-EPSILON 6	MULTIPLE- VALUES-LIMIT 18
FILE-STREAM 31	INTEGER-LENGTH 5	LONG-FLOAT- NEGATIVE-EPSILON 6	NAME-CHAR 7
FILE-STRING-LENGTH 39	INTERACTIVE- STREAM-P 32	LONG-SITE-NAME 47	NAMED 21
FILE-WRITE-DATE 41	INTERN 43	LOOP 21	NAMESTRING 41
FILL 12	INTERNAL- TIME-UNITS- PER-SECOND 47	LOOP-FINISH 23	NBUTLAST 9
FILL-POINTER 11	INTERSECTION 10	LOWER-CASE-P 6	NCNC 9, 23, 26
FINALLY 23	INTO 23	MACHINE-INSTANCE 47	NCNCING 23
FIND 13	INVALID-METHOD- ERROR 26	MACHINE-TYPE 47	NEVER 23
FIND-ALL-SYMBOLS 42	INVOKE-DEBUGGER 28	MACHINE-VERSION 47	NEWLINE 6
FIND-CLASS 24	INVOKE-RESTART 29	MACRO-FUNCTION 45	NEXT-METHOD-P 25
FIND-IF 13	INVOKE-RESTART- INTERACTIVELY 29	MACROEXPAND 46	NIL 2, 44
FIND-IF-NOT 13	ISQRT 3	MACROEXPAND-1 46	NINTERSECTION 10
FIND-METHOD 26	IT 23	MACROLET 18	NINTH 8
FIND-PACKAGE 42	KEYWORD 31, 42, 44	MAKE-ARRAY 10	NO-APPLICABLE- METHOD 26
FIND-RESTART 29	KEYWORDP 42	MAKE-BROADCAST- STREAM 39	NO-NEXT-METHOD 26
FIND-SYMBOL 43	LABELS 17	MAKE- CONCATENATED- STREAM 39	NOT 15, 30, 34
FINISH-OUTPUT 39	LAMBDA 17	MAKE-CONDITION 28	NOTANY 12
FIRST 8	LAMBDA-LIST- KEYWORDS 19	MAKE- DISPATCH-MACRO- CHARACTER 33	NOTEVERY 12
FIXNUM 31	LAMBDA- PARAMETERS- LIMIT 18	MAKE- ECHO-STREAM 39	NOTINLINE 47
FLET 17	LAST 8	MAKE-HASH-TABLE 14	NRECONC 9
FLOAT 4, 31	LCM 3	MAKE-INSTANCE 24	NREVERSE 12
FLOAT-DIGITS 6	LDB 5	MAKE-INSTANCES- OBSOLETE 24	NSET-DIFFERENCE 10
FLOAT-PRECISION 6	LDB-TEST 5	MAKE-INSTANCE- OBSOLETE 24	NSET-EXCLUSIVE-OR 10
FLOAT-RADIX 6	LDIFF 9	MAKE-INSTANCE- OBSOLETE 24	NSTRING-CAPITALIZE 7
FLOAT-SIGN 4	LEAST-NEGATIVE- DOUBLE-FLOAT 6	MAKE-LIST 8	NSTRING-DOWNCASE 7
FLOATING- POINT-INEXACT 31	LEAST-NEGATIVE- LONG-FLOAT 6	MAKE-LOAD-FORM 45	NSTRING-UPCASE 7
FLOATING- POINT-INVALID- OPERATION 31	LEAST-NEGATIVE- LONG-FLOAT 6	MAKE-LOAD-FORM- SAVING-SLOTS 45	NSUBS1 10
FLOATING-POINT- OVERFLOW 31	LEAST-NEGATIVE- NORMALIZED- DOUBLE-FLOAT 6	MAKE-METHOD 42	NSUBST 10
FLOATING-POINT- UNDERFLOW 31	LEAST-NEGATIVE- NORMALIZED- DOUBLE-FLOAT 6	MAKE-PACKAGE 42	NSUBST-IF 10
FLOATP 3	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE-PATHNAME 40	NSUBST-IF-NOT 10
FLOOR 4	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	NSUBSTITUTE 13
FMAKUNBOUND 18	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	NSUBSTITUTE-IF 13
FOR 21	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	NSUBSTITUTE-IF- IF-NOT 13
FORCE-OUTPUT 39	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	NTH 8
FORMAT 36	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	NTH-VALUE 17
FORMATTER 36	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	
FOURTH 8	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	
FRESH-LINE 34	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	
FROM 21	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	
FROUND 4	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	
FTRUNCATE 4	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	
FTYPE 46	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	
FUNCALL 17	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	
FUNCTION 17, 31, 34, 44	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	
FUNCTION- KEYWORDS 26	LEAST-NEGATIVE- NORMALIZED- LONG-FLOAT 6	MAKE- RANDOM-STATE 4	



NTHCDR 8	READ-CHAR-NO-HANG 32	SINGLE-FLOAT-NEGATIVE-EPSILON 6	THROW 20
NULL 8, 31	READ-DELIMITED-LIST 32	SINH 3	TIME 46
NUMBER 31	READ-FROM-STRING 32	SIXTH 8	TO 21
NUMBERP 3	READ-LINE 32	SLEEP 20	TRACE 46
NUMERATOR 4	READ-PRESERVING-WHITESPACE 32	SLOT-BOUND 24	TRANSLATE-LOGICAL-PATHNAME 41
NUNION 10	READ-SEQUENCE 33	SLOT-EXISTS-P 24	TRANSLATE-PATHNAME 41
	READER-ERROR 31	SLOT-MAKUNBOUND 24	TREE-EQUAL 10
	READTABLE 31	SLOT-MISSING 25	TRUENAME 41
	READTABLE-CASE 33	SLOT-UNBOUND 25	TRUNCATE 4
	READTABLEP 32	SLOT-VALUE 24	TWO-WAY-STREAM 31
	REAL 31	SOFTWARE-TYPE 47	TWO-WAY-STREAM-INPUT-STREAM 39
	REALP 3	SOFTWARE-VERSION 47	TWO-WAY-STREAM-OUTPUT-STREAM 39
	REALPART 4	SOME 12	TYPE 44, 46
	REDUCE 14	SORT 12	TYPE-ERROR 31
	REINITIALIZE-INSTANCE 24	SPACE 6, 47	TYPE-ERROR-DATUM 29
	REM 4	SPECIAL 47	TYPE-ERROR-EXPECTED-TYPE 29
	REMF 16	SPECIAL-OPERATOR-P 44	TYPE-OF 30
	REMHASH 14	SPEED 47	TYPECASE 30
	REMOVE 13	SQRT 3	TYPEP 30
	REMOVE-DUPLICATES 13	STABLE-SORT 12	
	REMOVE-IF 13	STANDARD 26	
	REMOVE-IF-NOT 13	STANDARD-CHAR 6, 31	
	REMOVE-METHOD 26	STANDARD-CHAR-P 6	
	REMPROP 16	STANDARD-CLASS 31	UNBOUND-SLOT 31
	RENAME-FILE 41	STANDARD-GENERIC-FUNCTION 31	UNBOUND-SLOT-INSTANCE 29
	RENAME-PACKAGE 42	STANDARD-METHOD 31	UNBOUND-VARIABLE 31
	REPEAT 23	STANDARD-OBJECT 31	UNDEFINED-FUNCTION 31
	REPLACE 14	STEP 46	UNEXPORT 43
	REQUIRE 43	STORAGE-CONDITION 31	UNINTERN 43
	REST 8	STORE-VALUE 29	UNION 10
	RESTART 31	STREAM 31	UNLESS 19, 23
	RESTART-BIND 29	STREAM-ELEMENT-TYPE 30	UNREAD-CHAR 32
	RESTART-CASE 28	STREAM-ERROR 31	UNSIGNED-BYTE 31
	RESTART-NAME 29	STREAM-ERROR-STREAM 29	UNTIL 23
	RETURN 20, 23	STREAM-EXTERNAL-FORMAT 40	UNTRACE 46
	RETURN-FROM 20	STREAMP 32	UNUSE-PACKAGE 42
	REVAPPEND 9	STRING 7, 31	UNWIND-PROTECT 20
	REVERSE 12	STRING-CAPITALIZE 7	UPDATE-INSTANCE-FOR-DIFFERENT-CLASS 24
	ROOM 46	STRING-DOWNCASE 7	UPDATE-INSTANCE-FOR-REDEFINED-CLASS 25
	ROTATEF 16	STRING-EQUAL 7	UPFROM 21
	ROUND 4	STRING-GREATERP 7	UPGRADED-ARRAY-ELEMENT-TYPE 30
	ROW-MAJOR-AREF 10	STRING-LEFT-TRIM 7	UPGRADED-COMPLEX-PART-TYPE 6
	RPLACA 9	STRING-LESSP 7	UPPER-CASE-P 6
	RPLACD 9	STRING-NOT-EQUAL 7	UPTO 21
		STRING-NOT-LESSP 7	USE-PACKAGE 42
		STRING-NOT-TRIM 7	USE-VALUE 29
		STRING-STREAM 31	USER-HOMEDIR-PATHNAME 41
		STRING-TRIM 7	USING 21, 23
		STRING-UPCASE 7	
		STRING/= 7	
		STRING< 7	V 38
		STRING<= 7	VALUES 17, 30
		STRING= 7	VALUES-LIST 17
		STRING> 7	VARIABLE 44
		STRING>= 7	VECTOR 11, 31
		STRINGP 7	VECTOR-POP 11
		STRUCTURE 44	VECTOR-PUSH 11
		STRUCTURE-CLASS 31	VECTOR-PUSH-EXTEND 11
		STRUCTURE-OBJECT 31	VECTOP 10
		STYLE-WARNING 31	
		SUBLIS 10	WARN 28
		SUBSEQ 12	WARNING 31
		SUBSETP 8	WHEN 19, 23
		SUBST 10	WHILE 23
		SUBST-IF 10	WILD-PATHNAME-P 32
		SUBST-IF-NOT 10	WITH 21
		SUBSTITUTE 13	WITH-ACCESSORS 24
		SUBSTITUTE-IF 13	WITH-COMPILE-UNIT 45
		SUBSTITUTE-IF-NOT 13	WITH-CONDITION-RESTARTS 29
		SUBTYPEP 30	WITH-HASH-TABLE-ITERATOR 14
		SUM 23	WITH-INPUT-FROM-STRING 40
		SUMMING 23	WITH-OPEN-FILE 40
		SVREF 11	WITH-OPEN-STREAM 40
		SVHASH 14	WITH-OUTPUT-TO-STRING 40
		SYMBOL 23, 31, 43	WITH-PACKAGE-ITERATOR 43
		SYMBOL-FUNCTION 44	WITH-SIMPLE-RESTART 28
		SYMBOL-MACROLET 18	WITH-SLOTS 24
		SYMBOL-NAME 44	WITH-STANDARD-IO-SYNTAX 32
		SYMBOL-PACKAGE 44	WRITE 35
		SYMBOL-PLIST 44	WRITE-BYTE 35
		SYMBOL-VALUE 44	WRITE-CHAR 35
		SYMBOLP 42	WRITE-LINE 35
		SYMBOLS 23	WRITE-SEQUENCE 35
		SYNONYM-STREAM 31	WRITE-STRING 35
		SYNONYM-STREAM-SYMBOL 39	WRITE-TO-STRING 35
		T 2, 31, 44	
		TAGBODY 20	Y-OR-N-P 32
		TAILP 8	YES-OR-NO-P 32
		TAN 3	
		TANH 3	
		TENTH 8	
		TERPRI 34	
		THE 21, 30	
		THEN 21	
		THEREIS 23	
		THIRD 8	
			ZEROP 3