

- (*atan* *a* [*b*])      ▷  $\arctan \frac{a}{b}$  in radians.
- (*sinh* *a*)  
(*cosh* *a*)      ▷ sinh *a*, cosh *a*, or tanh *a*, respectively.  
(*tanh* *a*)
- (*asinh* *a*)  
(*acosh* *a*)      ▷ asinh *a*, acosh *a*, or atanh *a*, respectively.  
(*atanh* *a*)
- (*cis* *a*)      ▷ Return  $e^{i a} = \cos a + i \sin a$ .
- (*conjugate* *a*)      ▷ Return complex conjugate of *a*.
- (*max* *num*<sup>+</sup>)  
(*min* *num*<sup>+</sup>)      ▷ Greatest or least, respectively, of *nums*.
- (*round* | *floor* | *ceiling* | *truncate*) *n* [*d*])  
    {  
    {*round* | *floor* | *ceiling* | *truncate*}  
    } *n* [*d*])  
    ▷ Return as **integer** or **float**, respectively,  $\frac{n}{d}$  rounded, or rounded towards  $-\infty$ ,  $+\infty$ , or 0, respectively; and remainder.
- (*mod* | *rem*) *n* *d*)  
    {  
    {*mod* | *rem*}  
    } *n* *d*)  
    ▷ Same as *floor* or *truncate*, respectively, but return remainder only.
- (*random* *limit* [*state* [*\*random-state\**]])  
    ▷ Return non-negative random number less than *limit*, and of the same type.
- (*make-random-state* [*state* [NIL | T] [*num*]])  
    ▷ Copy of **random-state** object *state* or of the current random state; or a randomly initialized fresh random state.
- \*random-state\**      ▷ Current random state.
- (*float-sign* *num-a* [*num-b*])      ▷ num-b with *num-a*'s sign.
- (*signum* *n*)  
    ▷ Number of magnitude 1 representing sign or phase of *n*.
- (*numerator* *rational*)  
(*denominator* *rational*)  
    ▷ Numerator or denominator, respectively, of *rational*'s canonical form.
- (*realpart* *number*)  
(*imagpart* *number*)  
    ▷ Real part or imaginary part, respectively, of *number*.
- (*complex* *real* [*imag*])      ▷ Make a complex number.
- (*phase* *num*)      ▷ Angle of *num*'s polar representation.
- (*abs* *n*)      ▷ Return |n|.
- (*rational* *real*)  
(*rationalize* *real*)  
    ▷ Convert *real* to rational. Assume complete/limited accuracy for *real*.
- (*float* *real* [*prototype* [*type*]])  
    ▷ Convert *real* into float with type of *prototype*.

## Quick Reference



## Common

---

# lisp

---

Bert Burgemeister

## Contents

<b>1</b>	<b>Numbers</b>	<b>3</b>	9.5 Control Flow . . . . .	21
1.1	Predicates . . . . .	3	9.6 Iteration . . . . .	22
1.2	Numeric Functions . . . . .	3	9.7 Loop Facility . . . . .	22
1.3	Logic Functions . . . . .	5	<b>10 CLOS</b>	<b>25</b>
1.4	Integer Functions . . . . .	6	10.1 Classes . . . . .	25
1.5	Implementation-Dependent . . . . .	6	10.2 Generic Functions . . . . .	26
<b>2</b>	<b>Characters</b>	<b>7</b>	10.3 Method Combination Types . . . . .	28
<b>3</b>	<b>Strings</b>	<b>8</b>	<b>11 Conditions and Errors</b>	<b>29</b>
<b>4</b>	<b>Conses</b>	<b>8</b>	<b>12 Types and Classes</b>	<b>31</b>
4.1	Predicates . . . . .	8	<b>13 Input/Output</b>	<b>33</b>
4.2	Lists . . . . .	9	13.1 Predicates . . . . .	33
4.3	Association Lists . . . . .	10	13.2 Reader . . . . .	34
4.4	Trees . . . . .	10	13.3 Character Syntax . . . . .	35
4.5	Sets . . . . .	11	13.4 Printer . . . . .	36
<b>5</b>	<b>Arrays</b>	<b>11</b>	13.5 Format . . . . .	38
5.1	Predicates . . . . .	11	13.6 Streams . . . . .	41
5.2	Array Functions . . . . .	11	13.7 Pathnames and Files . . . . .	42
5.3	Vector Functions . . . . .	12	<b>14 Packages and Symbols</b>	<b>44</b>
<b>6</b>	<b>Sequences</b>	<b>12</b>	14.1 Predicates . . . . .	44
6.1	Sequence Predicates . . . . .	12	14.2 Packages . . . . .	44
6.2	Sequence Functions . . . . .	13	14.3 Symbols . . . . .	45
<b>7</b>	<b>Hash Tables</b>	<b>15</b>	14.4 Standard Packages . . . . .	46
<b>8</b>	<b>Structures</b>	<b>16</b>	<b>15 Compiler</b>	<b>46</b>
<b>9</b>	<b>Control Structure</b>	<b>16</b>	15.1 Predicates . . . . .	46
9.1	Predicates . . . . .	16	15.2 Compilation . . . . .	46
9.2	Variables . . . . .	17	15.3 REPL and Debugging . . . . .	48
9.3	Functions . . . . .	18	15.4 Declarations . . . . .	49
9.4	Macros . . . . .	19	<b>16 External Environment</b>	<b>49</b>

## 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*<sub>bar</sub>] ▷ Either one *foo* or nothing; defaults to **bar**.

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

*foo*<sup>+</sup>; {*foo*}<sup>+</sup> ▷ One or more *foos*.

*foos* ▷ English plural denotes a list argument.

{*foo*|*bar*|*baz*}; {*foo*<sub>bar</sub>|*baz*} ▷ Either *foo*, or *bar*, or *baz*.

{*foo*<sub>bar</sub>|*baz*} ▷ Anything from none to each of *foo*, *bar*, and *baz*.

*foo* ▷ Argument *foo* is not evaluated.

*bar* ▷ Argument *bar* is possibly modified.

*foo*<sup>P</sup> ▷ *foo*\* is evaluated as in **sprogn**; see page 21.

*foo*<sub>2</sub>; *bar*<sub>2</sub>; *baz*<sub>n</sub> ▷ Primary, secondary, and *n*th return value.

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

## 1 Numbers

### 1.1 Predicates

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

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

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

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

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

### 1.2 Numeric Functions

(*f* + *a*<sub>□</sub><sup>\*</sup>) ▷ Return  $\sum a$  or  $\prod a$ , respectively.  
(*f* \* *a*<sub>□</sub><sup>\*</sup>)

(*f* - *a* *b*<sup>\*</sup>)  
(*f* / *a* *b*<sup>\*</sup>)  
▷ 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*)

{*m***incf**} *place* [*delta*<sub>□</sub>] (*m***decf**)  
▷ Increment or decrement the value of *place* by *delta*. Return new value.

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

(*f* **log** *a* [*b*<sub>□</sub>]) ▷ Return  $\log_b a$  or, without *b*,  $\ln a$ .

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

(*f* **lcm** *integer*<sup>\*</sup><sub>□</sub>)  
(*f* **gcd** *integer*<sup>\*</sup><sub>□</sub>)  
▷ Least common multiple or greatest common denominator, respectively, of *integers*. (**gcd**) returns 0.

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

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

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

## 3 Strings

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

(*fstringp* *foo*)  
(*fstring=* *foo* *bar*)    ▷ T if *foo* is of indicated type.

(*fstring=* *foo* *bar*)  
(*fstring-equal* *foo* *bar*)

$$\left. \begin{array}{l} \text{:start1 } \textit{start-foo}_{\text{0}} \\ \text{:start2 } \textit{start-bar}_{\text{0}} \\ \text{:end1 } \textit{end-foo}_{\text{NIL}} \\ \text{:end2 } \textit{end-bar}_{\text{NIL}} \end{array} \right\}$$

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

(*fstring*{*/=* |*-not-equal*})  
(*fstring*{*>* |*-greaterp*})  
(*fstring*{*>=* |*-not-lessp*})  
(*fstring*{*<* |*-lessp*})  
(*fstring*{*<=* |*-not-greaterp*})

$$\left. \begin{array}{l} \text{:start1 } \textit{start-foo}_{\text{0}} \\ \text{:start2 } \textit{start-bar}_{\text{0}} \\ \text{:end1 } \textit{end-foo}_{\text{NIL}} \\ \text{:end2 } \textit{end-bar}_{\text{NIL}} \end{array} \right\}$$

▷ 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.

(*fmake-string* *size* {*:initial-element* *char*  
*:element-type* *type*<sub>*character*</sub>})

▷ Return string of length *size*.

(*fstring* *x*)  
(*fstring-capitalize* *x*)  
(*fstring-upcase* *x*)  
(*fstring-downcase* *x*)

$$\left. \begin{array}{l} \text{:start } \textit{start}_{\text{0}} \\ \text{:end } \textit{end}_{\text{NIL}} \end{array} \right\}$$

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

(*fstring-capitalize* *string*)  
(*fstring-upcase* *string*)  
(*fstring-downcase* *string*)

$$\left. \begin{array}{l} \text{:start } \textit{start}_{\text{0}} \\ \text{:end } \textit{end}_{\text{NIL}} \end{array} \right\}$$

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

(*fstring-trim* *char-bag* *string*)  
(*fstring-left-trim* *string*)  
(*fstring-right-trim* *string*)

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

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

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

(*fparse-integer* *string*)

$$\left. \begin{array}{l} \text{:start } \textit{start}_{\text{0}} \\ \text{:end } \textit{end}_{\text{NIL}} \\ \text{:radix } \textit{int}_{\text{10}} \\ \text{:junk-allowed } \textit{bool}_{\text{NIL}} \end{array} \right\}$$

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

## 4 Conses

### 4.1 Predicates

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

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

## 1.3 Logic Functions

Negative integers are used in two's complement representation.

(*fboole* *operation* *int-a* *int-b*)  
▷ Return value of bitwise logical *operation*. *operations* are

*cboole-1*    ▷ *int-a*.  
*cboole-2*    ▷ *int-b*.  
*cboole-c1*    ▷  $\neg$ *int-a*.  
*cboole-c2*    ▷  $\neg$ *int-b*.  
*cboole-set*    ▷ All bits set.  
*cboole-clr*    ▷ All bits zero.  
*cboole-eqv*    ▷  $int-a \equiv int-b$ .  
*cboole-and*    ▷  $int-a \wedge int-b$ .  
*cboole-andc1*    ▷  $\neg int-a \wedge int-b$ .  
*cboole-andc2*    ▷  $int-a \wedge \neg int-b$ .  
*cboole-nand*    ▷  $\neg(int-a \wedge int-b)$ .  
*cboole-ior*    ▷  $int-a \vee int-b$ .  
*cboole-orc1*    ▷  $\neg int-a \vee int-b$ .  
*cboole-orc2*    ▷  $int-a \vee \neg int-b$ .  
*cboole-xor*    ▷  $\neg(int-a \equiv int-b)$ .  
*cboole-nor*    ▷  $\neg(int-a \vee int-b)$ .

(*flognot* *integer*)    ▷  $\neg$ *integer*.

(*flogeqv* *integer\**)  
(*flogand* *integer\**)  
▷ Return value of exclusive-nored or anded integers, respectively. Without any *integer*, return -1.

(*flogandc1* *int-a* *int-b*)    ▷  $\neg int-a \wedge int-b$ .

(*flogandc2* *int-a* *int-b*)    ▷  $int-a \wedge \neg int-b$ .

(*flognand* *int-a* *int-b*)    ▷  $\neg(int-a \wedge int-b)$ .

(*flogxor* *integer\**)  
(*flogior* *integer\**)  
▷ Return value of exclusive-ored or ored integers, respectively. Without any *integer*, return 0.

(*flogorc1* *int-a* *int-b*)    ▷  $\neg int-a \vee int-b$ .

(*flogorc2* *int-a* *int-b*)    ▷  $int-a \vee \neg int-b$ .

(*flognor* *int-a* *int-b*)    ▷  $\neg(int-a \vee int-b)$ .

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

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

(*flogcount* *int*)  
▷ Number of 1 bits in *int*  $\geq 0$ , 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**.

{(*f*deposit-field  
*f*dppb)} *int-a byte-spec int-b*

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

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

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

(*f*byte *size position*)

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

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

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

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

## 1.5 Implementation-Dependent

{*c*short-float  
*c*single-float  
*c*double-float  
*c*long-float} {epsilon  
negative-epsilon}

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

{*c*least-negative  
*c*least-negative-normalized  
*c*least-positive  
*c*least-positive-normalized} {short-float  
single-float  
double-float  
long-float}

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

{*c*most-negative  
*c*most-positive} {short-float  
single-float  
double-float  
long-float  
fixnum}

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

(*f*decode-float *n*)

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

▷ Return significand, exponent, and sign of **float** *n*.

(*f*scale-float *n i*)

▷ With *n*'s radix *b*, return  $nb^i$ .

(*f*float-radix *n*)

(*f*float-digits *n*)

(*f*float-precision *n*)

▷ Radix, number of digits in that radix, or precision in that radix, respectively, of float *n*.

(*f*upgraded-complex-part-type *foo* [*environment*<sub>ENV</sub>])

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

## 2 Characters

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

(*f*characterp *foo*)

(*f*standard-char-p *char*) ▷ T if argument is of indicated type.

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

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

(*f*alphanumericp *character*)

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

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

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

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

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

(*f*digit-char-p *character* [*radix*<sub>10</sub>])

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

(*f*char= *character*<sup>+</sup>)

(*f*char/= *character*<sup>+</sup>)

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

(*f*char-equal *character*<sup>+</sup>)

(*f*char-not-equal *character*<sup>+</sup>)

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

(*f*char> *character*<sup>+</sup>)

(*f*char>= *character*<sup>+</sup>)

(*f*char< *character*<sup>+</sup>)

(*f*char<= *character*<sup>+</sup>)

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

(*f*char-greaterp *character*<sup>+</sup>)

(*f*char-not-lessp *character*<sup>+</sup>)

(*f*char-lessp *character*<sup>+</sup>)

(*f*char-not-greaterp *character*<sup>+</sup>)

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

(*f*char-upcase *character*)

(*f*char-downcase *character*)

▷ Return corresponding uppercase/lowercase character, respectively.

(*f*digit-char *i* [*radix*<sub>10</sub>])

▷ Character representing digit *i*.

(*f*char-name *char*)

▷ *char*'s name if any, or NIL.

(*f*name-char *foo*)

▷ Character named *foo* if any, or NIL.

(*f*char-int *character*)

(*f*char-code *character*)

▷ Code of *character*.

(*f*code-char *code*)

▷ Character with *code*.

*c*char-code-limit

▷ Upper bound of (*f*char-code *char*);  $\geq 96$ .

(*f*character *c*)

▷ Return #\c.

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

(*f*bit *bit-array* [*subscripts*])  
 (*f*sbit *simple-bit-array* [*subscripts*])  
 ▷ Return element of *bit-array* or of *simple-bit-array*. **setfable**.

(*f*bit-not *bit-array* [*result-bit-array*])  
 ▷ 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*])  
 ▷ Return result of bitwise logical operations (cf. operations of *f*boole, page 5) on *bit-array-a* and *bit-array-b*. If *result-bit-array* is T, put result in *bit-array-a*; if it is NIL, make a new array for result.

*c*array-rank-limit    ▷ Upper bound of array rank;  $\geq 8$ .

*c*array-dimension-limit  
 ▷ Upper bound of an array dimension;  $\geq 1024$ .

*c*array-total-size-limit    ▷ Upper bound of array size;  $\geq 1024$ .

## 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*. **setfable**.

(*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*. **setfable**.

## 6 Sequences

### 6.1 Sequence Predicates

(*f*every  
*f*notevery

*test sequence*<sup>+</sup>)  
 ▷ Return NIL or T, respectively, as soon as *test* on any set of corresponding elements of *sequences* returns 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* {  
 :test *function*<sub>#=eq</sub>  
 :test-not *function*  
 :key *function*  
 })  
 ▷ Return tail of list starting with its first element matching *foo*. Return NIL if there is no such element.

(*f*member-if  
*f*member-if-not

*test list* [:key *function*])  
 ▷ 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* {  
 :test *function*<sub>#=eq</sub>  
 :test-not *function*  
 :key *function*  
 })  
 ▷ 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*<sup>+</sup>)  
 ▷ Return list of foos with last *foo* becoming cdr of last cons. Return *foo* if only one *foo* given.

(*f*make-list *num* [:initial-element *foo*])  
 ▷ 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*)    ▷ Cdr of list or NIL if *list* is NIL. **setfable**.  
 (*f*rest *list*)

(*f*nthcdr *n list*)    ▷ Return tail of list after calling *f*cdr *n* times.

(*f*first|*f*second|*f*third|*f*fourth|*f*fifth|*f*sixth|...|*f*ninth|*f*tenth

*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*.

(*f*butlast *list*  
*f*nbutlast *list*) [*num*])    ▷ list excluding last *num* conses.

(*f*rplaca  
*f*rplacd

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

(*f*ldiff *list* *foo*)  
 ▷ If *foo* is a tail of *list*, return preceding part of list. Otherwise return *list*.

(*f*adjoin *foo* *list* {  
 :test *function*<sub>#=eq</sub>  
 :test-not *function*  
 :key *function*  
 })  
 ▷ Return *list* if *foo* is already member of *list*. If not, return (*f*cons *foo list*).

(*m*pop *place*)    ▷ Set *place* to (*f*cdr *place*), return (*f*car *place*).

(*m*push *foo* *place*) ▷ Set *place* to (*f*cons *foo* *place*).

(*m*pushnew *foo* *place*  $\left\{ \begin{array}{l} \text{:test function} \text{ \#'eq} \\ \text{:test-not function} \\ \text{:key function} \end{array} \right\}$ )  
▷ Set *place* to (*f*adjoin *foo* *place*).

(*f*append [*proper-list*\* *foo*  $\text{\#T}$ ])

(*f*nconc [*non-circular-list*\* *foo*  $\text{\#T}$ ])  
▷ Return concatenated list or, with only one argument, *foo*.  
*foo* can be of any type.

(*f*revappend *list* *foo*)

(*f*nreconc *list* *foo*)  
▷ Return concatenated list after reversing order in *list*.

$\left\{ \begin{array}{l} \text{\#f} \text{mapcar} \\ \text{\#f} \text{maplist} \end{array} \right\}$  *function* *list*<sup>+</sup>

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

$\left\{ \begin{array}{l} \text{\#f} \text{mapcan} \\ \text{\#f} \text{mapcon} \end{array} \right\}$  *function* *list*<sup>+</sup>

▷ 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.

$\left\{ \begin{array}{l} \text{\#f} \text{mapc} \\ \text{\#f} \text{mapl} \end{array} \right\}$  *function* *list*<sup>+</sup>

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

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

### 4.3 Association Lists

(*f*pairlis *keys* *values* [*alist*  $\text{\#T}$ ])

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

(*f*acons *key* *value* *alist*)

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

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

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

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

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

### 4.4 Trees

(*f*tree-equal *foo* *bar*  $\left\{ \begin{array}{l} \text{:test test} \text{ \#'eq} \\ \text{:test-not test} \end{array} \right\}$ )

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

$\left\{ \begin{array}{l} \text{\#f} \text{subst} \\ \text{\#f} \text{nsbst} \end{array} \right\}$  *new* *old* *tree*  $\left\{ \begin{array}{l} \text{:test function} \text{ \#'eq} \\ \text{:test-not function} \\ \text{:key function} \end{array} \right\}$

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

$\left\{ \begin{array}{l} \text{\#f} \text{subst-if[-not]} \\ \text{\#f} \text{nsbst-if[-not]} \end{array} \right\}$  *new* *test* *tree* [*key function*]

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

$\left\{ \begin{array}{l} \text{\#f} \text{sublis} \\ \text{\#f} \text{nsublis} \end{array} \right\}$  *association-list* *tree*  $\left\{ \begin{array}{l} \text{:test function} \text{ \#'eq} \\ \text{:test-not function} \\ \text{:key function} \end{array} \right\}$

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

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

### 4.5 Sets

$\left\{ \begin{array}{l} \text{\#f} \text{intersection} \\ \text{\#f} \text{set-difference} \\ \text{\#f} \text{union} \\ \text{\#f} \text{set-exclusive-or} \\ \text{\#f} \text{nintersection} \\ \text{\#f} \text{nset-difference} \\ \text{\#f} \text{nunion} \\ \text{\#f} \text{nset-exclusive-or} \end{array} \right\}$  *a* *b*  $\left\{ \begin{array}{l} \text{:test function} \text{ \#'eq} \\ \text{:test-not function} \\ \text{:key function} \end{array} \right\}$

▷ 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

$\left\{ \begin{array}{l} \text{\#f} \text{make-array} \\ \text{\#f} \text{adjust-array} \end{array} \right\}$  *dimension-sizes* [*adjustable* *bool*  $\text{\#T}$ ]

$\left\{ \begin{array}{l} \text{:element-type} \text{ type} \text{\#} \\ \text{:fill-pointer} \{ \text{num} \} \text{bool} \text{\#} \\ \text{:initial-element} \text{ obj} \\ \text{:initial-contents} \text{ tree-or-array} \\ \text{:displaced-to} \text{ array} \text{\#} [\text{:displaced-index-offset} \text{ i} \text{\#}] \end{array} \right\}$

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

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

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

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

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

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

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

(*f*array-dimensions *array*)

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

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

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

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

(*rxhash* *foo*) ▷ Hash code unique for any argument *foo*.

## 8 Structures

```
(mdefstruct
  (foo
    {
      (:conc-name
       (:conc-name [slot-prefixfoo-]))
      (:constructor
       (:constructor [makerMAKE-foo] [(ord-λ*)]))
      (:copier
       (:copier [copierCOPY-foo]))
      (:include struct
       {
         (slot
          (slot [init]
                {
                  (:type st-type)
                  (:read-only b)
                }
                ))
         (list
          (vector
           (vector type)))
         }
        [(initial-offset n)]
        )
      (:print-object [o-printer])
      (:print-function [f-printer])
      :named
      (:predicate [p-namefoo-P]))
    }
    (doc
     (slot [init]
           {
             (:type slot-type)
             (:read-only bool)
           }
           ))
  )
```

▷ Define structure *foo* together with functions *MAKE-foo*, *COPY-foo* and *foo-P*; and **setfable** accessors *foo-slot*. Instances are of class *foo* or, if **defstruct** option **:type** is given, of the specified type. They can be created by (*MAKE-foo* {*slot value*}\*) or, if *ord-λ* (see page 18) is given, by (*maker* *arg*\* {*key value*}\*). In the latter case, *args* and *keys* correspond to the positional and keyword parameters defined in *ord-λ* whose *vars* in turn correspond to *slots*. **:print-object**/**:print-function** generate a **gprint-object** method for an instance *bar* of *foo* calling (*o-printer* *bar stream*) or (*f-printer* *bar stream print-level*), respectively. If **:type** without **:named** is given, no *foo-P* is created.

(*copy-structure* *structure*)  
▷ Return copy of *structure* with shared slot values.

## 9 Control Structure

### 9.1 Predicates

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

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

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

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

{*some*  
*notany*} *test sequence*+  
▷ Return value of *test* or NIL, respectively, as soon as *test* on any set of corresponding elements of *sequences* returns non-NIL.

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

### 6.2 Sequence Functions

(*make-sequence* *sequence-type* *size* [**:initial-element** *foo*])  
▷ Make sequence of *sequence-type* with *size* elements.

(*concatenate* *type sequence*\*)  
▷ Return concatenated sequence of *type*.

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

(*fill* *sequence* *foo* {**:start** *start*  
**:end** *end*})  
▷ Return sequence after setting elements between *start* and *end* to *foo*.

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

(*count* *foo sequence* {**:from-end** *bool*  
**:test** *function*  
**:test-not** *function*  
**:start** *start*  
**:end** *end*  
**:key** *function*})  
▷ Return number of elements in *sequence* which match *foo*.

{*count-if*  
*count-if-not*} *test sequence* {**:from-end** *bool*  
**:start** *start*  
**:end** *end*  
**:key** *function*})  
▷ Return number of elements in *sequence* which satisfy *test*.

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

(*subseq* *sequence* *start* [*end*])  
▷ Return subsequence of *sequence* between *start* and *end*. **setfable**.

{*sort*  
*stable-sort*} *sequence test* [**:key** *function*]  
▷ Return sequence sorted. Order of elements considered equal is not guaranteed/retained, respectively.

(*reverse* *sequence*)  
▷ Return sequence in reverse order.

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

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

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

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

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

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

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

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

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

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

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

▷ Make copy of sequence without duplicates.

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

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

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

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

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

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

(*f*map *type function sequence*<sup>+</sup>)

▷ 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*<sup>\*</sup>)

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

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

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

(*f*copy-seq *sequence*)

▷ Copy of sequence with shared elements.

## 7 Hash Tables

The Loop Facility provides additional hash table-related functionality; see **loop**, page 22.

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

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

$$\left. \begin{array}{l} \{ \text{make-hash-table} \} \end{array} \right\} \left. \begin{array}{l} \text{:test } \{ \text{f} \text{eq} | \text{f} \text{equal} | \text{f} \text{equalp} \}_{\neq \text{eq}} \\ \text{:size } \text{int} \\ \text{:rehash-size } \text{num} \\ \text{:rehash-threshold } \text{num} \end{array} \right\}$$

▷ Make a hash table.

(*f*gethash *key hash-table* [*default*]<sub>NIL</sub>)

▷ 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*clearhash *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*<sup>\*</sup>)<sup>\*</sup> *form*<sup>Ⓔ</sup>)

▷ 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.



(*m*define-symbol-macro *foo* *form*)  
 ▷ Define symbol macro foo which on evaluation evaluates expanded *form*.

(*s*macrolet ((*foo* (*macro-λ\**)  $\left\{ \left( \text{declare } \widehat{\text{local-decl}}^* \right)^* \right\}$  *macro-form*<sup>P<sub>k</sub></sup>\*)  
 (*declare*  $\widehat{\text{decl}}^*$ )<sup>P<sub>k</sub></sup> *form*<sup>P<sub>k</sub></sup>)  
 ▷ Evaluate forms with locally defined mutually invisible macros *foo* which are enclosed in implicit *s*blocks of the same name.

(*s*symbol-macrolet ((*foo* *expansion-form*\*) (*declare*  $\widehat{\text{decl}}^*$ )<sup>P<sub>k</sub></sup> *form*<sup>P<sub>k</sub></sup>\*)  
 ▷ Evaluate forms with locally defined symbol macros *foo*.

(*m*defsetf *function*  $\left\{ \widehat{\text{updater}} \left[ \widehat{\text{doc}} \right] \left( \text{setf-}\lambda^* \right) \left( s\text{-var}^* \right) \left\{ \left( \text{declare } \widehat{\text{decl}}^* \right)^* \right\} \text{form}^{\text{P}_k} \right\}$

where defsetf lambda list (*setf-λ\**) has the form

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

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

[*&allow-other-keys*] [*&environment* *var*])

▷ Specify how to **setf** a place accessed by *function*. **Short form:** (**setf** (*function* *arg*\*) *value-form*) is replaced by (*updater* *arg*\* *value-form*); the latter must return *value-form*. **Long form:** on invocation of (**setf** (*function* *arg*\*) *value-form*), *forms* must expand into code that sets the place accessed where *setf-λ* and *s-var*\* describe the arguments of *function* and the value(s) to be stored, respectively; and that returns the value(s) of *s-var*\*. *forms* are enclosed in an implicit *s*block named *function*.

(*m*define-setf-expander *function* (*macro-λ\**)  $\left\{ \left( \text{declare } \widehat{\text{decl}}^* \right)^* \right\}$  *form*<sup>P<sub>k</sub></sup>)

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

(*f*get-setf-expansion *place* [*environment*<sub>NIL</sub>])

▷ 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*  $\left\{ \text{var} \left( \text{var} \left[ \text{init}_{\text{NIL}} \left[ \text{supplied-p} \right] \right] \right)^* \right\}$ ] [*&rest* *var*]) *function* [*doc*])

▷ Define macro *foo* able to modify a place. On invocation of (*foo* *place* *arg*\*), the value of *function* applied to *place* and *args* will be stored into *place* and returned.

#### λlambda-list-keywords

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

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

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

**{&rest|&body}** *var*  
 ▷ Bind *var* to a list of remaining arguments.

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

(*f*not *foo*) ▷ T if *foo* is NIL; NIL otherwise.

(*f*boundp *symbol*) ▷ T if *symbol* is a special variable.

(*f*constantp *foo* [*environment*<sub>NIL</sub>])  
 ▷ T if *foo* is a constant form.

(*f*functionp *foo*) ▷ T if *foo* is of type **function**.

(*f*fboundp  $\left\{ \text{foo} \left( \text{setf } \text{foo} \right) \right\}$ ) ▷ T if *foo* is a global function or macro.

## 9.2 Variables

$\left\{ \begin{array}{l} m\text{defconstant} \\ m\text{defparameter} \end{array} \right\} \widehat{\text{foo}} \text{form} \left[ \widehat{\text{doc}} \right]$

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

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

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

$\left\{ \begin{array}{l} m\text{setf} \\ m\text{psetf} \end{array} \right\} \left\{ \text{place } \text{form}^* \right\}$

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

$\left\{ \begin{array}{l} s\text{setq} \\ m\text{psetq} \end{array} \right\} \left\{ \text{symbol } \text{form}^* \right\}$

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

(*f*set  $\widetilde{\text{symbol}}$  *foo*) ▷ Set *symbol*'s value cell to foo. Deprecated.

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

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

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

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

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

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

(*f*makunbound  $\widetilde{\text{foo}}$ ) ▷ Delete special variable foo if any.

(*f*get *symbol* *key* [*default*<sub>NIL</sub>])

(*f*getf *place* *key* [*default*<sub>NIL</sub>])

▷ 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  $\widetilde{\text{symbol}}$  *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*<sup>P<sub>k</sub></sup>)

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

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

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

$(_m\text{destructuring-bind } \text{destruct-}\lambda \text{ bar } (\text{declare } \widehat{\text{decl}}^*)^* \text{ form}^{\text{P}_k})$   
 ▷ 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

$(\text{var}^* \text{ \&optional } \{ \{ \text{var } \text{ [init}_{\text{NIL}} \text{ [supplied-p]]} \} \}^* \text{ \&rest var}$   
 $\text{ \&key } \{ \{ \{ \text{var } \text{ [init}_{\text{NIL}} \text{ [supplied-p]]} \} \}^* \text{ \&allow-other-keys}$   
 $\text{ \&aux } \{ \text{var } \text{ [init}_{\text{NIL}} \text{ [supplied-p]]} \}^* \text{ ]}$

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

$(\{ _m\text{defun } \{ \text{foo } (\text{ord-}\lambda^*) \} \{ \{ (\text{declare } \widehat{\text{decl}}^*)^* \} \} \{ _m\text{lambda } (\text{ord-}\lambda^*) \text{ form}^{\text{P}_k} \} \}$   
 ▷ Define a function named *foo* or (**setf** *foo*), or an anonymous function, respectively, which applies *forms* to *ord-λs*. For *mdefun*, *forms* are enclosed in an implicit **block** named *foo*.

$(\{ _s\text{flet } \{ \{ \{ \text{foo } (\text{ord-}\lambda^*) \} \} \{ \{ (\text{declare } \widehat{\text{local-decl}}^*)^* \} \} \{ _s\text{labels } \{ \{ \{ \text{foo } (\text{ord-}\lambda^*) \} \} \{ \{ (\text{declare } \widehat{\text{local-decl}}^*)^* \} \} \} \} \}$   
 $\text{ local-form}^{\text{P}_k})^* \text{ (declare } \widehat{\text{decl}}^*)^* \text{ form}^{\text{P}_k}$   
 ▷ 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 **block** around its corresponding *local-form\**. Only for **slabels**, functions *foo* are visible inside *local-forms*. Return values of forms.

$(_s\text{function } \{ \text{foo } \{ _m\text{lambda } \text{ form}^* \} \}$   
 ▷ Return lexically innermost function named *foo* or a lexical closure of the **mlambda** expression.

$(_f\text{apply } \{ \text{function } \{ \text{setf } \text{function} \} \} \text{ arg}^* \text{ args}$   
 ▷ Values of *function* called with *args* and the list elements of *args*. **setfable** if *function* is one of **faref**, **fbit**, and **fsbit**.

$(_f\text{funcall } \text{function } \text{arg}^*)$  ▷ Values of *function* called with *args*.

$(_s\text{multiple-value-call } \text{function } \text{form}^*)$   
 ▷ Call *function* with all the values of each *form* as its arguments. Return values returned by function.

$(_f\text{values-list } \text{list})$  ▷ Return elements of list.

$(_f\text{values } \text{foo}^*)$   
 ▷ Return as multiple values the primary values of the *foos*. **setfable**.

$(_f\text{multiple-value-list } \text{form})$  ▷ List of the values of form.

$(_m\text{nth-value } n \text{ form})$   
 ▷ Zero-indexed *n*th return value of *form*.

$(_f\text{complement } \text{function})$   
 ▷ Return new function with same arguments and same side effects as *function*, but with complementary truth value.

$(_f\text{constantly } \text{foo})$   
 ▷ Function of any number of arguments returning *foo*.

$(_f\text{identity } \text{foo})$  ▷ Return *foo*.

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

$(_f\text{fdefinition } \{ \text{foo } \{ \text{setf } \text{foo} \} \})$   
 ▷ Definition of global function *foo*. **setfable**.

$(_f\text{fmakunbound } \text{foo})$   
 ▷ 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

$(\text{ \&whole var } [E] \{ \text{var } \{ (\text{macro-}\lambda^*) \} \}^* [E]$   
 $\text{ \&optional } \{ \{ \{ \text{var } \{ (\text{macro-}\lambda^*) \} \} \text{ [init}_{\text{NIL}} \text{ [supplied-p]]} \} \}^* [E]$   
 $\text{ \&rest } \{ \text{rest-var } \{ (\text{macro-}\lambda^*) \} \} [E]$   
 $\text{ \&body } \{ \{ \text{var } \{ (\text{macro-}\lambda^*) \} \} \text{ [init}_{\text{NIL}} \text{ [supplied-p]]} \}^* [E]$   
 $\text{ \&key } \{ \{ \{ \text{var } \{ (\text{macro-}\lambda^*) \} \} \} \text{ [init}_{\text{NIL}} \text{ [supplied-p]]} \}^* [E]$   
 $\text{ \&allow-other-keys } \text{ \&aux } \{ \text{var } \text{ [init}_{\text{NIL}} \text{ [supplied-p]]} \}^* [E]$   
 or  
 $(\text{ \&whole var } [E] \{ \text{var } \{ (\text{macro-}\lambda^*) \} \}^* [E]$   
 $\text{ \&optional } \{ \{ \{ \text{var } \{ (\text{macro-}\lambda^*) \} \} \text{ [init}_{\text{NIL}} \text{ [supplied-p]]} \} \}^* [E] . \text{rest-var}$

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.

$(\{ _m\text{defmacro } \{ \text{foo } \{ \text{setf } \text{foo} \} \} \{ _f\text{define-compiler-macro } \{ \{ (\text{declare } \widehat{\text{decl}}^*)^* \} \} \text{ form}^{\text{P}_k} \} \}$   
 ▷ 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 **block** named *foo*.

**{upfrom|from|downfrom}** *start*  
 ▷ Start stepping with *start*

**{upto|downto|to|below|above}** *form*  
 ▷ Specify *form* as the end value for stepping.

**{in|on}** *list*  
 ▷ Bind *var* to successive elements/tails, respectively, of *list*.

**by**  $\{step_{\square} | function_{\#cdn}\}$   
 ▷ Specify the (positive) decrement or increment or the *function* of one argument returning the next part of the list.

**=** *foo* **[then** *bar*<sub>*foo*</sub>**]**  
 ▷ Bind *var* initially to *foo* and later to *bar*.

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

**being** **{the|each}**  
 ▷ Iterate over a hash table or a package.

**{hash-key|hash-keys}** **{of|in}** *hash-table* **[using** *(hash-value value)*  
 ▷ Bind *var* successively to the keys of *hash-table*; bind *value* to corresponding values.

**{hash-value|hash-values}** **{of|in}** *hash-table* **[using** *(hash-key key)*  
 ▷ Bind *var* successively to the values of *hash-table*; bind *key* to corresponding keys.

**{symbol|symbols|present-symbol|present-symbols|external-symbol|external-symbols}** **{of|in}** *package*<sub>*\*package\**</sub>  
 ▷ Bind *var* successively to the accessible symbols, or the present symbols, or the external symbols respectively, of *package*.

**{do|doing}** *form*<sup>+</sup> ▷ Evaluate *forms* in every iteration.

**{if|when|unless}** *test* *i-clause* **{and** *j-clause*<sup>\*</sup> **[else** *k-clause* **{and** *l-clause*<sup>\*</sup> **]** **[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 *fappend* or *fnconc*, respectively. If no *list* is given, collect into an anonymous list which is returned after termination.

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

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

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

**&allow-other-keys**

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

**&environment** *var*

▷ Bind *var* to the lexical compilation environment.

**&aux** *var*<sup>\*</sup> ▷ Bind *vars* as in **let**<sup>\*</sup>.

## 9.5 Control Flow

**(*s*if** *test* **then** *[else*<sub>*NIL*</sub>**])**

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

**(*m*cond** (*test* *then*<sup>*P*</sup><sub>*TEST*</sub>)<sup>\*</sup>)

▷ Return the values of the first *then*<sup>\*</sup> whose *test* returns T; return *NIL* if all *tests* return *NIL*.

**(*m*when** *test* *foo*<sup>*P*</sup>)  
**(*m*unless** *test* *foo*<sup>*P*</sup>)

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

**(*m*case** *test*  $\left\{ \left\{ \widehat{key}^* \right\} \right\}$  *foo*<sup>*P*</sup>)<sup>\*</sup> **[**(**{otherwise}** *bar*<sup>*P*</sup>)<sub>*NIL*</sub>**]**)

▷ Return the values of the first *foo*<sup>\*</sup> one of whose *keys* is **eq** *test*. Return values of *bars* if there is no matching *key*.

**(*m*ecase** *test*  $\left\{ \left\{ \widehat{key}^* \right\} \right\}$  *foo*<sup>*P*</sup>)<sup>\*</sup>  
**(*m*ccase** *test*  $\left\{ \left\{ \widehat{key}^* \right\} \right\}$  *foo*<sup>*P*</sup>)<sup>\*</sup>)

▷ Return the values of the first *foo*<sup>\*</sup> one of whose *keys* is **eq** *test*. Signal non-correctable/correctable **type-error** if there is no matching *key*.

**(*m*and** *form*<sup>\*</sup> *n*)

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

**(*m*or** *form*<sup>\*</sup> *n*<sub>*NIL*</sub>)

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

**(*s*progn** *form*<sup>\*</sup> *n*<sub>*NIL*</sub>)

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

**(*s*multiple-value-prog1** *form-r* *form*<sup>\*</sup>)**(*m*prog1** *form-r* *form*<sup>\*</sup>)**(*m*prog2** *form-a* *form-r* *form*<sup>\*</sup>)

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

**(*m*prog**  $\left\{ \left\{ name \right\} \right\}$   $\left\{ \left\{ (name [value_{\substack{NIL}}]) \right\} \right\}$  <sup>\*</sup>) **(declare**  $\widehat{decl}$ <sup>\*</sup>)<sup>\*</sup>  $\left\{ \widehat{tag} \right\}$  <sup>\*</sup>  
**(*m*prog\***  $\left\{ \left\{ (name [value_{\substack{NIL}}]) \right\} \right\}$  <sup>\*</sup>) **(declare**  $\widehat{decl}$ <sup>\*</sup>)<sup>\*</sup>  $\left\{ \widehat{tag} \right\}$  <sup>\*</sup>)

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

**(*s*unwind-protect** *protected* *cleanup*<sup>\*</sup>)

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

**(*s*block** *name* *form*<sup>*P*</sup>)

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

**(*s*return-from** *foo* *[result*<sub>*NIL*</sub>**])****(*m*return** *[result*<sub>*NIL*</sub>**])**

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

(*s*tagbody {*tag*|*form*}\*)  
 ▷ Evaluate *forms* in a lexical environment. *tags* (symbols or integers) have lexical scope and dynamic extent, and are targets for *s*go. Return NIL.

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

(*s*catch *tag form*<sup>\*</sup>)  
 ▷ 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.

### 9.6 Iteration

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

(*m*dotimes (*var i* [*result*<sub>NIL</sub>]) (declare *decl*<sup>\*</sup>) {*tag*|*form*}\*)  
 ▷ Evaluate *s*tagbody-like body with *var* successively bound to integers from 0 to *i* - 1. Upon evaluation of *result*, *var* is *i*. Implicitly, the whole form is a *s*block named NIL.

(*m*dolist (*var list* [*result*<sub>NIL</sub>]) (declare *decl*<sup>\*</sup>) {*tag*|*form*}\*)  
 ▷ Evaluate *s*tagbody-like body with *var* successively bound to the elements of *list*. Upon evaluation of *result*, *var* is NIL. Implicitly, the whole form is a *s*block named NIL.

### 9.7 Loop Facility

(*m*loop *form*<sup>\*</sup>)  
 ▷ **Simple Loop.** If *forms* do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit *s*block named NIL.

(*m*loop *clause*<sup>\*</sup>)  
 ▷ **Loop Facility.** For Loop Facility keywords see below and Figure 1.

named *n*<sub>NIL</sub> ▷ Give *m*loop's implicit *s*block a name.

{with {*var-s* | (*var-s*<sup>\*</sup>)} [*d-type*] [= *foo*]}<sup>+</sup>  
 {and {*var-p* | (*var-p*<sup>\*</sup>)} [*d-type*] [= *bar*]}<sup>\*</sup>

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

{fixnum|float|T|NIL}{of-type {*type* | (*type*<sup>\*</sup>)}}

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

{{for|as} {*var-s* | (*var-s*<sup>\*</sup>)} [*d-type*]}<sup>+</sup> {and {*var-p* | (*var-p*<sup>\*</sup>)} [*d-type*]}<sup>\*</sup>

▷ 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*.

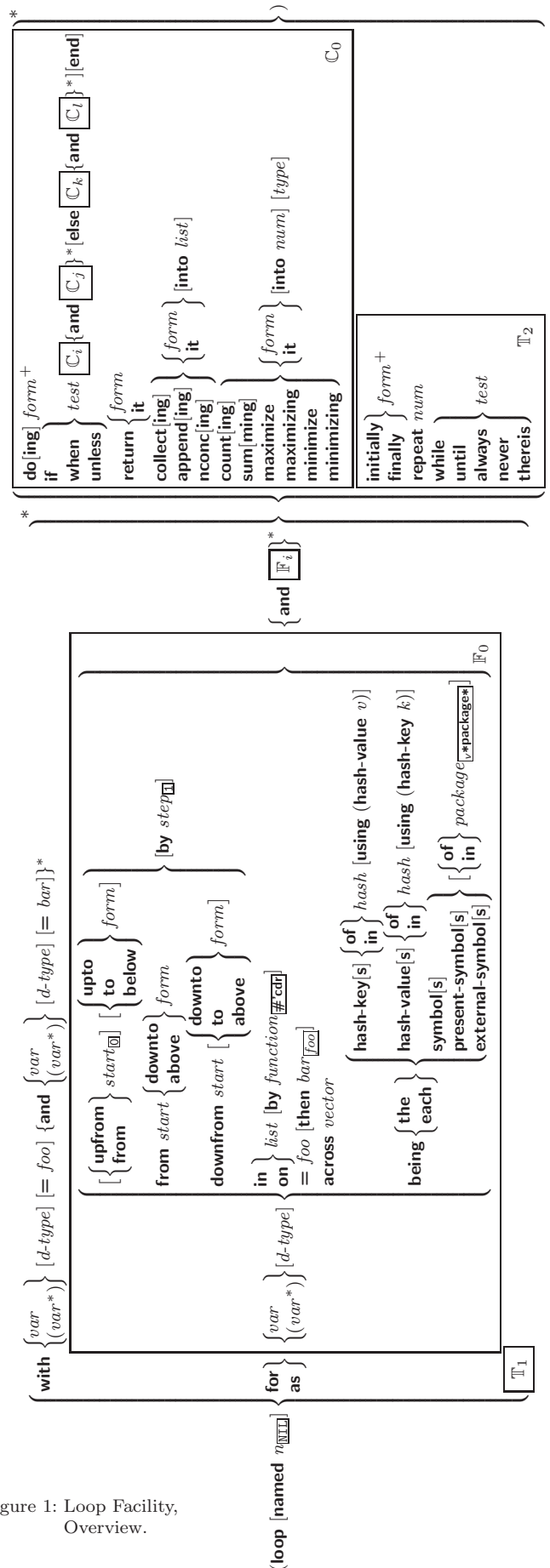


Figure 1: Loop Facility, Overview.

(*g*function-keywords *method*)

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

(*g*method-qualifiers *method*)

▷ List of qualifiers of *method*.

## 10.3 Method Combination Types

standard

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

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

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

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

{  
  :**documentation** *string*  
  :**identity-with-one-argument** *bool*<sub>NIL</sub>  
  :**operator** *operator*<sub>*c-type*</sub>  
})

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

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

{  
  \*  
  (*qualifier*\* [*\**])  
  *predicate*  
  :**description** *control*  
  :**order** {**:most-specific-first** | **:most-specific-last** | **:most-specific-first**}\*  
  :**required** *bool*  
  {  
    (**:arguments** *method-combination-λ*\*)  
    (**:generic-function** *symbol*)  
    {  
      (**declare** *decl*)\*  
      *doc*  
    } *body*<sub>R</sub>  
  }  
})

▷ **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 left-most *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 18, the latter enhanced by an optional **&whole** argument.

(*m*call-method

{  
  *method*  
  (**make-method** *form*)  
} [(**next-method** *form*)  
(**make-method** *form*)]\*)

{**initially**|**finally**} *form*<sup>+</sup>

▷ 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.

(*m*loop-finish)

▷ Terminate **mloop** immediately executing any **finally** clauses and returning any accumulated results.

## 10 CLOS

### 10.1 Classes

(*f*slot-exists-p *foo bar*)

▷  $\underline{T}$  if *foo* has a slot *bar*.

(*f*slot-boundp *instance slot*)

▷  $\underline{T}$  if *slot* in *instance* is bound.

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

{  
  *slot*  
  {  
    :**reader** *reader*\*  
    :**writer** {*writer* | (**setf** *writer*)}\*  
    :**accessor** *accessor*\*  
    :**allocation** {**:instance** | **:class** | **:instance**}\*  
    :**initarg** [:*initarg-name*]\*  
    :**initform** *form*  
    :**type** *type*  
    :**documentation** *slot-doc*  
  }  
  {  
    (**:default-initargs** {*name value*}\*)  
    (**:documentation** *class-doc*)  
    (**:metaclass** *name*<sub>standard-class</sub>)  
  }  
}

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

(*f*find-class *symbol* [*errorp*<sub>nil</sub>] [*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 **gshared-initialize**.

(*f*slot-value *foo slot*)

▷ Return value of *slot* in *foo*. **setfable**.

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

▷ Make *slot* in instance unbound.

$\left\{ \begin{array}{l} \text{mwith-slots } (\widehat{\text{slot}} (\widehat{\text{var slot}})^*) \\ \text{mwith-accessors } (\widehat{\text{var accessor}})^* \end{array} \right\}$  *instance* (**declare**  $\widehat{\text{decl}}^*$ )<sup>+</sup> *form*<sup>P</sup>\*)  
 ▷ 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**.

$(\text{gclass-name } \textit{class})$   
 $(\text{setf } \text{gclass-name}) \textit{new-name } \textit{class}$     ▷ Get/set name of class.

$(\text{fclass-of } \textit{foo})$     ▷ Class *foo* is a direct instance of.

$(\text{gchange-class } \widehat{\text{instance } \textit{new-class}} \{[:]\textit{initarg } \textit{value}}^* \textit{other-keyarg}^*)$   
 ▷ 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.

$(\text{gmake-instances-obsolete } \textit{class})$   
 ▷ Update all existing instances of *class* using  $\text{gupdate-instance-for-redefined-class}$ .

$\left\{ \begin{array}{l} \text{ginitialize-instance } \textit{instance} \\ \text{gupdate-instance-for-different-class } \textit{previous } \textit{current} \end{array} \right\}$   
 $\{[:]\textit{initarg } \textit{value}}^* \textit{other-keyarg}^*)$   
 ▷ Set slots on behalf of  $\text{gmake-instance}$ /of  $\text{gchange-class}$  by means of  $\text{gshared-initialize}$ .

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

$(\text{gallocate-instance } \textit{class} \{[:]\textit{initarg } \textit{value}}^* \textit{other-keyarg}^*)$   
 ▷ Return uninitialized instance of *class*. Called by  $\text{gmake-instance}$ .

$(\text{gshared-initialize } \textit{instance} \left\{ \begin{array}{l} \textit{initform-slots} \\ \text{T} \end{array} \right\} \{[:]\textit{initarg-slot } \textit{value}}^*$   
 $\textit{other-keyarg}^*)$   
 ▷ 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.

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

$(\text{gslot-unbound } \textit{class } \textit{instance } \textit{slot})$   
 ▷ 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

$(\text{fnext-method-p})$     ▷ T if enclosing method has a next method.

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

▷ 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 `mdefmethod`. For *c-type* see section 10.3.

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

▷ Define or modify generic function *foo*. *gf-class* and *lambda-list* must be compatible with a pre-existing generic function or with existing methods, respectively. Changes to *method-class* do not propagate to existing methods. For *c-type* see section 10.3.

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

▷ 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

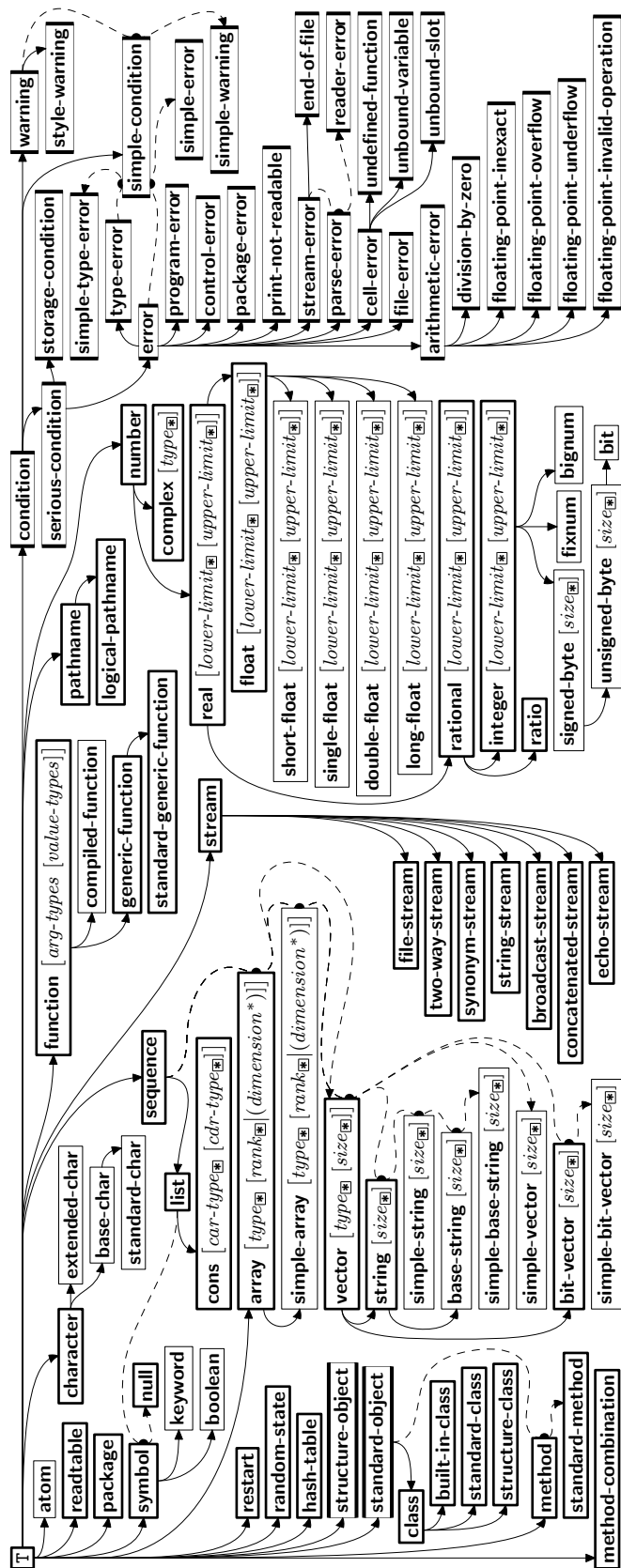


Figure 2: Precedence Order of System Classes (□), Classes (▢), Types (▣), and Condition Types (▤). Every type is also a supertype of NIL, the empty type.

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

## 11 Conditions and Errors

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

```
(mdefine-condition foo (parent-type* condition)
  (slot
    ([:reader reader]*
     [:writer {writer (self writer)}]*
     [:accessor accessor]*
     [:allocation {instance :class :instance}]
     [:initarg [:initarg-name]*]
     :initform form
     :type type
     :documentation slot-doc
     (:default-initargs {name value}*)
     (:documentation condition-doc)
     (:report {string report-function})))
```

▷ Define, as a subtype of *parent-types*, condition type *foo*. In a new condition, a *slot*'s value defaults to *form* unless set via *[:initarg-name]*; it is readable via (*reader i*) or (*accessor i*), and writable via (*writer value i*) or (*self (accessor i) value*). With *:allocation :class*, *slot* is shared by all conditions of type *foo*. A condition is reported by *string* or by *report-function* of arguments *condition* and *stream*.

```
(fmake-condition condition-type {[:initarg-name value]}*)
  ▷ Return new instance of condition-type.
```

```
{fsignal} {condition
           condition-type {[:initarg-name value]}*}
{fwarn}   {control arg*}
           ▷ Unless handled, signal as condition, warning or error, respectively, condition or a new instance of condition-type or, with fformat control and args (see page 38), simple-condition, simple-warning, or simple-error, respectively. From fsignal and fwarn, return NIL.
```

```
(ferror continue-control {condition continue-arg*
                          condition-type {[:initarg-name value]}*}
                          control arg*)
  ▷ Unless handled, signal as correctable error condition or a new instance of condition-type or, with fformat control and args (see page 38), simple-error. In the debugger, use fformat arguments continue-control and continue-args to tag the continue option. Return NIL.
```

```
(mignore-errors formP*)
  ▷ Return values of forms or, in case of errors, NIL and the condition.
```

```
(finvoke-debugger condition)
  ▷ Invoke debugger with condition.
```

```
(massert test [(place*) {condition continue-arg*
                          condition-type {[:initarg-name value]}*}
               control arg*])
  ▷ If test, which may depend on places, returns NIL, signal as correctable error condition or a new instance of condition-type or, with fformat control and args (see page 38), error. When using the debugger's continue option, places can be altered before re-evaluation of test. Return NIL.
```

(*m*handler-case *foo* (*type* ([*var*]) (declare  $\widehat{\text{decl}}^*$ )<sup>P</sup>\* *condition-form*<sup>P</sup>\*)  
 [(:no-error (*ord-λ*\*) (declare  $\widehat{\text{decl}}^*$ )<sup>P</sup>\* *form*<sup>P</sup>\*)])  
 ▷ If, on evaluation of *foo*, a condition of *type* is signalled, evaluate matching *condition-forms* with *var* bound to the condition, and return their values. Without a condition, bind *ord-λs* to values of *foo* and return values of *forms* or, without a :no-error clause, return values of *foo*. See page 18 for (*ord-λ*\*)<sup>P</sup>.

(*m*handler-bind ((*condition-type* *handler-function*)<sup>\*</sup>) *form*<sup>P</sup>\*)  
 ▷ 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*<sup>P</sup>\*)  
 ▷ Return values of *forms* unless *restart* is called during their evaluation. In this case, describe *restart* using *f*format *control* and *args* (see page 38) and return NIL and T.<sub>2</sub>

(*m*restart-case *form* (*restart* (*ord-λ*\*)<sup>\*</sup>)  $\left\{ \begin{array}{l} \text{:interactive } \textit{arg-function} \\ \text{:report } \left\{ \begin{array}{l} \textit{report-function} \\ \textit{string} \textit{restart} \end{array} \right\} \\ \text{:test } \textit{test-function} \end{array} \right\}$   
 (declare  $\widehat{\text{decl}}^*$ )<sup>\*</sup> *restart-form*<sup>P</sup>\*)  
 ▷ 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*\*)<sup>\*</sup>, where *args* match *ord-λ*\*, or by (*invoke-restart-interactively* *restart*) where a list of the respective *args* is supplied by #'*arg-function*. See page 18 for *ord-λ*.\*

(*m*restart-bind ( $\left\{ \begin{array}{l} \text{restart} \\ \text{NIL} \end{array} \right\}$  *restart-function*  
 $\left\{ \begin{array}{l} \text{:interactive-function } \textit{arg-function} \\ \text{:report-function } \textit{report-function} \\ \text{:test-function } \textit{test-function} \end{array} \right\}$ \*)<sup>\*</sup>) *form*<sup>P</sup>\*)  
 ▷ 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*\*)<sup>\*</sup>, where *args* must be suitable for the corresponding *restart-function*, or by (*invoke-restart-interactively* *restart*) where a list of the respective *args* is supplied by *arg-function*.

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

( $\left\{ \begin{array}{l} \text{f} \textit{find-restart} \\ \text{f} \textit{compute-restarts} \end{array} \right\}$  *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*.

$\left\{ \begin{array}{l} \text{f} \textit{abort} \\ \text{f} \textit{muffle-warning} \\ \text{f} \textit{continue} \\ \text{f} \textit{store-value } \textit{value} \\ \text{f} \textit{use-value } \textit{value} \end{array} \right\}$  [*condition*<sub>NTL</sub>]

▷ 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*<sup>P</sup>\*)  
 ▷ 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\*<sub>NTL</sub>  
 ▷ Condition type debugger is to be invoked on.

\*debugger-hook\*<sub>NTL</sub>  
 ▷ Function of condition and function itself. Called before debugger.

## 12 Types and Classes

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

(*f*typep *foo* *type* [*environment*<sub>NTL</sub>]) ▷ T if *foo* is of *type*.

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

(*s*the  $\widehat{\text{type}}$  *form*) ▷ Declare values of *form* to be of *type*.

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

(*m*typecase *foo* ( $\widehat{\text{type}}$  *a-form*<sup>P</sup>\*)<sup>\*</sup> [( $\left\{ \begin{array}{l} \text{otherwise} \\ \text{T} \end{array} \right\}$  *b-form*<sub>NTL</sub><sup>P</sup>\*)])  
 ▷ Return values of the first *a-form*\* whose *type* is *foo* of. Return values of *b-forms* if no *type* matches.

( $\left\{ \begin{array}{l} \text{m} \textit{etypecase} \\ \text{m} \textit{ctypecase} \end{array} \right\}$  *foo* ( $\widehat{\text{type}}$  *form*<sup>P</sup>\*)<sup>\*</sup>)  
 ▷ Return values of the first *form*\* whose *type* is *foo* of. Signal non-correctable/correctable **type-error** if no *type* matches.



- # $[n]*b^*$**       ▷ Bit vector of some (or  $n$ )  $bs$  filled with last  $b$  if necessary.
- #S(*type* {*slot value*}\*)**      ▷ Structure of *type*.
- #P*string***      ▷ A pathname.
- #:*foo***      ▷ Uninterned symbol *foo*.
- #.*form***      ▷ Read-time value of *form*.
- $\nu$ \*read-eval\* $\square$**       ▷ If NIL, a **reader-error** is signalled at **#.**
- #*integer*= *foo***      ▷ Give *foo* the label *integer*.
- #*integer*#**      ▷ Object labelled *integer*.
- #<**      ▷ Have the reader signal **reader-error**.
- #+*feature when-feature***  
**#-*feature unless-feature***  
 ▷ Means *when-feature* if *feature* is T; means *unless-feature* if *feature* is NIL. *feature* is a symbol from  $\nu$ \***features\***, or (**{and|or}** *feature\**), or (**not** *feature*).
- $\nu$ \***features\*****  
 ▷ List of symbols denoting implementation-dependent features.
- |*c\**|; \c**  
 ▷ Treat arbitrary character(s) *c* as alphabetic preserving case.

## 13.4 Printer

- ( $\left\{ \begin{array}{l} \text{fprn1} \\ \text{fprint} \\ \text{fpprint} \\ \text{fprinc} \end{array} \right\}$  *foo* [*stream*  $\widetilde{\nu$ \*standard-output\*}]**  
 ▷ Print *foo* to *stream* **freadably**, **freadably** between a newline and a space, **freadably** after a newline, or **human-readably** without any extra characters, respectively. **fprn1**, **fprint** and **fpprint** return *foo*.
- (*fprn1-to-string* *foo*)**  
**(*fprinc-to-string* *foo*)**  
 ▷ Print *foo* to *string* **freadably** or **human-readably**, respectively.
- (*gprint-object* *object* *stream*)**  
 ▷ Print *object* to *stream*. Called by the Lisp printer.
- (*mprint-unreadable-object* (*foo* *stream*  $\left\{ \begin{array}{l} \text{:type } \text{bool}_{\text{NIL}} \\ \text{:identity } \text{bool}_{\text{NIL}} \end{array} \right\}$ ) *form*<sup>P\*</sup>)**  
 ▷ Enclosed in **#<** and **>**, print *foo* by means of *forms* to *stream*. Return **NIL**.
- (*fterpri* [*stream*  $\widetilde{\nu$ \*standard-output\*}])**  
 ▷ Output a newline to *stream*. Return **NIL**.
- (*f***fresh-line** [*stream*  $\widetilde{\nu$ \*standard-output\*}])**  
 ▷ Output a newline to *stream* and return **T** unless *stream* is already at the start of a line.
- (*f***write-char** *char* [*stream*  $\widetilde{\nu$ \*standard-output\*}])**  
 ▷ Output *char* to *stream*.
- ( $\left\{ \begin{array}{l} \text{fwrite-string} \\ \text{fwrite-line} \end{array} \right\}$  *string* [*stream*  $\widetilde{\nu$ \*standard-output\*}** [**{:start** *start* <sub>$\square$</sub>  **end** *end* <sub>$\text{NIL}$</sub>  **}]])**  
 ▷ Write *string* to *stream* without/with a trailing newline.
- (*f***write-byte** *byte* *stream*)**      ▷ Write *byte* to binary *stream*.

- (*f***type-of** *foo*)**      ▷ **Type** of *foo*.
- (*m***check-type** *place* *type* [*string*  $\widetilde{\text{[a|an] type}}$ ])**  
 ▷ Signal correctable **type-error** if *place* is not of *type*. Return **NIL**.
- (*f***stream-element-type** *stream*)**      ▷ **Type** of *stream* objects.
- (*f***array-element-type** *array*)**      ▷ Element **type** *array* can hold.
- (*f***upgraded-array-element-type** *type* [*environment*  $\widetilde{\text{NIL}}$ ])**  
 ▷ **Element type** of most specialized array capable of holding elements of *type*.
- (*m***deftype** *foo* (*macro- $\lambda$ \**)  $\left\{ \begin{array}{l} \text{(declare } \widehat{\text{decl}}^* \text{)} \\ \text{doc} \end{array} \right\}$  *form*<sup>P\*</sup>)**  
 ▷ Define type *foo* which when referenced as (*foo*  $\widehat{\text{arg}}^*$ ) (or as *foo* if *macro- $\lambda$*  doesn't contain any required parameters) applies expanded *forms* to *args* returning the new type. For (*macro- $\lambda$ \**) see page 19 but with default value of **\*** instead of NIL. *forms* are enclosed in an implicit **block** named *foo*.
- (*eql* *foo*)**  
**(*member* *foo\**)**      ▷ Specifier for a type comprising *foo* or *foos*.
- (*satisfies* *predicate*)**  
 ▷ Type specifier for all objects satisfying *predicate*.
- (*mod* *n*)**      ▷ Type specifier for all non-negative integers  $< n$ .
- (*not* *type*)**      ▷ Complement of type.
- (*and* *type\** <sub>$\square$</sub> )**      ▷ Type specifier for intersection of *types*.
- (*or* *type\** <sub>$\text{NIL}$</sub> )**      ▷ Type specifier for union of *types*.
- (*values* *type\** [**&optional** *type\** [**&rest** *other-args*]])**  
 ▷ Type specifier for multiple values.
- \***      ▷ As a type argument (cf. Figure 2): no restriction.

## 13 Input/Output

### 13.1 Predicates

- (*f***stream-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

$\left\{ \begin{array}{l} \text{f y-or-n-p} \\ \text{f yes-or-no-p} \end{array} \right\}$  [*control arg\**])

▷ Ask user a question and return T or NIL depending on their answer. See page 38, **fformat**, for *control* and *args*.

(*m with-standard-io-syntax form\**)

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

$\left\{ \begin{array}{l} \text{f read} \\ \text{f read-preserving-whitespace} \end{array} \right\}$  [*stream* v\*standard-input\* [*eof-err* T [*eof-val* NIL] [*recursive* NIL]]]

▷ Read printed representation of object.

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

$\left\{ \begin{array}{l} \text{:start } \textit{start}_{\text{Q}} \\ \text{:end } \textit{end}_{\text{NIL}} \\ \text{:preserve-whitespace } \textit{bool}_{\text{NIL}} \end{array} \right\}$ )]])

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

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

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

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

▷ Return next character from *stream*.

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

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

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

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

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

▷ Put last **fread-char** *character* back into *stream*; return NIL.

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

▷ Read next byte from binary *stream*.

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

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

(*f read-sequence* *sequence* [*stream* v\*standard-input\*] [*:start* *start* Q] [*: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 readable-case* *readtable*) upcase

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

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

▷ Return copy of from-readtable.

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

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

**v\*readtable\*** ▷ Current readtable.

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

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

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

**v\*read-suppress\*** NIL ▷ If T, reader is syntactically more tolerant.

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

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

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

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

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

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

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

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

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

▷ Dispatch function associated with *char* followed by *sub-char*.

## 13.3 Character Syntax

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

**;** *one-line-comment\**

▷ Comments. There are stylistic conventions:

**;;;** *title* ▷ Short title for a block of code.

**;;** *intro* ▷ Description before a block of code.

**;;** *state* ▷ State of program or of following code.

**;** *explanation*

**;** *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** ▷ (**quote** *foo*); *foo* unevaluated.

**`([foo] [*bar*] [*@baz*] [*.,quux*] [*bing*])**

▷ Backquote. **quote** *foo* and *bing*; evaluate *bar* and splice the lists *baz* and *quux* into their elements. When nested, outermost commas inside the innermost backquote expression belong to this backquote.

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

**#Bn**; **#On**; *n*; **#Xn**; **#rRn**

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

*n/d* ▷ The **ratio**  $\frac{n}{d}$ .

$\left\{ [m].n \left[ \left\{ \text{S|F|D|L|E} \right\} x_{\text{E}} \right] m \left[ .[n] \left\{ \text{S|F|D|L|E} \right\} x \right] \right\}$

▷  $m.n \cdot 10^x$  as **short-float**, **single-float**, **double-float**, **long-float**, or the type from **\*read-default-float-format\***.

**#C(a b)** ▷ (**fcomplex** *a b*), the complex number  $a + bi$ .

**#'foo** ▷ (**function** *foo*); the function named *foo*.

**#nAsequence** ▷ *n*-dimensional array.

**#[n](foo\*)**

▷ Vector of some (or *n*) *foos* filled with last *foo* if necessary.

$\sim$   $[:]$   $[\mathbf{Q}] < \{ [prefix_{\mathbf{mm}} \sim;] [per-line-prefix \sim \mathbf{Q};] \}$  *body*  $[-;]$   
 $suffix_{\mathbf{mm}}$   $\sim; [\mathbf{Q}] >$   
 ▷ **Logical Block.** Act like `pprint-logical-block` using *body* as  $f$ format control string on the elements of the list argument or, with  $\mathbf{Q}$ , on the remaining arguments, which are extracted by `pprint-pop`. With  $;$ , *prefix* and *suffix* default to ( and ). When closed by  $\sim \mathbf{Q}; >$ , spaces in *body* are replaced with conditional newlines.

$\{ \sim [n_{\mathbf{Q}}] i | \sim [n_{\mathbf{Q}}] : i \}$   
 ▷ **Indent.** Set indentation to *n* relative to leftmost/to current position.

$\sim [c_{\mathbf{Q}}] [,i_{\mathbf{Q}}] [:] [\mathbf{Q}] \mathbf{T}$   
 ▷ **Tabulate.** Move cursor forward to column number  $c + ki$ ,  $k \geq 0$  being as small as possible. With  $;$ , calculate column numbers relative to the immediately enclosing section. With  $\mathbf{Q}$ , move to column number  $c_0 + c + ki$  where  $c_0$  is the current position.

$\{ \sim [m_{\mathbf{Q}}] * | \sim [m_{\mathbf{Q}}] : * | \sim [n_{\mathbf{Q}}] \mathbf{Q} * \}$   
 ▷ **Go-To.** Jump *m* arguments forward, or backward, or to argument *n*.

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

$\sim [x [,y [,z]]] ^$   
 ▷ **Escape Upward.** Leave immediately  $\sim < \sim >$ ,  $\sim < \sim : >$ ,  $\sim \{ \sim \}$ ,  $\sim ?$ , or the entire  $f$ format operation. With one to three prefixes, act only if  $x = 0$ ,  $x = y$ , or  $x \leq y \leq z$ , respectively.

$\sim [i] [:] [\mathbf{Q}] [ \{ [text \sim;] * text [ \sim; default ] \sim } ]$   
 ▷ **Conditional Expression.** Use the zero-indexed argument (or *i*th if given) *text* as a  $f$ format control subclause. With  $;$ , use the first *text* if the argument value is NIL, or the second *text* if it is T. With  $\mathbf{Q}$ , do nothing for an argument value of NIL. Use the only *text* and leave the argument to be read again if it is T.

$\{ \sim ? | \sim \mathbf{Q} ? \}$   
 ▷ **Recursive Processing.** Process two arguments as control string and argument list, or take one argument as control string and use then the rest of the original arguments.

$\sim [prefix \{ ,prefix \} *] [:] [\mathbf{Q}] / [package [:] : [cl-user]] function /$   
 ▷ **Call Function.** Call all-uppercase *package::function* with the arguments *stream*, *format-argument*, *colon-p*, *at-sign-p* and *prefixes* for printing *format-argument*.

$\sim [:] [\mathbf{Q}] \mathbf{W}$   
 ▷ **Write.** Print argument of any type obeying every printer control variable. With  $;$ , pretty-print. With  $\mathbf{Q}$ , print without limits on length or depth.

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

$(fwrite-sequence \widetilde{sequence} \widetilde{stream} \left\{ \begin{array}{l} :start \text{ start}_{\mathbf{mm}} \\ :end \text{ end}_{\mathbf{mm}} \end{array} \right\})$   
 ▷ Write elements of *sequence* to binary or character *stream*.

$(\left\{ \begin{array}{l} fwrite \\ fwrite-to-string \end{array} \right\} \text{foo} \left\{ \begin{array}{l} :array \text{ bool} \\ :base \text{ radix} \\ :case \left\{ \begin{array}{l} :upcase \\ :downcase \\ :capitalize \end{array} \right\} \\ :circle \text{ bool} \\ :escape \text{ bool} \\ :gensym \text{ bool} \\ :length \{ int | NIL \} \\ :level \{ int | NIL \} \\ :lines \{ int | NIL \} \\ :miser-width \{ int | NIL \} \\ :pprint-dispatch \text{ dispatch-table} \\ :pretty \text{ bool} \\ :radix \text{ bool} \\ :readably \text{ bool} \\ :right-margin \{ int | NIL \} \\ :stream \text{ stream}_{\mathbf{v} \text{ standard-output} *} \end{array} \right\})$

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

$(fpprint-fill \widetilde{stream} \text{foo} [parenthesis_{\mathbf{mm}} [noop]])$

$(fpprint-tabular \widetilde{stream} \text{foo} [parenthesis_{\mathbf{mm}} [noop] [n_{\mathbf{mm}}]])$

$(fpprint-linear \widetilde{stream} \text{foo} [parenthesis_{\mathbf{mm}} [noop]])$

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

$(mpprint-logical-block (\widetilde{stream} \text{list} \left\{ \left\{ \begin{array}{l} :prefix \text{ string} \\ :per-line-prefix \text{ string} \end{array} \right\} \right\} \left\{ \begin{array}{l} :suffix \text{ string}_{\mathbf{mm}} \end{array} \right\} \right))$

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

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

$(mpprint-pop)$

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

$(fpprint-tab \left\{ \begin{array}{l} :line \\ :line-relative \\ :section \\ :section-relative \end{array} \right\} c \ i \ [ \widetilde{stream}_{\mathbf{v} \text{ standard-output} *} ])$

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

$(fpprint-indent \left\{ \begin{array}{l} :block \\ :current \end{array} \right\} n \ [ \widetilde{stream}_{\mathbf{v} \text{ standard-output} *} ])$

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

$(mpprint-exit-if-list-exhausted)$

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

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

▷ 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*`<sub>[I]</sub>     ▷ Radix for printing rationals, from 2 to 36.
- `v*print-case*`<sub>[upcase]</sub>
  - ▷ Print symbol names all uppercase (**:upcase**), all lowercase (**:downcase**), capitalized (**:capitalize**).
- `v*print-circle*`<sub>[NIL]</sub>
  - ▷ If T, avoid indefinite recursion while printing circular structure.
- `v*print-escape*`<sub>[NIL]</sub>
  - ▷ If NIL, do not print escape characters and package prefixes.
- `v*print-gensym*`<sub>[NIL]</sub>   ▷ If T, print **#:** before uninterned symbols.
- `v*print-length*`<sub>[NIL]</sub>
- `v*print-level*`<sub>[NIL]</sub>
- `v*print-lines*`<sub>[NIL]</sub>
  - ▷ 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*`<sub>[NIL]</sub>   ▷ If T, print rationals with a radix indicator.
- `v*print-readably*`<sub>[NIL]</sub>
  - ▷ If T, print *f* readably or signal error **print-not-readable**.
- `v*print-right-margin*`<sub>[NIL]</sub>
  - ▷ Right margin width in ems while pretty-printing.
- (*f* **set-pprint-dispatch** *type function* [*priority*<sub>[I]</sub> [*table*<sub>[v\*print-pprint-dispatch\*]</sub>])
  - ▷ 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*<sub>[v\*print-pprint-dispatch\*]</sub>])
  - ▷ 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*<sub>[v\*print-pprint-dispatch\*]</sub>])
  - ▷ Return *copy* of *table* or, if *table* is NIL, initial value of **v\*print-pprint-dispatch\***.
- `v*print-pprint-dispatch*`   ▷ Current pretty print dispatch table.

### 13.5 Format

- (*m* **formatter** *control*)
  - ▷ Return *function* of *stream* and *arg\** applying *f* **format** to *stream*, *control*, and *arg\** returning NIL or any excess *args*.
- (*f* **format** {T|NIL|*out-string*|*out-stream*} *control arg\**)
  - ▷ Output string *control* which may contain ~ directives possibly taking some *args*. Alternatively, *control* can be a function returned by *m* **formatter** which is then applied to *out-stream* and *arg\**. Output to *out-string*, *out-stream* or, if first argument is T, to **v\*standard-output\***. Return NIL. If first argument is NIL, return *formatted output*.
- ~ [*min-col*<sub>[I]</sub>] [*,col-inc*<sub>[I]</sub>] [*,min-pad*<sub>[I]</sub>] [*,pad-char*<sub>[I]</sub>]
  - [:] [**@**] {**A**|**S**}
  - ▷ **Aesthetic/Standard**. Print argument of any type for consumption by humans/by the reader, respectively. With :, print NIL as () rather than nil; with @, add *pad-chars* on the left rather than on the right.

- ~ [*radix*<sub>[I]</sub>] [*,width*<sub>[I]</sub>] [*,pad-char*<sub>[I]</sub>] [*,comma-char*<sub>[I]</sub>] [*,comma-interval*<sub>[I]</sub>] [:] [**@**] **R**
  - ▷ **Radix**. (With one or more prefix arguments.) Print argument as number; with :, group digits *comma-interval* each; with @, always prepend a sign.
- {~**R**|~:~**R**|~**@R**|~**@:R**}
  - ▷ **Roman**. Take argument as number and print it as English cardinal number, as English ordinal number, as Roman numeral, or as old Roman numeral, respectively.
- ~ [*width*<sub>[I]</sub>] [*,pad-char*<sub>[I]</sub>] [*,comma-char*<sub>[I]</sub>] [*,comma-interval*<sub>[I]</sub>] [:] [**@**] {**D**|**B**|**O**|**X**}
  - ▷ **Decimal/Binary/Octal/Hexadecimal**. Print integer argument as number. With :, group digits *comma-interval* each; with @, always prepend a sign.
- ~ [*width*<sub>[I]</sub>] [*,dec-digits*<sub>[I]</sub>] [*,shift*<sub>[I]</sub>] [*,overflow-char*<sub>[I]</sub>] [*,pad-char*<sub>[I]</sub>] [**@**] **F**
  - ▷ **Fixed-Format Floating-Point**. With @, always prepend a sign.
- ~ [*width*<sub>[I]</sub>] [*,dec-digits*<sub>[I]</sub>] [*,exp-digits*<sub>[I]</sub>] [*,scale-factor*<sub>[I]</sub>] [*,overflow-char*<sub>[I]</sub>] [*,pad-char*<sub>[I]</sub>] [*,exp-char*<sub>[I]</sub>] [**@**] {**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 @, always prepend a sign.
- ~ [*dec-digits*<sub>[I]</sub>] [*,int-digits*<sub>[I]</sub>] [*,width*<sub>[I]</sub>] [*,pad-char*<sub>[I]</sub>] [:] [**@**] **\$**
  - ▷ **Monetary Floating-Point**. Print argument as fixed-format floating-point number. With :, put sign before any padding; with @, always prepend a sign.
- {~**C**|~:~**C**|~**@C**|~**@:C**}
  - ▷ **Character**. Print, spell out, print in #\ syntax, or tell how to type, respectively, argument as (possibly non-printing) character.
- {~( *text* ~)|~:( *text* ~)|~**@**( *text* ~)|~**@:**( *text* ~)}
- ▷ **Case-Conversion**. Convert *text* to lowercase, convert first letter of each word to uppercase, capitalize first word and convert the rest to lowercase, or convert to uppercase, respectively.
- {~**P**|~:~**P**|~**@P**|~**@:P**}
  - ▷ **Plural**. If argument *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*<sub>[I]</sub>] %     ▷ **Newline**. Print *n* newlines.
- ~ [*n*<sub>[I]</sub>] &
  - ▷ **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*<sub>[I]</sub>] |     ▷ **Page**. Print *n* page separators.
- ~ [*n*<sub>[I]</sub>] ~     ▷ **Tilde**. Print *n* tildes.
- ~ [*min-col*<sub>[I]</sub>] [*,col-inc*<sub>[I]</sub>] [*,min-pad*<sub>[I]</sub>] [*,pad-char*<sub>[I]</sub>] [:] [**@**] < [*nl-text* ~ [*spare*<sub>[I]</sub>] [*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.

(*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 22.

### 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*\*)\*)  
})  
▷ 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*[\*packages\*]])  
▷ 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.

*\*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*.

## 13.6 Streams

(*f*open *path* {  
  :direction {  
    :input  
    :output  
    :io  
    :probe  
  }  
  :element-type {  
    :default  
  } *character*  
  :if-exists {  
    :new-version  
    :error  
    :rename  
    :rename-and-delete  
    :overwrite  
    :append  
    :supersede  
    NIL  
  } {  
    :new-version if *path*  
    specifies :newest;  
    NIL otherwise  
  }  
  :if-does-not-exist {  
    :error  
    :create  
    NIL  
  } [NIL for :direction :probe;  
  { :create :error } otherwise]  
  :external-format *format*[default]  
})

▷ Open file-stream to *path*.

(*f*make-concatenated-stream *input-stream*\*)  
(*f*make-broadcast-stream *output-stream*\*)  
(*f*make-two-way-stream *input-stream-part* *output-stream-part*)  
(*f*make-echo-stream *from-input-stream* *to-output-stream*)  
(*f*make-synonym-stream *variable-bound-to-stream*)  
▷ Return stream of indicated type.

(*f*make-string-input-stream *string* [*start*[0] [*end*[NIL]]])  
▷ Return a string-stream supplying the characters from *string*.

(*f*make-string-output-stream [:element-type *type*[character]])  
▷ Return a string-stream accepting characters (available via *f*get-output-stream-string).

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

(*f*two-way-stream-input-stream *two-way-stream*)  
(*f*two-way-stream-output-stream *two-way-stream*)  
(*f*echo-stream-input-stream *echo-stream*)  
(*f*echo-stream-output-stream *echo-stream*)  
▷ Return source stream or sink stream of *two-way-stream*/*echo-stream*, respectively.

(*f*synonym-stream-symbol *synonym-stream*)  
▷ Return symbol of *synonym-stream*.

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

(*f*file-position *stream* {  
  :start  
  :end  
  :position  
})  
▷ Return position within stream, or set it to *position* and return T on success.

(*f*file-string-length *stream* *foo*)  
▷ Length *foo* would have in *stream*.

(*f*listen [*stream*[\*standard-input\*]])  
▷ T if there is a character in input *stream*.

(*f*clear-input [*stream*[\*standard-input\*]])  
▷ Clear input from *stream*, return NIL.

{  
  (*f*clear-output)  
  (*f*force-output)  
  (*f*finish-output)  
} [*stream*[\*standard-output\*]])  
▷ End output to *stream* and return NIL immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.

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

(*m*with-open-file (*stream path open-arg\**) (declare *decl\**)<sup>*P*</sup> *form\**)  
 ▷ Use *f*open with *open-args* to temporarily create *stream* to *path*; return values of forms.

(*m*with-open-stream (*foo stream*) (declare *decl\**)<sup>*P*</sup> *form\**)  
 ▷ Evaluate *forms* with *foo* locally bound to *stream*. Return values of forms.

(*m*with-input-from-string (*foo string* {*:index* *index*  
*:start* *start*<sub>0</sub>  
*:end* *end*<sub>NIL</sub>}) (declare *decl\**)<sup>*P*</sup> *form\**)  
 ▷ 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*<sub>NIL</sub> [*:element-type* *type*<sub>character</sub>]]) (declare *decl\**)<sup>*P*</sup> *form\**)  
 ▷ Evaluate *forms* with *foo* locally bound to an output **string-stream**. Append output to *string* and return values of forms if *string* is given. Return string containing output otherwise.

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

*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*<sub>NIL</sub>:*unspecific*}  
*:device* {*device*<sub>NIL</sub>:*unspecific*}  
*:directory* {*directory*:*wild*<sub>NIL</sub>:*unspecific*}  
 {*:absolute* {*directory*  
*:wild*  
*:wild-inferiors*}  
*:relative* {*directory*  
*:up*  
*:back*}  
*:name* {*file-name*:*wild*<sub>NIL</sub>:*unspecific*}  
*:type* {*file-type*:*wild*<sub>NIL</sub>:*unspecific*}  
*:version* {*newest* *version*:*wild*<sub>NIL</sub>:*unspecific*}  
*:defaults* *path*<sub>host from v\*default-pathname-defaults\*</sub>  
*:case* {*:local*:*common*}<sub>local</sub>})

▷ 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* [*:case* {*:local*  
*:common*}<sub>local</sub>])

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

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

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

*v*\*default-pathname-defaults\*

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

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

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

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

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

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

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

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

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

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

(*f*logical-pathname *logical-path-or-stream*)  
 ▷ Logical pathname of *logical-path-or-stream*. Logical pathnames are represented as all-uppercase  
 "[*host*:*dir*]{*dir*\*}+};}\*{*name*\*}[.]{*type*\*}+}[.]{*version*\*  
 [*newest*<sub>NEWEST</sub>]}]".

(*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.

## 15.3 REPL and Debugging

$v+$  |  $v++$  |  $v+++$   
 $v*$  |  $v**$  |  $v***$   
 $v/$  |  $v//$  |  $v///$

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

$v-$  ▷ Form currently being evaluated by the REPL.

( $f$ apropos *string* [*package*<sub>NTI</sub>])  
 ▷ Print interned symbols containing *string*.

( $f$ apropos-list *string* [*package*<sub>NTI</sub>])  
 ▷ List of interned symbols containing *string*.

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

( $f$ ed [*file-or-function*<sub>NTI</sub>]) ▷ Invoke editor if possible.

( $f$ macroexpand-1) | ( $f$ macroexpand) *form* [*environment*<sub>NTI</sub>]  
 ▷ Return macro expansion, once or entirely, respectively, of *form* and T if *form* was a macro form. Return form and NIL otherwise.

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

( $m$ trace {*function* | ( $setf$  *function*)})  
 ▷ Cause *functions* to be traced. With no arguments, return list of traced functions.

( $m$ untrace {*function* | ( $setf$  *function*)})  
 ▷ Stop *functions*, or each currently traced function, from being traced.

$v*$ trace-output\*  
 ▷ Output stream  $m$ trace and  $m$ time send their output to.

( $m$ step *form*)  
 ▷ Step through evaluation of *form*. Return values of form.

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

( $m$ time *form*)  
 ▷ Evaluate *forms* and print timing information to  $v*$ trace-output\*. Return values of form.

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

( $f$ describe *foo* [*stream*<sub>v\*standard-output\*</sub>])  
 ▷ Send information about *foo* to *stream*.

( $g$ describe-object *foo* [*stream*])  
 ▷ Send information about *foo* to *stream*. Called by  $f$ describe.

( $f$ disassemble *function*)  
 ▷ Send disassembled representation of *function* to  $v*$ standard-output\*. Return NIL.

( $f$ room [{NIL|default|T}|default])  
 ▷ Print information about internal storage management to  $v*$ standard-output\*.

( $f$ 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*<sub>v\*package\*</sub>]  
 ▷ 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*<sub>v\*package\*</sub>])  
 ▷ Remove *symbol* from *package*, return T on success.

( $f$ import |  $f$ shadowing-import) *symbols* [*package*<sub>v\*package\*</sub>]  
 ▷ 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*<sub>v\*package\*</sub>])  
 ▷ 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*<sub>v\*package\*</sub>])  
 ▷ Make *symbols* external to *package*. Return T.

( $f$ unexport *symbols* [*package*<sub>v\*package\*</sub>])  
 ▷ Revert *symbols* to internal status. Return T.

( $m$ do-symbols |  $m$ do-external-symbols |  $m$ do-all-symbols (*var* [*package*<sub>v\*package\*</sub>] [*result*<sub>NTI</sub>]))  
 (declare *decl*\*) \* (*tag* | *form*) \*  
 ▷ Evaluate  $s$ 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  $s$ block named NIL.

( $m$ with-package-iterator (*foo packages* [:internal|:external|:inherited])  
 (declare *decl*\*) \* *form*<sub>P</sub>\*)  
 ▷ 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*<sub>NTI</sub>])  
 ▷ 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.

(*fgensym* [*s*])  
 ▷ Return fresh, uninterned symbol `#:sn` with *n* from `v*gensym-counter*`. Increment `v*gensym-counter*`.

(*fgentemp* [*prefix*] [*package*])  
 ▷ Intern fresh symbol in package. Deprecated.

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

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

(*gdocumentation* (*setf* *gdocumentation*) *new-doc*) *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.

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

`nil`<sub>c</sub>  
 ▷ 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`<sub>cl</sub>  
 ▷ Exports the defined names of Common Lisp except for those in the **keyword** package.

`common-lisp-user`<sub>cl-user</sub>  
 ▷ Current package after startup; uses package **common-lisp**.

**keyword**  
 ▷ Contains symbols which are defined to be of type **keyword**.

## 15 Compiler

### 15.1 Predicates

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

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

### 15.2 Compilation

(*fcompile*  $\left\{ \begin{array}{l} \text{NIL} \text{ } \text{definition} \\ \text{name} \\ \text{(setf name)} \end{array} \right\}$  [*definition*])  
 ▷ Return compiled function or replace *name*'s function definition with the compiled function. Return `T` in case of **warnings** or **errors**, and `T` in case of **warnings** or **errors** excluding **style-warnings**.

(*fcompile-file* *file*  $\left\{ \begin{array}{l} \text{:output-file } \text{out-path} \\ \text{:verbose } \text{bool} \text{ } \text{v*compile-verbose*} \\ \text{:print } \text{bool} \text{ } \text{v*compile-print*} \\ \text{:external-format } \text{file-format} \text{ } \text{v*compile-external-format*} \end{array} \right\}$ )  
 ▷ Write compiled contents of *file* to *out-path*. Return `true` output path or `NIL`, `T` in case of **warnings** or **errors**, `T` in case of **warnings** or **errors** excluding **style-warnings**.

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

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

`v*compile-file`  $\left\{ \begin{array}{l} \text{pathname*} \text{ } \text{v*compile-file-pathname*} \\ \text{truenamename*} \text{ } \text{v*load} \end{array} \right\}$   
 ▷ Input file used by *fcompile-file*/by *fload*.

`v*compile`  $\left\{ \begin{array}{l} \text{print*} \\ \text{verbose*} \end{array} \right\}$   
 ▷ Defaults used by *fcompile-file*/by *fload*.

(*seval-when* ( $\left\{ \begin{array}{l} \text{:compile-toplevel} \text{ } \text{compile} \\ \text{:load-toplevel} \text{ } \text{load} \\ \text{:execute} \text{ } \text{eval} \end{array} \right\}$ ) *forms*<sup>P</sup>)  
 ▷ Return values of forms if *seval-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.)

(*slocally* (*declare*  $\widehat{\text{decl}}$ )\* *form*<sup>P</sup>)  
 ▷ Evaluate *forms* in a lexical environment with declarations *decl* in effect. Return values of forms.

(*mwith-compilation-unit* (*override* *bool*) *form*<sup>P</sup>)  
 ▷ Return values of forms. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of *forms*.

(*sload-time-value* *form* [*read-only*])  
 ▷ Evaluate *form* at compile time and treat its value as literal at run time.

(*squote*  $\widehat{\text{foo}}$ ) ▷ Return unevaluated *foo*.

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

(*fmake-load-form-saving-slots* *foo*  $\left\{ \begin{array}{l} \text{:slot-names } \text{slots} \text{ } \text{v*make-load-form-saving-slots-slot-names*} \\ \text{:environment } \text{environment} \end{array} \right\}$ )  
 ▷ 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*.

(*fmacro-function* *symbol* [*environment*])  
 (*fcompiler-macro-function*  $\left\{ \begin{array}{l} \text{name} \\ \text{(setf name)} \end{array} \right\}$  [*environment*])  
 ▷ Return specified macro function, or compiler macro function, respectively, if any. Return `NIL` otherwise. **setfable**.

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



NINTH 9  
NO-APPLICABLE-METHOD 27  
NO-NEXT-METHOD 27  
NOT 17, 33, 36  
NOTANY 13  
NOTEVERY 12  
NOTINLINE 49  
NRECONC 10  
NREVERSE 13  
NSET-DIFFERENCE 11  
NSET-EXCLUSIVE-OR 11  
NSTRING-CAPITALIZE 8  
NSTRING-DOWNCASE 8  
NSTRING-UPCASE 8  
NSUBLIS 11  
NSUBST 10  
NSUBST-IF 10  
NSUBST-IF-NOT 10  
NSUBSTITUTE 14  
NSUBSTITUTE-IF 14  
NSUBSTITUTE-IF-NOT 14  
NTH 9  
NTH-VALUE 19  
NTHCDR 9  
NULL 8, 32  
NUMBER 32  
NUMBERP 3  
NUMERATOR 4  
NUNION 11

ODDP 3  
OF 24  
OF-TYPE 22  
ON 24  
OPEN 41  
OPEN-STREAM-P 33  
OPTIMIZE 49  
OR 21, 28, 33, 36  
OTHERWISE 21, 31  
OUTPUT-STREAM-P 33

PACKAGE 32  
PACKAGE-ERROR 32  
PACKAGE-ERROR-PACKAGE 31  
PACKAGE-NAME 44  
PACKAGE-NICKNAMES 44  
PACKAGE-SHADOWING-SYMBOLS 45  
PACKAGE-USE-LIST 44  
PACKAGE-USED-BY-LIST 44  
PACKAGEP 44  
PAIRLIS 10  
PARSE-ERROR 32  
PARSE-INTEGER 8  
PARSE-NAMESTRING 43  
PATHNAME 32, 43  
PATHNAME-DEVICE 42  
PATHNAME-DIRECTORY 42  
PATHNAME-HOST 42  
PATHNAME-MATCH-P 33  
PATHNAME-NAME 42  
PATHNAME-TYPE 42  
PATHNAME-VERSION 42  
PATHNAMEP 33  
PEEK-CHAR 34  
PHASE 4  
PI 3  
PLUSP 3  
POP 9  
POSITION 14  
POSITION-IF 14  
POSITION-IF-NOT 14  
PPRINT 36  
PPRINT-DISPATCH 38  
PPRINT-EXIT-IF-LIST-EXHAUSTED 37  
PPRINT-FILL 37  
PPRINT-INDENT 37  
PPRINT-LINEAR 37  
PPRINT-LOGICAL-BLOCK 37  
PPRINT-NEWLINE 37  
PPRINT-POP 37  
PPRINT-TAB 37  
PPRINT-TABULAR 37  
PRESENT-SYMBOL 24  
PRESENT-SYMBOLS 24  
PRIN1 36  
PRIN1-TO-STRING 36  
PRINC 36  
PRINC-TO-STRING 36  
PRINT 36  
PRINT-NOT-READABLE 32  
PRINT-NOT-READABLE-OBJECT 31  
PRINT-OBJECT 36  
PRINT-UNREADABLE-OBJECT 36

PROBE-FILE 43  
PROCLAIM 49  
PROG 21  
PROG1 21  
PROG2 21  
PROG\* 21  
PROGN 21, 28  
PROGRAM-ERROR 32  
PROGV 17  
PROVIDE 45  
PSETF 17  
PSETQ 17  
PUSH 10  
PUSHNEW 10

QUOTE 35, 47

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

SAFETY 49  
SATISFIES 33  
SBIT 12  
SCALE-FLOAT 6  
SCHAR 8  
SEARCH 14  
SECOND 9  
SEQUENCE 32  
SERIOUS-CONDITION 32  
SET 17  
SET-DIFFERENCE 11  
SET-DISPATCH-MACRO-CHARACTER 35  
SET-EXCLUSIVE-OR 11  
SET-MACRO-CHARACTER 35  
SET-PPRINT-DISPATCH 38  
SET-SYNTAX-FROM-CHAR 34  
SETF 17, 46  
SETQ 17  
SEVENTH 9  
SHADOW 45  
SHADOWING-IMPORT 45  
SHARED-INITIALIZE 26  
SHIFTF 17

SHORT-FLOAT 32, 35  
SHORT-FLOAT-EPSILON 6  
SHORT-FLOAT-NEGATIVE-EPSILON 6  
SHORT-SITE-NAME 49  
SIGNAL 29  
SIGNED-BYTE 32  
SIGNUM 4  
SIMPLE-ARRAY 32  
SIMPLE-BASE-STRING 32  
SIMPLE-BIT-VECTOR 32  
SIMPLE-BIT-VECTOR-P 11  
SIMPLE-CONDITION-FORMAT-ARGUMENTS 31  
SIMPLE-CONDITION-FORMAT-CONTROL 31  
SIMPLE-ERROR 32  
SIMPLE-STRING 32  
SIMPLE-STRING-P 8  
SIMPLE-TYPE-ERROR 32  
SIMPLE-VECTOR 32  
SIMPLE-VECTOR-P 11  
SIMPLE-WARNING 32  
SIN 3  
SINGLE-FLOAT 32, 35  
SINGLE-FLOAT-EPSILON 6  
SINGLE-FLOAT-NEGATIVE-EPSILON 6  
SINH 4  
SIXTH 9  
SLEEP 22  
SLOT-BOUND 25  
SLOT-EXISTS-P 25  
SLOT-MAKUNBOUND 25  
SLOT-MISSING 26  
SLOT-UNBOUND 26  
SLOT-VALUE 25  
SOFTWARE-TYPE 49  
SOFTWARE-VERSION 49  
SOME 13  
SORT 13  
SPACE 7, 49  
SPECIAL 49  
SPECIAL-OPERATOR-P 46  
SPEED 49  
SQRT 3  
STABLE-SORT 13  
STANDARD 28  
STANDARD-CHAR 7, 32  
STANDARD-CHAR-P 7  
STANDARD-CLASS 32  
STANDARD-GENERIC-FUNCTION 32  
STANDARD-METHOD 32  
STANDARD-OBJECT 32  
STEP 48  
STORAGE-CONDITION 32  
STORE-VALUE 30  
STREAM 32  
STREAM-ELEMENT-TYPE 33  
STREAM-ERROR 32  
STREAM-ERROR-STREAM 31  
STREAM-EXTERNAL-FORMAT 42  
STREAMP 33  
STRING 8, 32  
STRING-CAPITALIZE 8  
STRING-DOWNCASE 8  
STRING-EQUAL 8  
STRING-GREATERP 8  
STRING-LEFT-TRIM 8  
STRING-LESSP 8  
STRING-NOT-EQUAL 8  
STRING-NOT-GREATERP 8  
STRING-NOT-LESSP 8  
STRING-RIGHT-TRIM 8  
STRING-STREAM 32  
STRING-TRIM 8  
STRING-UPCASE 8  
STRING/= 8  
STRING< 8  
STRING<= 8  
STRING= 8  
STRING> 8  
STRING>= 8  
STRINGP 8  
STRUCTURE 46  
STRUCTURE-CLASS 32  
STRUCTURE-OBJECT 32  
STYLE-WARNING 32  
SUBLIS 11  
SUBSEQ 13  
SUBSETP 9  
SUBST 10

SUBST-IF 10  
SUBST-IF-NOT 10  
SUBSTITUTE 14  
SUBSTITUTE-IF 14  
SUBSTITUTE-IF-NOT 14  
SUBTYPEP 31  
SUM 24  
SUMMING 24  
SVREF 12  
SXHASH 16  
SYMBOL 24, 32, 45  
SYMBOL-FUNCTION 46  
SYMBOL-MACROLET 20  
SYMBOL-NAME 46  
SYMBOL-PACKAGE 46  
SYMBOL-PLIST 46  
SYMBOL-VALUE 46  
SYMBOLP 44  
SYMBOLS 24  
SYNONYM-STREAM 32  
SYNONYM-STREAM-SYMBOL 41

T 2, 32, 46  
TAGBODY 22  
TAILP 9  
TAN 3  
TANH 4  
TENTH 9  
TERPRI 36  
THE 24, 31  
THEN 24  
THEREIS 25  
THIRD 9  
THROW 22  
TIME 48  
TO 24  
TRACE 48  
TRANSLATE-LOGICAL-PATHNAME 43  
TRANSLATE-PATHNAME 43  
TREE-EQUAL 10  
TRUENAME 43  
TRUNCATE 4  
TWO-WAY-STREAM 32  
TWO-WAY-STREAM-INPUT-STREAM 41  
TWO-WAY-STREAM-OUTPUT-STREAM 41  
TYPE 46, 49  
TYPE-ERROR 32  
TYPE-ERROR-DATUM 31  
TYPE-ERROR-EXPECTED-TYPE 31  
TYPE-OF 33  
TYPECASE 31  
TYPEP 31

UNBOUND-SLOT 32  
UNBOUND-SLOT-INSTANCE 31  
UNBOUND-VARIABLE 32  
UNDEFINED-FUNCTION 32  
UNEXPORT 45  
UNINTERN 45  
UNION 11  
UNLESS 21, 24  
UNREAD-CHAR 34  
UNRECORDED-CHAR 32  
UNTRACE 48  
UNUSE-PACKAGE 44  
UNWIND-PROTECT 11  
UPDATE-INSTANCE-FOR-DIFFERENT-CLASS 26  
UPDATE-INSTANCE-FOR-REDEFINED-CLASS 26  
UPFROM 24  
UPGRADED-ARRAY-ELEMENT-TYPE 33  
UPGRADED-COMPLEX-PART-TYPE 6  
UPPER-CASE-P 7  
UPTO 24  
USE-PACKAGE 44  
USE-VALUE 30  
USER-HOMEDIR-PATHNAME 43  
USING 24

V 40  
VALUES 18, 33  
VALUES-LIST 18  
VARIABLE 46  
VECTOR 12, 32  
VECTOR-POP 12  
VECTOR-PUSH 12  
VECTOR-PUSH-EXTEND 12  
VECTORP 11

WARN 29  
WARNING 32

## 15.4 Declarations

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

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

**(declaration foo\*)** ▷ Make *foos* names of declarations.

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

**([*type*] *type variable*\*)**  
**(*ftype* *type function*\*)**  
 ▷ Declare *variables* or *functions* to be of *type*.

**([*ignorable*] {*var* (*function function*)}\*)**  
**([*ignore*])**  
 ▷ Suppress warnings about used/unused bindings.

**(inline *function*\*)**  
**(notinline *function*\*)**  
 ▷ Tell compiler to integrate/not to integrate, respectively, called *functions* into the calling routine.

**(optimize {*compilation-speed*(*compilation-speed* *n*<sub>3</sub>)  
*debug*(*debug* *n*<sub>3</sub>)  
*safety*(*safety* *n*<sub>3</sub>)  
*space*(*space* *n*<sub>3</sub>)  
*speed*(*speed* *n*<sub>3</sub>)**)

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

**(special *var*\*)** ▷ Declare *vars* to be dynamic.

## 16 External Environment

**(*f*get-internal-real-time)**  
**(*f*get-internal-run-time)**  
 ▷ Current time, or computing time, respectively, in clock ticks.

**internal-time-units-per-second**  
 ▷ Number of clock ticks per second.

**(*f*encode-universal-time *sec min hour date month year* [*zone* *curr*])**  
**(*f*get-universal-time)**  
 ▷ Seconds from 1900-01-01, 00:00, ignoring leap seconds.

**(*f*decode-universal-time *universal-time* [*time-zone* *current*])**  
**(*f*get-decoded-time)**  
 ▷ Return second, minute, hour, date, month, year, day, daylight-p, and zone.

**(*f*short-site-name)**  
**(*f*long-site-name)**  
 ▷ String representing physical location of computer.

**(*f*lisp-implementation)**  
**(*f*software)**  
**(*f*machine)**  
 ▷ Name or version of implementation, operating system, or hardware, respectively.

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

## Index

- " 35  
' 35  
( 35  
) 46  
\* 3, 32, 33, 43, 48  
\*\* 43, 48  
\*\*\* 48  
\*BREAK-ON-SIGNALS\* 31  
\*COMPILE-FILE-PATHNAME\* 47  
\*COMPILE-FILE-TRUENAME\* 47  
\*COMPILE-PRINT\* 47  
\*COMPILE-VERBOSE\* 47  
\*DEBUG-IO\* 42  
\*DEBUGGER-HOOK\* 31  
\*DEFAULT-PATHNAME-DEFAULTS\* 43  
\*ERROR-OUTPUT\* 42  
\*FEATURES\* 36  
\*GENSYM-COUNTER\* 46  
\*LOAD-PATHNAME\* 47  
\*LOAD-PRINT\* 47  
\*LOAD-TRUENAME\* 47  
\*LOAD-VERBOSE\* 47  
\*MACROEXPAND-HOOK\* 48  
\*MODULES\* 45  
\*PACKAGE\* 44  
\*PRINT-ARRAY\* 38  
\*PRINT-BASE\* 38  
\*PRINT-CASE\* 38  
\*PRINT-CIRCLE\* 38  
\*PRINT-ESCAPE\* 38  
\*PRINT-GENSYM\* 38  
\*PRINT-LENGTH\* 38  
\*PRINT-LEVEL\* 38  
\*PRINT-LINES\* 38  
\*PRINT-MISER-WIDTH\* 38  
\*PRINT-PPRINT-DISPATCH\* 38  
\*PRINT-PRETTY\* 38  
\*PRINT-RADIX\* 38  
\*PRINT-READABLY\* 38  
\*PRINT-RIGHT-MARGIN\* 38  
\*QUERY-IO\* 42  
\*RANDOM-STATE\* 4  
\*READ-BASE\* 35  
\*READ-DEFAULT-FLOAT-FORMAT\* 35  
\*READ-EVAL\* 36  
\*READ-SUPPRESS\* 35  
\*READTABLE\* 34  
\*STANDARD-INPUT\* 42  
\*STANDARD-OUTPUT\* 42  
\*TERMINAL-IO\* 42  
\*TRACE-OUTPUT\* 48  
+ 3, 28, 48  
++ 48  
+++ 48  
. 35  
.. 35  
@ 35  
- 3, 48  
/ 3, 35, 48  
// 48  
/// 48  
/= 3  
: 44  
:: 44  
:ALLOW-OTHER-KEYS 21  
: 35  
< 3  
<= 3  
= 3, 22, 24  
> 3  
>= 3  
\ 36  
# 40  
#\ 35  
# 35  
#( 35  
#\* 36  
#+ 36  
#- 36  
# 36  
#< 36  
#= 36  
#A 35  
#B 35  
#C( 35  
#O 35  
#P 36  
#R 35  
#S( 36  
#X 35  
## 36  
#| 35  
&ALLOW-OTHER-KEYS 21  
&AUX 21  
&BODY 20  
&ENVIRONMENT 21  
&KEY 20  
&OPTIONAL 20  
&REST 20  
&WHOLE 20  
~( ~) 39  
~\* 40  
~/ / 40  
~< ~:> 40  
~< ~> 39  
~? 40  
~A 38  
~B 39  
~C 39  
~D 39  
~E 39  
~F 39  
~G 39  
~I 40  
~O 39  
~P 39  
~R 39  
~S 38  
~T 40  
~W 40  
~X 39  
~[ ~] 40  
~\$ 39  
~% 39  
~& 39  
~^ 40  
~\_ 39  
~| 39  
~{ ~} 40  
~> 39  
~<~> 39  
~ 35  
| 36  
|+ 3  
|- 3  
ABORT 30  
ABOVE 24  
ABS 4  
ACONS 10  
ACOS 3  
ACOSH 4  
ACROSS 24  
ADD-METHOD 27  
ADJOIN 9  
ADJUST-ARRAY 11  
ADJUSTABLE-ARRAY-P 11  
ALLOCATE-INSTANCE 26  
ALPHA-CHAR-P 7  
ALPHANUMERICP 7  
ALWAYS 25  
AND 21, 22, 24, 28, 33, 36  
APPEND 10, 24, 28  
APPENDING 24  
APPLY 18  
APRÓPOS 48  
APRÓPOS-LIST 48  
AREF 11  
ARITHMETIC-ERROR 32  
ARITHMETIC-ERROR-OPERANDS 31  
ARITHMETIC-ERROR-OPERATION 31  
ARRAY 32  
ARRAY-DIMENSION 11  
ARRAY-DIMENSION-LIMIT 12  
ARRAY-DIMENSIONS 11  
ARRAY-  
DISPLACEMENT 12  
ARRAY-  
ELEMENT-TYPE 33  
ARRAY-HAS-FILL-POINTER-P 11  
ARRAY-IN-BOUNDS-P 11  
ARRAY-RANK 11  
ARRAY-RANK-LIMIT 12  
ARRAY-ROW-MAJOR-INDEX 11  
ARRAY-TOTAL-SIZE 11  
ARRAY-TOTAL-SIZE-LIMIT 12  
ARRAYP 11  
AS 22  
ASH 6  
ASIN 3  
ASINH 4  
ASSERT 29  
ASSOC 10  
ASSOC-IF 10  
ASSOC-IF-NOT 10  
ATAN 4  
ATANH 4  
ATOM 9, 32  
BASE-CHAR 32  
BASE-STRING 32  
BEING 24  
BELOW 24  
BIGNUM 32  
BIT 12, 32  
BIT-AND 12  
BIT-ANDC1 12  
BIT-ANDC2 12  
BIT-EQV 12  
BIT-IOR 12  
BIT-NAND 12  
BIT-NOR 12  
BIT-NOT 12  
BIT-ORC1 12  
BIT-ORC2 12  
BIT-VECTOR 32  
BIT-VECTOR-P 11  
BIT-XOR 12  
BLOCK 21  
BOOLE 5  
BOOLE-1 5  
BOOLE-2 5  
BOOLE-AND 5  
BOOLE-ANDC1 5  
BOOLE-ANDC2 5  
BOOLE-C1 5  
BOOLE-C2 5  
BOOLE-CLR 5  
BOOLE-EQV 5  
BOOLE-IOR 5  
BOOLE-NAND 5  
BOOLE-NOR 5  
BOOLE-ORC1 5  
BOOLE-ORC2 5  
BOOLE-SET 5  
BOOLE-XOR 5  
BOOLEAN 32  
BOTH-CASE-P 7  
BOUNDP 17  
BREAK 48  
BROADCAST-STREAM 32  
BROADCAST-STREAM-STREAMS 41  
BUILT-IN-CLASS 32  
BUTLAST 9  
BY 24  
BYTE 6  
BYTE-POSITION 6  
BYTE-SIZE 6  
CAAR 9  
CADR 9  
CALL-ARGUMENTS-LIMIT 19  
CALL-METHOD 28  
CALL-NEXT-METHOD 27  
CAR 9  
CASE 21  
CATCH 22  
CCASE 21  
CDAR 9  
CDDR 9  
CDR 9  
CEILING 4  
CELL-ERROR 32  
CELL-ERROR-NAME 31  
CERROR 29  
CHANGE-CLASS 26  
CHAR 8  
CHAR-CODE 7  
CHAR-CODE-LIMIT 7  
CHAR-DOWNCASE 7  
CHAR-EQUAL 7  
CHAR-GREATERP 7  
CHAR-INT 7  
CHAR-LESSP 7  
CHAR-NAME 7  
CHAR-NOT-EQUAL 7  
CHAR-NOT-GREATERP 7  
CHAR-NOT-LESSP 7  
CHAR-UPCASE 7  
CHAR/= 7  
CHAR< 7  
CHAR<= 7  
CHAR= 7  
CHAR> 7  
CHAR>= 7  
CHARACTER 7, 32, 35  
CHARACTERP 7  
CHECK-TYPE 33  
CIS 4  
CL 4  
CL-USER 46  
CLASS 32  
CLASS-NAME 26  
CLASS-OF 26  
CLEAR-INPUT 41  
CLEAR-OUTPUT 41  
CLOSE 42  
CLR 1  
CLRHASH 15  
CODE-CHAR 7  
COERCE 31  
COLLECT 24  
COLLECTING 24  
COMMON-LISP 46  
COMMON-LISP-USER 46  
COMPILE-SPEED 49  
COMPILE 46  
COMPILE-FILE 47  
COMPILE-FILE-PATHNAME 47  
COMPILED-FUNCTION 32  
COMPILED-FUNCTION-P 46  
COMPILER-MACRO 46  
COMPILER-MACRO-FUNCTION 47  
COMPLEMENT 19  
COMPLEX 4, 32, 35  
COMPLEXP 3  
COMPUTE-APPLICABLE-METHODS 27  
COMPUTE-RESTARTS 30  
CONCATENATE 13  
CONCATENATED-STREAM 32  
CONCATENATED-STREAM-STREAMS 41  
CONCATENATED-STREAMS 41  
CONDI 21  
CONDITION 32  
CONJUGATE 4  
CONS 9, 32  
CONSP 8  
CONSTANTLY 19  
CONSTANTP 17  
CONTINUE 30  
CONTROL-ERROR 32  
COPY-ALIST 10  
COPY-LIST 10  
COPY-PPRINT-DISPATCH 38  
COPY-READTABLE 34  
COPY-SEQ 15  
COPY-STRUCTURE 16  
COPY-SYMBOL 46  
COPY-TREE 11  
COS 3  
COSH 4  
COUNT 13, 24  
COUNT-IF 13  
COUNT-IF-NOT 13  
COUNTING 24  
CTYPECASE 31  
DEBUG 49  
DECLF 3  
DECLAIM 49  
DECLARATION 49  
DECLARE 49  
DECODE-FLOAT 6  
DECODE-UNIVERSAL-TIME 49  
DEFCLASS 25  
DEFCONSTANT 17  
DEFGENERIC 26  
DEFINE-COMPILER-MACRO 19  
DEFINE-CONDITION 29  
DEFINE-METHOD-COMBINATION 28  
DEFINE-MODIFY-MACRO 20  
DEFINE-SETF-EXPANDER 20  
DEFINE-SYMBOL-MACRO 20  
DEFMACRO 19  
DEFMETHOD 27  
DEFPARAMETER 17  
DEFSETF 20  
DEFSTRUCT 16  
DEFTYPE 33  
DEFUN 18  
DEFVAR 17  
DELETE 14  
DELETE-DUPLICATES 14  
DELETE-FILE 43  
DELETE-IF 14  
DELETE-IF-NOT 14  
DELETE-PACKAGE 44  
DENOMINATOR 4  
DEPOSIT-FIELD 6  
DESCRIBE 48  
DESCRIBE-OBJECT 48  
DESTRUCTURING-BIND 18  
DIGIT-CHAR 7  
DIGIT-CHAR-P 7  
DIRECTOR 44  
DIRECTOR-DIRECTORY-NAMESTRING 43  
DISASSEMBLE 48  
DIVISION-BY-ZERO 32  
DO 22, 24  
DO-ALL-SYMBOLS 45  
DO-EXTERNAL-SYMBOLS 45  
DO-SYMBOLS 45  
DO\* 22  
DOCUMENTATION 46  
DOING 24  
DOLIST 22  
DOTIMES 22  
DOUBLE-FLOAT 32, 35  
DOUBLE-FLOAT-EPSON 6  
DOUBLE-FLOAT-NEGATIVE-EPSILON 6  
DOWNFROM 24  
DOWNTO 24  
DPB 6  
DRIBBLE 48  
DYNAMIC-EXTENT 49  
EACH 24  
ECASE 21  
ECHO-STREAM 32  
ECHO-STREAM-INPUT-STREAM 41  
ECHO-STREAM-OUTPUT-STREAM 41  
ED 48  
EIGHTH 9  
ELSE 24  
ELT 13  
ENCODE-UNIVERSAL-TIME 49  
END 24  
END-OF-FILE 32  
ENDP 8  
ENOUGH-NAMESTRING 43  
ENSURE-DIRECTORIES-EXIST 44  
ENSURE-GENERIC-FUNCTION 27  
EQ 16  
EQ 16, 33  
EQUAL 16  
EQUALP 16  
ERROR 29, 32  
ETYPESCASE 31  
EVAL 47  
EVAL-WHEN 47  
EVENP 3  
EVERY 12  
EXP 3  
EXPORT 45  
EXPT 3  
EXTENDED-CHAR 32  
EXTERNAL-SYMBOL 24  
EXTERNAL-SYMBOLS 24  
FBOUND 17  
FCEILING 4  
FDEFINITION 19  
FFLOOR 4  
FIFTH 9  
FILE-AUTHOR 43  
FILE-ERROR 32  
FILE-ERROR-PATHNAME 31  
FILE-LENGTH 43  
FILE-NAMESTRING 43  
FILE-POSITION 41  
FILE-STREAM 32  
FILE-STRING-LENGTH 41  
FILE-WRITE-DATE 43  
FILL 13  
FILL-POINTER 12  
FINALLY 25  
FIND 14  
FIND-ALL-SYMBOLS 45  
FIND-CLASS 25  
FIND-IF 14  
FIND-IF-NOT 14  
FIND-METHOD 27  
FIND-PACKAGE 45  
FIND-RESTART 30  
FIND-SYMBOL 45  
FINISH-OUTPUT 41  
FIRST 9  
FIXNUM 32  
FLET 18  
FLOAT 4, 32  
FLOAT-DIGITS 6  
FLOAT-PRECISION 6  
FLOAT-RADIX 6  
FLOAT-SIGN 4  
FLOATING-POINT-POINT-INEXACT 32  
FLOATING-POINT-INVALID-OPERATION 32  
FLOATING-POINT-OVERFLOW 32  
FLOATING-POINT-UNDERFLOW 32  
FLOATP 3  
FLOOR 4  
FMAKUNBOUND 19  
FOR 22  
FORCE-OUTPUT 41  
FORMAT 38  
FORMATTER 38  
FOURTH 9  
FRESH-LINE 36  
FROM 24  
FROUND 4  
FTRUNCATE 4  
FTYPE 49  
FUNCALL 18  
FUNCTION 18, 32, 35, 46  
FUNCTION-KEYWORDS 28  
FUNCTION-LAMBDA-EXPRESSION 19  
FUNCTIONP 17  
GCD 3  
GENERIC-FUNCTION 32  
GENSYM 46  
GENTEMP 46  
GET 17  
GET-DECODED-TIME 49  
GET-DISPATCH-MACRO-CHARACTER 35  
GET-INTERNAL-REAL-TIME 49  
GET-INTERNAL-RUN-TIME 49  
GET-MACRO-CHARACTER 35  
GET-OUTPUT-STREAM-STRING 41  
GET-PROPERTIES 17  
GET-SETF-EXPANSION 49  
GET-UNIVERSAL-TIME 49  
GETF 17  
GETHASH 15  
GO 22  
GRAPHIC-CHAR-P 7  
HANDLER-BIND 30  
HANDLER-CASE 30  
HASH-KEY 24  
HASH-KEYS 24  
HASH-TABLE 32  
HASH-TABLE-COUNT 15  
HASH-TABLE-P 15  
HASH-TABLE-REHASH-THRESHOLD 15  
HASH-TABLE-SIZE 15  
HASH-TABLE-TEST 15  
HASH-VALUE 24  
HASH-VALUES 24  
HOST-NAMESTRING 43  
IDENTITY 19  
IF 21, 24  
IGNORABLE 49  
IGNORE 49  
IGNORE-ERRORS 29  
IMAGPART 4  
IMPORT 45  
IN 24  
IN-PACKAGE 44  
INCF 3  
INITIALIZE-INSTANCE 26  
INITIALLY 25  
INLINE 49  
INPUT-STREAM-P 33  
INSPECT 48  
INTEGER 32  
INTEGER-DECODE-FLOAT 6  
INTEGER-LENGTH 6  
INTEGER-LOG 5  
LOGNOT 5  
LOGGRC1 5  
LOGGRC2 5  
LOGTEST 5  
LOGXOR 5  
LONG-FLOAT 32, 35  
LONG-FLOAT-EPSILON 6  
LONG-FLOAT-NEGATIVE-EPSILON 6  
LONG-SITE-NAME 49  
LOOP 22  
LOOP-FINISH 25  
LOWER-CASE-P 7  
MACHINE-INSTANCE 49  
MACHINE-TYPE 49  
MACHINE-VERSION 49  
MACRO-FUNCTION 47  
MACROEXPAND 48  
MACROEXPAND-1 48  
MACROLET 20  
MAKE-ARRAY 11  
MAKE-BROADCAST-STREAM 41  
MAKE-CONCATENATED-STREAM 41  
MAKE-CONDITION 29  
MAKE-DISPATCH-MACRO-CHARACTER 35  
MAKE-ECHO-STREAM 41  
MAKE-HASH-TABLE 15  
MAKE-INSTANCE 25  
MAKE-INSTANCES-OBSOLETE 26  
MAKE-LIST 9  
MAKE-LOAD-FORM 47  
MAKE-LOAD-FORM-SAVING-SLOTS 47  
MAKE-METHOD 28  
MAKE-PACKAGE 44  
MAKE-PATHNAME 42  
MAKE-RANDOM-STATE 4  
MAKE-SEQUENCE 13  
MAKE-STRING 8  
MAKE-STRING-INPUT-STREAM 41  
MAKE-STRING-OUTPUT-STREAM 41  
MAKE-SYMBOL 45  
MAKE-SYNONYM-STREAM 41  
MAKE-TWO-WAY-STREAM 41  
MAKUNBOUND 17  
MAP 15  
MAP-INTO 15  
MAPC 10  
MAPCAN 10  
MAPCAR 10  
MAPCON 10  
MAPHASH 15  
MAPL 10  
MAPLIST 10  
MASK-FIELD 6  
MAX 4, 28  
MAXIMIZE 24  
MAXIMIZING 24  
MEMBER 9, 33  
MEMBER-IF 9  
MEMBER-IF-NOT 9  
MERGE 13  
MERGE-PATHNAMES 43  
METHOD 32  
METHOD-COMBINATION 32, 46  
METHOD-ERROR 27  
METHOD-QUALIFIERS 28  
MIN 4, 28  
MINIMIZE 24  
MINIMIZING 24  
MINUSP 3  
MISMATCH 13  
MOD 4, 33  
MOST-NEGATIVE-DOUBLE-FLOAT 6  
MOST-NEGATIVE-FIXNUM 6  
MOST-NEGATIVE-LONG-FLOAT 6  
MOST-NEGATIVE-SHORT-FLOAT 6  
MOST-NEGATIVE-SINGLE-FLOAT 6  
MOST-POSITIVE-DOUBLE-FLOAT 6  
MOST-POSITIVE-FIXNUM 6  
MOST-POSITIVE-LONG-FLOAT 6  
MOST-POSITIVE-SHORT-FLOAT 6  
MOST-POSITIVE-SINGLE-FLOAT 6  
MULTIPLE-VALUE-BIND 18  
MULTIPLE-VALUE-CALL 18  
MULTIPLE-VALUE-LIST 18  
MULTIPLE-VALUE-PROG1 21  
MULTIPLE-VALUE-SETQ 17  
MULTIPLE-VALUES-LIMIT 19  
NAME-CHAR 7  
NAMED 22  
NAMESTRING 43  
NBUTLAST 9  
NCONC 10, 24, 28  
NCONCING 24  
NEVER 25  
NEWLINE 7  
NEXT-METHOD-P 26  
NIL 2, 46  
NINTERSECTION 11  
LAMBDA 18  
LAMBDA-LIST-KEYWORDS 20  
LAMBDA-PARAMETERS-LIMIT 19  
LAST 9  
LCM 3  
LDB 6  
LDB-TEST 6  
LDIFF 9  
LEAST-NEGATIVE-DOUBLE-FLOAT 6  
LEAST-NEGATIVE-LONG-FLOAT 6  
LEAST-NEGATIVE-SHORT-FLOAT 6  
LEAST-NEGATIVE-NORMALIZED-DOUBLE-FLOAT 6  
LEAST-NEGATIVE-NORMALIZED-LONG-FLOAT 6  
LEAST-POSITIVE-DOUBLE-FLOAT 6  
LEAST-POSITIVE-LONG-FLOAT 6  
LEAST-POSITIVE-NORMALIZED-DOUBLE-FLOAT 6  
LEAST-POSITIVE-NORMALIZED-LONG-FLOAT 6  
LEAST-POSITIVE-SHORT-FLOAT 6  
LEAST-POSITIVE-NORMALIZED-SINGLE-FLOAT 6  
LEAST-POSITIVE-SINGLE-FLOAT 6  
LEAST-POSITIVE-SHORT-FLOAT 6  
LEAST-POSITIVE-NORMALIZED-SINGLE-FLOAT 6  
LEAST-POSITIVE-SINGLE-FLOAT 6  
LENGTH 13  
LET 18  
LET\* 18  
LISP-IMPLEMENTATION-TYPE 49  
LISP-IMPLEMENTATION-VERSION 49  
LIST 9, 28, 32  
LIST-ALL-PACKAGES 44  
LIST-LENGTH 9  
LIST\* 9  
LISTEN 41  
LISTP 8  
LOAD 47  
LOAD-LOGICAL-PATHNAME-TRANSLATIONS 43  
LOAD-TIME-VALUE 47  
LOCALLY 47  
LOG 3  
LOGAND 5  
LOGANDC1 5  
LOGANDC2 5  
LOGBITP 5  
LOGCOUNT 5  
LOGEQV 5  
LOGICAL-PATHNAME 32, 43  
LOGICAL-PATHNAME-TRANSLATIONS 43  
LOGIOR 5  
LOGNAND 5  
LOGNOR 5  
LOGNOT 5  
LOGGRC1 5  
LOGGRC2 5  
LOGTEST 5  
LOGXOR 5  
LONG-FLOAT 32, 35  
LONG-FLOAT-EPSILON 6  
LONG-FLOAT-NEGATIVE-EPSILON 6  
LONG-SITE-NAME 49  
LOOP 22  
LOOP-FINISH 25  
LOWER-CASE-P 7  
MACHINE-INSTANCE 49  
MACHINE-TYPE 49  
MACHINE-VERSION 49  
MACRO-FUNCTION 47  
MACROEXPAND 48  
MACROEXPAND-1 48  
MACROLET 20  
MAKE-ARRAY 11  
MAKE-BROADCAST-STREAM 41  
MAKE-CONCATENATED-STREAM 41  
MAKE-CONDITION 29  
MAKE-DISPATCH-MACRO-CHARACTER 35  
MAKE-ECHO-STREAM 41  
MAKE-HASH-TABLE 15  
MAKE-INSTANCE 25  
MAKE-INSTANCES-OBSOLETE 26  
MAKE-LIST 9  
MAKE-LOAD-FORM 47  
MAKE-LOAD-FORM-SAVING-SLOTS 47  
MAKE-METHOD 28  
MAKE-PACKAGE 44  
MAKE-PATHNAME 42  
MAKE-RANDOM-STATE 4  
MAKE-SEQUENCE 13  
MAKE-STRING 8  
MAKE-STRING-INPUT-STREAM 41  
MAKE-STRING-OUTPUT-STREAM 41  
MAKE-SYMBOL 45  
MAKE-SYNONYM-STREAM 41  
MAKE-TWO-WAY-STREAM 41  
MAKUNBOUND 17  
MAP 15  
MAP-INTO 15  
MAPC 10  
MAPCAN 10  
MAPCAR 10  
MAPCON 10  
MAPHASH 15  
MAPL 10  
MAPLIST 10  
MASK-FIELD 6  
MAX 4, 28  
MAXIMIZE 24  
MAXIMIZING 24  
MEMBER 9, 33  
MEMBER-IF 9  
MEMBER-IF-NOT 9  
MERGE 13  
MERGE-PATHNAMES 43  
METHOD 32  
METHOD-COMBINATION 32, 46  
METHOD-ERROR 27  
METHOD-QUALIFIERS 28  
MIN 4, 28  
MINIMIZE 24  
MINIMIZING 24  
MINUSP 3  
MISMATCH 13  
MOD 4, 33  
MOST-NEGATIVE-DOUBLE-FLOAT 6  
MOST-NEGATIVE-FIXNUM 6  
MOST-NEGATIVE-LONG-FLOAT 6  
MOST-NEGATIVE-SHORT-FLOAT 6  
MOST-NEGATIVE-SINGLE-FLOAT 6  
MOST-POSITIVE-DOUBLE-FLOAT 6  
MOST-POSITIVE-FIXNUM 6  
MOST-POSITIVE-LONG-FLOAT 6  
MOST-POSITIVE-SHORT-FLOAT 6  
MOST-POSITIVE-SINGLE-FLOAT 6  
MULTIPLE-VALUE-BIND 18  
MULTIPLE-VALUE-CALL 18  
MULTIPLE-VALUE-LIST 18  
MULTIPLE-VALUE-PROG1 21  
MULTIPLE-VALUE-SETQ 17  
MULTIPLE-VALUES-LIMIT 19  
NAME-CHAR 7  
NAMED 22  
NAMESTRING 43  
NBUTLAST 9  
NCONC 10, 24, 28  
NCONCING 24  
NEVER 25  
NEWLINE 7  
NEXT-METHOD-P 26  
NIL 2, 46  
NINTERSECTION 11

---

WHEN 21, 24	TABLE-ITERATOR 15	WITH- SIMPLE-RESTART 30	WRITE-STRING 36
WHILE 25	WITH-INPUT- FROM-STRING 42	WITH-SLOTS 26	WRITE-TO-STRING 37
WILD-PATHNAME-P 33	WITH-OPEN-FILE 42	WITH-STANDARD- IO-SYNTAX 34	
WITH 22	WITH-OPEN-STREAM 42	WRITE 37	Y-OR-N-P 34
WITH-ACCESSORS 26	WITH-OUTPUT- TO-STRING 42	WRITE-BYTE 36	YES-OR-NO-P 34
WITH-COMPILATION- UNIT 47	WITH-PACKAGE- ITERATOR 45	WRITE-CHAR 36	
WITH-CONDITION- RESTARTS 31		WRITE-LINE 36	
WITH-HASH-		WRITE-SEQUENCE 37	ZEROP 3



