

Quick Reference

cl

Common

lisp

Bert Burgemeister

Contents

1	Numbers	3	9.5	Control Flow . . .	19
1.1	Predicates	3	9.6	Iteration	20
1.2	Numeric Functns .	3	9.7	Loop Facility . . .	21
1.3	Logic Functions .	4	10	CLOS	23
1.4	Integer Functions .	5	10.1	Classes	23
1.5	Implementation- Dependent	6	10.2	Generic Functns .	25
2	Characters	6	10.3	Method Combi- nation Types . . .	26
3	Strings	7	11	Conditions and Errors	27
4	Conses	8	12	Types and Classes	29
4.1	Predicates	8	13	Input/Output	31
4.2	Lists	8	13.1	Predicates	31
4.3	Association Lists .	9	13.2	Reader	31
4.4	Trees	10	13.3	Character Syntax .	33
4.5	Sets	10	13.4	Printer	34
5	Arrays	10	13.5	Format	36
5.1	Predicates	10	13.6	Streams	38
5.2	Array Functions .	10	13.7	Paths and Files .	40
5.3	Vector Functions .	11	14	Packages and Symbols	41
6	Sequences	12	14.1	Predicates	41
6.1	Seq. Predicates . .	12	14.2	Packages	41
6.2	Seq. Functions . .	12	14.3	Symbols	43
7	Hash Tables	14	14.4	Std Packages . . .	43
8	Structures	15	15	Compiler	43
9	Control Structure	15	15.1	Predicates	43
9.1	Predicates	15	15.2	Compilation	43
9.2	Variables	16	15.3	REPL & Debug . . .	45
9.3	Functions	16	15.4	Declarations	46
9.4	Macros	18	16	External Environment	46

Typographic Conventions

name;	^{Fu} name;	^M name;	^{sO} name;	^{gF} name;	^{var} *name*;	^{co} name	
	▷ Symbol defined in Common Lisp; esp. function, macro, special operator, generic function, variable, constant.						
<i>them</i>	▷ Placeholder for actual code.						
me	▷ Literal text.						
[<i>foo</i> bar]	▷ Either one <i>foo</i> or nothing; defaults to bar .						
<i>foo</i> *; { <i>foo</i> }*	▷ Zero or more <i>foos</i> .						
<i>foo</i> ⁺ ; { <i>foo</i> } ⁺	▷ One or more <i>foos</i> .						
<i>foos</i>	▷ English plural denotes a list argument.						
{ <i>foo</i> <i>bar</i> <i>baz</i> };	$\begin{cases} \textit{foo} \\ \textit{bar} \\ \textit{baz} \end{cases}$	▷ Either <i>foo</i> , or <i>bar</i> , or <i>baz</i> .					
$\begin{cases} \textit{foo} \\ \textit{bar} \\ \textit{baz} \end{cases}$	▷ Anything from none to each of <i>foo</i> , <i>bar</i> , and <i>baz</i> .						
$\widehat{\textit{foo}}$	▷ Argument <i>foo</i> is not evaluated.						
$\widetilde{\textit{bar}}$	▷ Argument <i>bar</i> is possibly modified.						
<i>foo</i> ^{P*}	▷ <i>foo</i> * is evaluated as in ^{sO} progn ; see p. 19.						
$\underline{\textit{foo}; \textit{bar}; \textit{baz}}$	▷ Primary, secondary, and <i>n</i> th return value.						
T ; NIL	▷ t , or truth in general; and nil or () .						

1 Numbers

1.1 Predicates

$(\stackrel{\text{Fu}}{=} \text{number}^+)$
 $(/\stackrel{\text{Fu}}{=} \text{number}^+)$

▷ T if all *numbers*, or none, respectively, are equal in value.

$(\stackrel{\text{Fu}}{>} \text{number}^+)$
 $(\stackrel{\text{Fu}}{>=} \text{number}^+)$
 $(\stackrel{\text{Fu}}{<} \text{number}^+)$
 $(\stackrel{\text{Fu}}{<=} \text{number}^+)$

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

$(\stackrel{\text{Fu}}{\text{minusp}} a)$
 $(\stackrel{\text{Fu}}{\text{zerop}} a)$

▷ T if $a < 0$, $a = 0$, or $a > 0$, respectively.

$(\stackrel{\text{Fu}}{\text{plusp}} a)$

$(\stackrel{\text{Fu}}{\text{evenp}} \text{integer})$
 $(\stackrel{\text{Fu}}{\text{oddp}} \text{integer})$

▷ T if *integer* is even or odd, respectively.

$(\stackrel{\text{Fu}}{\text{numberp}} \text{foo})$
 $(\stackrel{\text{Fu}}{\text{realp}} \text{foo})$
 $(\stackrel{\text{Fu}}{\text{rationalp}} \text{foo})$
 $(\stackrel{\text{Fu}}{\text{floatp}} \text{foo})$

▷ T if *foo* is of indicated type.

$(\stackrel{\text{Fu}}{\text{integerp}} \text{foo})$
 $(\stackrel{\text{Fu}}{\text{complexp}} \text{foo})$
 $(\stackrel{\text{Fu}}{\text{random-state-p}} \text{foo})$

1.2 Numeric Functions

$(\stackrel{\text{Fu}}{+} a_{\square}^*)$
 $(\stackrel{\text{Fu}}{*} a_{\square}^*)$

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

$(\stackrel{\text{Fu}}{-} a b^*)$
 $(\stackrel{\text{Fu}}{/} a b^*)$

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

$(\stackrel{\text{Fu}}{1+} a)$
 $(\stackrel{\text{Fu}}{1-} a)$

▷ Return $\underline{a + 1}$ or $\underline{a - 1}$, respectively.

$(\left\{ \begin{array}{l} \text{M} \\ \text{M} \end{array} \right\} \text{incf})$
 $(\left\{ \begin{array}{l} \text{M} \\ \text{M} \end{array} \right\} \text{decf})$

$\widetilde{\text{place}} [\text{delta}_{\square}]$

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

$(\stackrel{\text{Fu}}{\text{exp}} p)$
 $(\stackrel{\text{Fu}}{\text{expt}} b p)$

▷ Return $\underline{e^p}$ or $\underline{b^p}$, respectively.

$(\stackrel{\text{Fu}}{\text{log}} a [b])$

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

$(\stackrel{\text{Fu}}{\text{sqrt}} n)$
 $(\stackrel{\text{Fu}}{\text{isqrt}} n)$

▷ $\underline{\sqrt{n}}$ in complex or natural numbers, respectively.

$(\stackrel{\text{Fu}}{\text{lcm}} \text{integer}^*_{\square})$
 $(\stackrel{\text{Fu}}{\text{gcd}} \text{integer}^*)$

▷ Least common multiple or greatest common denominator, respectively, of *integers*. (**gcd**) returns 0.

$\overset{\text{Co}}{\text{pi}}$

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

$(\stackrel{\text{Fu}}{\text{sin}} a)$
 $(\stackrel{\text{Fu}}{\text{cos}} a)$
 $(\stackrel{\text{Fu}}{\text{tan}} a)$

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

$(\stackrel{\text{Fu}}{\text{asin}} a)$
 $(\stackrel{\text{Fu}}{\text{acos}} a)$

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

$(\stackrel{\text{Fu}}{\text{atan}} a [b_{\square}])$

▷ $\underline{\arctan \frac{a}{b}}$ in radians.

$(\stackrel{\text{Fu}}{\text{sinh}} a)$
 $(\stackrel{\text{Fu}}{\text{cosh}} a)$
 $(\stackrel{\text{Fu}}{\text{tanh}} a)$

▷ $\underline{\sinh a}$, $\underline{\cosh a}$, or $\underline{\tanh a}$, respectively.

- (^{Fu}**asinh** *a*)
(^{Fu}**acosh** *a*) ▷ asinh *a*, acosh *a*, or atanh *a*, respectively.
(^{Fu}**atanh** *a*)
- (^{Fu}**cis** *a*) ▷ Return $e^{i a} = \cos a + i \sin a$.
- (^{Fu}**conjugate** *a*) ▷ Return complex conjugate of *a*.
- (^{Fu}**max** *num*⁺)
(^{Fu}**min** *num*⁺) ▷ Greatest or least, respectively, of *nums*.
- (^{Fu}**round** | ^{Fu}**fround**)
(^{Fu}**floor** | ^{Fu}**ffloor**)
(^{Fu}**ceiling** | ^{Fu}**fceiling**)
(^{Fu}**truncate** | ^{Fu}**ftuncate**)
) *n* [_{int}])
▷ Return as **integer** or **float**, respectively, n/d rounded, or rounded towards $-\infty$, $+\infty$, or 0, respectively; and remainder.
- (^{Fu}**mod**)
(^{Fu}**rem**) *n* *d*)
▷ Same as **floor** or **truncate**, respectively, but return remainder only.
- (^{Fu}**random** *limit* [*state* ^{var}***random-state***])
▷ Return non-negative random number less than *limit*, and of the same type.
- (^{Fu}**make-random-state** [*state* | **NIL** | **T**] [_{int}])
▷ Copy of **random-state** object *state* or of the current random state; or a randomly initialized fresh random state.
- ^{var}***random-state*** ▷ Current random state.
- (^{Fu}**float-sign** *num-a* [*num-b* _{int}]) ▷ *num-b* with *num-a*'s sign.
- (^{Fu}**signum** *n*)
▷ Number of magnitude 1 representing sign or phase of *n*.
- (^{Fu}**numerator** *rational*)
(^{Fu}**denominator** *rational*)
▷ Numerator or denominator, respectively, of *rational*'s canonical form.
- (^{Fu}**realpart** *number*)
(^{Fu}**imagpart** *number*)
▷ Real part or imaginary part, respectively, of *number*.
- (^{Fu}**complex** *real* [*imag* _{int}]) ▷ Make a complex number.
- (^{Fu}**phase** *number*) ▷ Angle of *number*'s polar representation.
- (^{Fu}**abs** *n*) ▷ Return |*n*|.
- (^{Fu}**rational** *real*)
(^{Fu}**rationalize** *real*)
▷ Convert *real* to rational. Assume complete/limited accuracy for *real*.
- (^{Fu}**float** *real* [*prototype* _{0.0F0}])
▷ Convert *real* into float with type of *prototype*.

1.3 Logic Functions

Negative integers are used in two's complement representation.

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

boole-eqv ^{co}	▷ <u>$int-a \equiv int-b$</u> .
boole-and ^{co}	▷ <u>$int-a \wedge int-b$</u> .
boole-andc1 ^{co}	▷ <u>$\neg int-a \wedge int-b$</u> .
boole-andc2 ^{co}	▷ <u>$int-a \wedge \neg int-b$</u> .
boole-nand ^{co}	▷ <u>$\neg(int-a \wedge int-b)$</u> .
boole-ior ^{co}	▷ <u>$int-a \vee int-b$</u> .
boole-orc1 ^{co}	▷ <u>$\neg int-a \vee int-b$</u> .
boole-orc2 ^{co}	▷ <u>$int-a \vee \neg int-b$</u> .
boole-xor ^{co}	▷ <u>$\neg(int-a \equiv int-b)$</u> .
boole-nor ^{co}	▷ <u>$\neg(int-a \vee int-b)$</u> .

(**lognot** ^{Fu} *integer*) ▷ $\neg integer$.

(**logeqv** ^{Fu} *integer**)

(**logand** ^{Fu} *integer**)

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

(**logandc1** ^{Fu} *int-a int-b*) ▷ $\neg int-a \wedge int-b$.

(**logandc2** ^{Fu} *int-a int-b*) ▷ $int-a \wedge \neg int-b$.

(**lognand** ^{Fu} *int-a int-b*) ▷ $\neg(int-a \wedge int-b)$.

(**logxor** ^{Fu} *integer**)

(**logior** ^{Fu} *integer**)

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

(**logorc1** ^{Fu} *int-a int-b*) ▷ $\neg int-a \vee int-b$.

(**logorc2** ^{Fu} *int-a int-b*) ▷ $int-a \vee \neg int-b$.

(**lognor** ^{Fu} *int-a int-b*) ▷ $\neg(int-a \vee int-b)$.

(**logbitp** ^{Fu} *i integer*)

▷ T if zero-indexed *i*th bit of *integer* is set.

(**logtest** ^{Fu} *int-a int-b*)

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

(**logcount** ^{Fu} *int*)

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

1.4 Integer Functions

(**integer-length** ^{Fu} *integer*)

▷ Number of bits necessary to represent *integer*.

(**ldb-test** ^{Fu} *byte-spec integer*)

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

(**ash** ^{Fu} *integer count*)

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

(**ldb** ^{Fu} *byte-spec integer*)

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

(**deposit-field** ^{Fu} *int-a byte-spec int-b*)

(**dppb** ^{Fu})

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

(**mask-field** ^{Fu} *byte-spec integer*)

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

(**byte** ^{Fu} *size position*)

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

(**byte-size** ^{Fu} *byte-spec*)

(**byte-position** ^{Fu} *byte-spec*)

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

1.5 Implementation-Dependent

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

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

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

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

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

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

$\text{(decode-float } n)$
 $\text{(integer-decode-float } n)$

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

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

$\text{(float-radix } n)$
 $\text{(float-digits } n)$
 $\text{(float-precision } n)$

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

$\text{(upgraded-complex-part-type } foo \ [environment_{\text{NIL}}])$

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

2 Characters

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

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

$\text{(graphic-char-p } character)$
 $\text{(alpha-char-p } character)$
 $\text{(alphanumericp } character)$

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

$\text{(upper-case-p } character)$
 $\text{(lower-case-p } character)$
 $\text{(both-case-p } character)$

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

$\text{(digit-char-p } character \ [radix_{10}])$

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

$\text{(char= } character^+)$
 $\text{(char/= } character^+)$

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

$\text{(char-equal } character^+)$
 $\text{(char-not-equal } character^+)$

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

$\text{(char> } character^+)$
 $\text{(char>= } character^+)$
 $\text{(char< } character^+)$
 $\text{(char<= } character^+)$

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

^{Fu}(**char-greaterp** *character*⁺)

^{Fu}(**char-not-lessp** *character*⁺)

^{Fu}(**char-lessp** *character*⁺)

^{Fu}(**char-not-greaterp** *character*⁺)

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

^{Fu}(**char-upcase** *character*)

^{Fu}(**char-downcase** *character*)

▷ Return corresponding uppercase/lowercase character, respectively.

^{Fu}(**digit-char** *i* [*radix*₁₀]) ▷ Character representing digit *i*.

^{Fu}(**char-name** *character*) ▷ *character*'s name if any, or NIL.

^{Fu}(**name-char** *foo*) ▷ Character named *foo* if any, or NIL.

^{Fu}(**char-int** *character*)

^{Fu}(**char-code** *character*) ▷ Code of *character*.

^{Fu}(**code-char** *code*) ▷ Character with *code*.

^{So}**char-code-limit** ▷ Upper bound of (^{Fu}**char-code** *char*); ≥ 96.

^{Fu}(**character** *c*) ▷ Return #\c.

3 Strings

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

^{Fu}(**stringp** *foo*)

^{Fu}(**simple-string-p** *foo*) ▷ T if *foo* is of indicated type.

^{Fu}(**string=** *foo bar* {**string-equal**}) $\left\{ \begin{array}{l} \text{:start1 } start\text{-}foo_{\square} \\ \text{:start2 } start\text{-}bar_{\square} \\ \text{:end1 } end\text{-}foo_{\text{NIL}} \\ \text{:end2 } end\text{-}bar_{\text{NIL}} \end{array} \right\}$

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

^{Fu}(**string**{**/=** |**-not-equal**}) $\left\{ \begin{array}{l} \text{:start1 } start\text{-}foo_{\square} \\ \text{:start2 } start\text{-}bar_{\square} \\ \text{:end1 } end\text{-}foo_{\text{NIL}} \\ \text{:end2 } end\text{-}bar_{\text{NIL}} \end{array} \right\}$

^{Fu}(**string**{**>** |**-greaterp**}) $\left\{ \begin{array}{l} \text{:start1 } start\text{-}foo_{\square} \\ \text{:start2 } start\text{-}bar_{\square} \\ \text{:end1 } end\text{-}foo_{\text{NIL}} \\ \text{:end2 } end\text{-}bar_{\text{NIL}} \end{array} \right\}$

^{Fu}(**string**{**>=** |**-not-lessp**})

^{Fu}(**string**{**<** |**-lessp**})

^{Fu}(**string**{**<=** |**-not-greaterp**})

▷ 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.

^{Fu}(**make-string** *size* {**initial-element** *char* | **element-type** *type*_{character}})

▷ Return string of length *size*.

^{Fu}(**string** *x*)

^{Fu}(**string-capitalize** *x* {**string-upcase** | **string-downcase**}) $\left\{ \begin{array}{l} \text{:start } start_{\square} \\ \text{:end } end_{\text{NIL}} \end{array} \right\}$

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

^{Fu}(**nstring-capitalize** *string* {**nstring-upcase** | **nstring-downcase**}) $\left\{ \begin{array}{l} \text{:start } start_{\square} \\ \text{:end } end_{\text{NIL}} \end{array} \right\}$

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

^{Fu}(**string-trim** *char-bag string*)

^{Fu}(**string-left-trim** *char-bag string*)

^{Fu}(**string-right-trim** *char-bag string*)

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

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

(^{Fu}**parse-integer** *string* $\left\{ \begin{array}{l} \text{:start } start_{\underline{0}} \\ \text{:end } end_{\underline{NIL}} \\ \text{:radix } int_{\underline{10}} \\ \text{:junk-allowed } bool_{\underline{NIL}} \end{array} \right\}$)
 ▷ Return integer parsed from *string* and index of parse end.

4 Conses

4.1 Predicates

(^{Fu}**consp** *foo*)
 (^{Fu}**listp** *foo*)
 ▷ Return T if *foo* is of indicated type.

(^{Fu}**endp** *list*)
 (^{Fu}**null** *foo*)
 ▷ Return T if *list/foo* is NIL.

(^{Fu}**atom** *foo*)
 ▷ Return T if *foo* is not a **cons**.

(^{Fu}**tailp** *foo* *list*)
 ▷ Return T if *foo* is a tail of *list*.

(^{Fu}**member** *foo* *list* $\left\{ \begin{array}{l} \text{:test } function_{\underline{\#='eql}} \\ \text{:test-not } function \\ \text{:key } function \end{array} \right\}$)
 ▷ Return tail of *list* starting with its first element matching *foo*. Return NIL if there is no such element.

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

(^{Fu}**subsetp** *list-a* *list-b* $\left\{ \begin{array}{l} \text{:test } function_{\underline{\#='eql}} \\ \text{:test-not } function \\ \text{:key } function \end{array} \right\}$)
 ▷ Return T if *list-a* is a subset of *list-b*.

4.2 Lists

(^{Fu}**cons** *foo* *bar*)
 ▷ Return new cons (*foo . bar*).

(^{Fu}**list** *foo**)
 ▷ Return list of *foos*.

(^{Fu}**list*** *foo**)
 ▷ Return list of *foos* with last *foo* becoming cdr of last cons. Return *foo* if only one *foo* given.

(^{Fu}**make-list** *num* [*:initial-element* *foo*_{NIL}])
 ▷ New list with *num* elements set to *foo*.

(^{Fu}**list-length** *list*)
 ▷ Length of *list*; NIL for circular *list*.

(^{Fu}**car** *list*)
 ▷ Car of *list* or NIL if *list* is NIL. **setfable**.

(^{Fu}**cdr** *list*)
 (^{Fu}**rest** *list*)
 ▷ Cdr of *list* or NIL if *list* is NIL. **setfable**.

(^{Fu}**nthcdr** *n* *list*)
 ▷ Return tail of *list* after calling **cdr** *n* times.

($\{ \text{first}^{\text{Fu}} | \text{second}^{\text{Fu}} | \text{third}^{\text{Fu}} | \text{fourth}^{\text{Fu}} | \text{fifth}^{\text{Fu}} | \text{sixth}^{\text{Fu}} | \dots | \text{ninth}^{\text{Fu}} | \text{tenth}^{\text{Fu}} \}$ *list*)
 ▷ Return nth element of *list* if any, or NIL otherwise. **setfable**.

(^{Fu}**nth** *n* *list*)
 ▷ Zero-indexed nth element of *list*. **setfable**.

(^{Fu}**cXr** *list*)
 ▷ With *X* being one to four **as** and **ds** representing ^{Fu}**cars** and ^{Fu}**cdrs**, e.g. (^{Fu}**cadr** *bar*) is equivalent to (^{Fu}**car** (^{Fu}**cdr** *bar*)). **setfable**.

(^{Fu}**last** *list* [*num*_{1}])
 ▷ Return list of last num conses of *list*.

($\left\{ \begin{array}{l} \text{butlast} \\ \text{nbutlast} \end{array} \right\}^{\text{Fu}}$ *list*) [*num*_□] ▷ *list* excluding last *num* conses.

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

(ldiff^{Fu} *list* *foo*)
 ▷ If *foo* is a tail of *list*, return preceding part of *list*. Otherwise return *list*.

($\text{adjoin}^{\text{Fu}}$ *foo* *list* $\left\{ \left\{ \begin{array}{l} \text{:test } \text{function}_{\#'\text{eql}} \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\} \right\}$)
 ▷ Return *list* if *foo* is already member of *list*. If not, return (cons^{Fu} *foo* *list*).

(pop^{M} $\widetilde{\text{place}}$) ▷ Set *place* to (cdr^{Fu} *place*), return (car^{Fu} *place*).

(push^{M} *foo* $\widetilde{\text{place}}$) ▷ Set *place* to (cons^{Fu} *foo* *place*).

($\text{pushnew}^{\text{M}}$ *foo* $\widetilde{\text{place}}$ $\left\{ \left\{ \begin{array}{l} \text{:test } \text{function}_{\#'\text{eql}} \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\} \right\}$)
 ▷ Set *place* to ($\text{adjoin}^{\text{Fu}}$ *foo* *place*).

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

($\text{revappend}^{\text{Fu}}$ *list* *foo*)
 ($\text{reconc}^{\text{Fu}}$ *list* *foo*)
 ▷ Return concatenated list after reversing order in *list*.

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

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

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

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

4.3 Association Lists

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

(acons^{Fu} *key* *value* *alist*)
 ▷ Return *alist* with a (*key* . *value*) pair added.

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

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

4.4 Trees

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

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

$(\overset{\text{Fu}}{\text{subst}} \text{ new old tree } \left\{ \begin{array}{l} \text{:test } \text{function}_{\text{\#'eq}} \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\})$

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

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

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

$(\overset{\text{Fu}}{\text{sublis}} \text{ association-list tree } \left\{ \begin{array}{l} \text{:test } \text{function}_{\text{\#'eq}} \\ \text{:test-not } \text{function} \\ \text{:key } \text{function} \end{array} \right\})$

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

$(\overset{\text{Fu}}{\text{copy-tree}} \text{ tree})$ ▷ Copy of *tree* with same shape and leaves.

4.5 Sets

$(\overset{\text{Fu}}{\text{intersection}} \text{ } \left. \begin{array}{l} \text{set-difference} \\ \text{union} \\ \text{set-exclusive-or} \end{array} \right\} a b$
 $(\overset{\text{Fu}}{\text{nintersection}} \text{ } \left. \begin{array}{l} \text{nset-difference} \\ \text{nunion} \\ \text{nset-exclusive-or} \end{array} \right\} \tilde{a} b$
 $(\overset{\text{Fu}}{\text{nintersection}} \text{ } \left. \begin{array}{l} \text{nset-difference} \\ \text{nunion} \\ \text{nset-exclusive-or} \end{array} \right\} \tilde{a} \tilde{b}$

▷ 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

$(\overset{\text{Fu}}{\text{array}} \text{ foo})$

$(\overset{\text{Fu}}{\text{vectorp}} \text{ foo})$

$(\overset{\text{Eu}}{\text{simple-vector-p}} \text{ foo})$

▷ T if *foo* is of indicated type.

$(\overset{\text{Fu}}{\text{bit-vector-p}} \text{ foo})$

$(\overset{\text{Eu}}{\text{simple-bit-vector-p}} \text{ foo})$

$(\overset{\text{Fu}}{\text{adjustable-array-p}} \text{ array})$

$(\overset{\text{Fu}}{\text{array-has-fill-pointer-p}} \text{ array})$

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

$(\overset{\text{Fu}}{\text{array-in-bounds-p}} \text{ array } [\text{subscripts}])$

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

5.2 Array Functions

$(\overset{\text{Fu}}{\text{make-array}} \text{ dimension-sizes } [\text{:adjustable } \text{bool}_{\text{\#T}}])$
 $(\overset{\text{Fu}}{\text{adjust-array}} \text{ array } \text{dimension-sizes}$

$\left. \begin{array}{l} \text{:element-type } \text{type}_{\text{\#}} \\ \text{:fill-pointer } \{ \text{num} | \text{bool} \}_{\text{\#NIL}} \\ \text{:initial-element } \text{obj} \\ \text{:initial-contents } \text{sequence} \\ \text{:displaced-to } \text{array}_{\text{\#NIL}} [\text{:displaced-index-offset } \text{i}_{\text{\#0}}] \end{array} \right\}$

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

$(\overset{\text{Fu}}{\text{aref}} \text{ array } [\text{subscripts}])$

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

$(\overset{\text{Fu}}{\text{row-major-aref}} \text{ array } \text{i})$

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

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

^{Fu}(**array-dimensions** *array*)
 ▷ List containing the lengths of *array*'s dimensions.

^{Fu}(**array-dimension** *array* *i*)
 ▷ Length of *i*th dimension of *array*.

^{Fu}(**array-total-size** *array*) ▷ Number of elements in *array*.

^{Fu}(**array-rank** *array*) ▷ Number of dimensions of *array*.

^{Fu}(**array-displacement** *array*) ▷ Target array and offset.₂

^{Fu}(**bit** *bit-array* [*subscripts*])

^{Fu}(**sbit** *simple-bit-array* [*subscripts*])
 ▷ Return element of *bit-array* or of *simple-bit-array*. **setf**able.

^{Fu}(**bit-not** *bit-array* [*result-bit-array*_{NIL}])
 ▷ Return result of bitwise negation of *bit-array*. If *result-bit-array* is T, put result in *bit-array*; if it is NIL, make a new array for result.

^{Fu}

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

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

^{co}**array-rank-limit** ▷ Upper bound of array rank; ≥ 8 .

^{co}**array-dimension-limit**
 ▷ Upper bound of an array dimension; ≥ 1024 .

^{co}**array-total-size-limit** ▷ Upper bound of array size; ≥ 1024 .

5.3 Vector Functions

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

^{Fu}(**vector** *foo**) ▷ Return fresh simple vector of *foos*.

^{Fu}(**svref** *vector* *i*) ▷ Return element *i* of simple *vector*. **setf**able.

^{Fu}(**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.

^{Fu}(**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.

^{Fu}(**vector-pop** *vector*)
 ▷ Return element of *vector* its fillpointer points to after decrementation.

^{Fu}(**fill-pointer** *vector*) ▷ Fill pointer of *vector*. **setf**able.

6 Sequences

6.1 Sequence Predicates

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

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

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

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

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

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

6.2 Sequence Functions

$\left(\begin{array}{l} \text{Fu} \\ \text{make-sequence} \end{array} \right) sequence-type\ size\ [\text{:initial-element}\ foo]$

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

$\left(\begin{array}{l} \text{Fu} \\ \text{concatenate} \end{array} \right) type\ sequence^*$

▷ Return concatenated sequence of *type*.

$\left(\begin{array}{l} \text{Fu} \\ \text{merge} \end{array} \right) type\ \widetilde{sequence-a}\ \widetilde{sequence-b}\ test\ [\text{:key}\ function_{\text{NIL}}]$

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

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

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

$\left(\begin{array}{l} \text{Fu} \\ \text{length} \end{array} \right) sequence$

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

$\left(\begin{array}{l} \text{Fu} \\ \text{count} \end{array} \right) foo\ sequence\ \left\{ \begin{array}{l} \text{:from-end}\ bool_{\text{NIL}} \\ \left\{ \begin{array}{l} \text{:test}\ function_{\neq eq} \\ \text{:test-not}\ function \end{array} \right\} \\ \text{:start}\ start_{\text{0}} \\ \text{:end}\ end_{\text{NIL}} \\ \text{:key}\ function \end{array} \right\}$

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

$\left(\begin{array}{l} \text{Fu} \\ \text{count-if} \\ \text{Fu} \\ \text{count-if-not} \end{array} \right) test\ sequence\ \left\{ \begin{array}{l} \text{:from-end}\ bool_{\text{NIL}} \\ \text{:start}\ start_{\text{0}} \\ \text{:end}\ end_{\text{NIL}} \\ \text{:key}\ function \end{array} \right\}$

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

$\left(\begin{array}{l} \text{Fu} \\ \text{elt} \end{array} \right) sequence\ index$

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

$\left(\begin{array}{l} \text{Fu} \\ \text{subseq} \end{array} \right) sequence\ start\ [end_{\text{NIL}}]$

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

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

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

$\left(\begin{array}{l} \text{Fu} \\ \text{reverse} \end{array} \right) sequence$

$\left(\begin{array}{l} \text{Fu} \\ \text{nreverse} \end{array} \right) \widetilde{sequence}$

▷ Return sequence in reverse order.

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

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

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

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

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

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

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

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

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

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

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

▷ Make copy of sequence without duplicates.

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

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

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

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

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

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

(^{Fu}**map** *type function sequence*⁺)
 ▷ Apply *function* successively to corresponding elements of the *sequences*. Return values as a sequence of *type*. If *type* is NIL, return NIL.

(^{Fu}**map-into** *result-sequence function sequence*^{*})
 ▷ Store into result-sequence successively values of *function* applied to corresponding elements of the *sequences*.

(^{Fu}**reduce** *function sequence* $\left\{ \begin{array}{l} \text{:initial-value } foo_{\text{NIL}} \\ \text{:from-end } bool_{\text{NIL}} \\ \text{:start } start_{\text{0}} \\ \text{:end } end_{\text{NIL}} \\ \text{:key function} \end{array} \right\}$)

▷ 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.

(^{Fu}**copy-seq** *sequence*)
 ▷ Copy of *sequence* with shared elements.

7 Hash Tables

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

(^{Fu}**hash-table-p** *foo*) ▷ Return T if *foo* is of type **hash-table**.

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

▷ Make a hash table.

(^{Fu}**gethash** *key hash-table* [*default* NIL])
 ▷ Return object with *key* if any or default otherwise; and T if found, NIL otherwise. **setfable**.

(^{Fu}**hash-table-count** *hash-table*)
 ▷ Number of entries in *hash-table*.

(^{Fu}**remhash** *key hash-table*)
 ▷ Remove from *hash-table* entry with *key* and return T if it existed. Return NIL otherwise.

(^{Fu}**clrhash** *hash-table*) ▷ Empty hash-table.

(^{Fu}**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** $\widehat{decl^*}$)^{*} *form*^{P*})
 ▷ Return values of forms. In *forms*, invocations of (*foo*) return: T if an entry is returned; its key; its value.

(^{Fu}**hash-table-test** *hash-table*)
 ▷ Test function used in *hash-table*.

(^{Fu}**hash-table-size** *hash-table*)

(^{Fu}**hash-table-rehash-size** *hash-table*)

(^{Fu}**hash-table-rehash-threshold** *hash-table*)

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

(^{Fu}**sxhash** *foo*)

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

8 Structures

(^Mdefstruct *foo*)

```

{
  (:conc-name [slot-prefix foo-])
  (:constructor [maker MAKE-foo] [(ord-λ*)])
  (:copier [copier COPY-foo])
  (:include struct {
    (slot [init {
      (:type sl-type)
      (:read-only b)
    }])
  })
  (:type {
    list
    vector
    (vector type)
  })
  (:named {
    (:initial-offset n)
  })
  (:print-object [o-printer])
  (:print-function [f-printer])
  (:predicate [p-name foo-P])
}
[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 p. 16) is given, by (*maker arg** **{:key value}***). In the latter case, *args* and **:keys** correspond to the positional and keyword parameters defined in *ord-λ* whose *vars* in turn correspond to *slots*. **:print-object**/**:print-function** generate a **print-object** method for an instance *bar* of *foo* calling (*o-printer bar stream*) or (*f-printer bar stream print-level*), respectively. If **:type** without **:named** is given, no *foo-P* is created.

(^{Fu}copy-structure *structure*)

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

9 Control Structure

9.1 Predicates

(^{Fu}eq *foo bar*) ▷ T if *foo* and *bar* are identical.

(^{Fu}equal *foo bar*)
▷ T if *foo* and *bar* are identical, or the same **character**, or **numbers** of the same type and value.

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

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

(^{Fu}not *foo*) ▷ T if *foo* is NIL; NIL otherwise.

(^{Fu}boundp *symbol*) ▷ T if *symbol* is a special variable.

(^{Fu}constantp *foo* [*environment* NIL])
▷ T if *foo* is a constant form.

(^{Fu}functionp *foo*) ▷ T if *foo* is of type **function**.

(^{Fu}**fboundp** $\left\{ \begin{array}{l} \widehat{foo} \\ (\text{setf } \widehat{foo}) \end{array} \right\}$) \triangleright T if *foo* is a global function or macro.

9.2 Variables

($\left\{ \begin{array}{l} \text{M} \\ \text{M} \end{array} \right\}$ **defconstant** / **defparameter**) \widehat{foo} *form* [\widehat{doc}])
 \triangleright Assign value of *form* to global constant/dynamic variable *foo*.

(^M**defvar** \widehat{foo} [*form* [\widehat{doc}]])
 \triangleright Unless bound already, assign value of *form* to dynamic variable *foo*.

($\left\{ \begin{array}{l} \text{M} \\ \text{M} \end{array} \right\}$ **setf** / **psetf**) $\{ \textit{place form} \}^*$
 \triangleright Set *places* to primary values of *forms*. Return values of last *form*/NIL; work sequentially/in parallel, respectively.

($\left\{ \begin{array}{l} \text{SO} \\ \text{M} \end{array} \right\}$ **setq** / **psetq**) $\{ \textit{symbol form} \}^*$
 \triangleright Set *symbols* to primary values of *forms*. Return value of last *form*/NIL; work sequentially/in parallel, respectively.

(^{Fu}**set** $\widetilde{\textit{symbol form}}$)
 \triangleright Set *symbol*'s value cell to *foo*. Deprecated.

(^M**multiple-value-setq** *vars form*)
 \triangleright Set elements of *vars* to the values of *form*. Return *form*'s primary value.

(^M**shiftf** $\widetilde{\textit{place}^+ \textit{foo}}$)
 \triangleright Store value of *foo* in rightmost *place* shifting values of *places* left, returning first *place*.

(^M**rotatef** $\widetilde{\textit{place}^*}$)
 \triangleright Rotate values of *places* left, old first becoming new last *place*'s value. Return NIL.

(^{Fu}**makunbound** $\widetilde{\textit{foo}}$) \triangleright Delete special variable *foo* if any.

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

(^{Fu}**get-properties** *property-list keys*)
 \triangleright 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, $\frac{3}{2}$ NIL, and $\frac{3}{3}$ NIL if there was no matching key in *property-list*.

(^{Fu}**remprop** $\widetilde{\textit{symbol key}}$)
 (^M**remf** *place key*)
 \triangleright Remove first entry *key* from property list stored in *symbol*/in *place*, respectively. Return T if *key* was there, or NIL otherwise.

9.3 Functions

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

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

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

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

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

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

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

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

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

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

▷ Values of function called with args and the list elements of args. setfable if function is one of arf^{Fu}, bit^{Fu}, and sbit^{Fu}.

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

($\text{SO} \text{multiple-value-call } \text{function } \text{form}^*$)

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

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

($\text{Fu} \text{values } \text{foo}^*$)

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

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

($\text{M} \text{nth-value } n \text{ form}$)

▷ Zero-indexed nth return value of form.

($\text{Fu} \text{complement } \text{function}$)

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

($\text{Fu} \text{constantly } \text{foo}$)

▷ Function of any number of arguments returning foo.

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

($\text{Fu} \text{function-lambda-expression } \text{function}$)

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

($\text{Fu} \text{fdefinition } \left\{ \begin{array}{l} \text{foo} \\ (\text{setf } \text{foo}) \end{array} \right\}$)

▷ Definition of global function foo. setfable.

($\text{Fu} \text{fmakunbound } \text{foo}$)

▷ Remove global function or macro definition foo.

$\text{CO} \text{call-arguments-limit}$

$\text{CO} \text{lambda-parameters-limit}$

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

$\text{CO} \text{multiple-values-limit}$

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

9.4 Macros

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

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

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

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

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

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

or

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

One toplevel $[E]$ may be replaced by **&environment** *var*. *supplied-p* is T if there is a corresponding argument. *init* forms can refer to any *init* and *supplied-p* to their left.

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

▷ 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*.

$$(\text{define-symbol-macro } \textit{foo} \textit{form})$$

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

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

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

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

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

$$(\text{defsetf } \widehat{\textit{function}})$$

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

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

$$[\&optional \left\{ \begin{array}{l} \textit{var} \\ (\textit{var} [init_{\text{NIL}} [\textit{supplied-p}]]]) \end{array} \right\}^*] [\&rest \textit{var}]$$

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

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

▷ 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 **block** named *function*.

$$(\text{define-setf-expander } \textit{function} (\textit{macro-}\lambda^*) (\text{declare } \widehat{\textit{decl}}^*)^* [\widehat{\textit{doc}}] \textit{form}^{\text{R}^*})$$

▷ 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 **get-setf-expansion** where the elements of macro lambda list *macro-λ** are bound to corresponding *args*. *forms* are enclosed in an implicit **block** named *function*.

(^{Fu}**get-setf-expansion** *place* [*environment*_{NIL}])

▷ Return lists of temporary variables *arg-vars* and of corresponding *args* as given with *place*, list *newval-vars* with temporary variables corresponding to the new values, and *set-form* and *get-form* specifying in terms of *arg-vars* and *newval-vars* how to **setf** and how to read *place*.

(^M**define-modify-macro** *foo* ([&**optional**

{*var* (*var* [*init*_{NIL} [*supplied-p*])}]*) [**&rest** *var*]) *function* [*doc*])

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

^{co}**lambda-list-keywords**

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

&whole *var*

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

&optional *var**

▷ Bind *vars* to corresponding arguments if any.

{**&rest**|**&body**} *var*

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

&key *var**

▷ Bind *vars* to corresponding keyword arguments.

&allow-other-keys

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

&environment *var*

▷ Bind *var* to the lexical compilation environment.

&aux *var** ▷ Bind *vars* as in **let***^{so}.

9.5 Control Flow

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

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

(^M**cond** (*test* *then* ^{P*}*test**)*)

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

(^M**when** | ^M**unless**) *test* *foo*^{P*})

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

(^M**case** *test* ({*key** } *foo*^{P*})* [(^T**otherwise**) *bar*^{P*}]_{NIL}])

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

(^M**ecase** | ^M**ccase**) *test* ({*key** } *foo*^{P*})*)

▷ Return the values of the first *foo** one of whose *keys* is **eql** *test*. Signal non-correctable/correctable **type-error** and return NIL if there is no matching *key*.

(^M**and** *form**_{NIL})

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

(^M**or** *form**_{NIL})

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

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

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

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

(^M**prog1** *form-r form**)

(^M**prog2** *form-a form-r form**)

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

(^{SO}**let** ^{SO}**let***) ({ { *name* (*name* [*value*_{NIL}]) } }) (**declare** \widehat{decl}^*) * *form*^{P*})

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

(^M**prog** ^M**prog***) ({ { *name* (*name* [*value*_{NIL}]) } }) (**declare** \widehat{decl}^*) * { *tag* } * *form*)

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

(^{sO}**progv** *symbols values form*^{P*})

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

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

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

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

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

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

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

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

▷ Evaluate *forms* in a lexical environment, and return their values unless interrupted by ^{sO}**return-from**.

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

(^M**return** [*result*_{NIL}])

▷ Have nearest enclosing ^{sP}**block** named *foo*/named NIL, respectively, return with values of *result*.

(^{sO}**tagbody** { *tag* | *form* } *)

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

(^{sO}**go** *tag*)

▷ Within the innermost possible enclosing ^{sO}**tagbody**, jump to a tag **eq** *tag*.

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

▷ Evaluate *forms* and return their values unless interrupted by ^{sO}**throw**.

(^{sO}**throw** *tag form*)

▷ Have the nearest dynamically enclosing ^{sO}**catch** with a tag ^{Fu}**eq** *tag* return with the values of *form*.

(^{Fu}**sleep** *n*) ▷ Wait *n* seconds, return NIL.

9.6 Iteration

(^M**do** ^M**do***) ({ { *var* (*var* [*start* [*step*]]) } }) (*stop result*^{P*}) (**declare** \widehat{decl}^*) * { *tag* } * *form*)

▷ Evaluate **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*. Implicitly, the whole form is a **block** named NIL.

- (^M**dotimes** (*var* *i* [*result*_{NIL}]) (**declare** \widehat{decl}^*)^{*} $\{\widehat{tag}|form\}^*$)
 ▷ Evaluate **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 **block** named NIL.
- (^M**dolist** (*var* *list* [*result*_{NIL}]) (**declare** \widehat{decl}^*)^{*} $\{\widehat{tag}|form\}^*$)
 ▷ Evaluate **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 **block** named NIL.

9.7 Loop Facility

- (^M**loop** *form*^{*})
 ▷ **Simple Loop.** If *forms* do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit **block** named NIL.

- (^M**loop** *clause*^{*})
 ▷ **Loop Facility.** For Loop Facility keywords see below and Figure 1.

named *n*_{NIL} ▷ Give ^M**loop**'s implicit **block** a name.

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

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

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

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

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

▷ Begin of iteration control clauses. Initialize and step (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel. Destructuring type specifier *d-type* as with **with**.

$\left\{ \text{upfrom|from|downfrom} \right\}$ *start*
 ▷ Start stepping with *start*

$\left\{ \text{upto|downto|to|below|above} \right\}$ *form*
 ▷ Specify *form* as the end value for stepping.

$\left\{ \text{in|on} \right\}$ *list*
 ▷ Bind *var* to successive elements/tails, respectively, of *list*.

by $\left\{ \text{step} \left[\begin{array}{l} \textit{step} \\ \#\text{cdr} \end{array} \right] \right\}$
 ▷ Specify the (positive) decrement or increment or the *function* of one argument returning the next part of the list.

= *foo* [**then** *bar*_{*foo*}]
 ▷ Bind *var* initially to *foo* and later to *bar*.

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

being $\left\{ \text{the|each} \right\}$
 ▷ Iterate over a hash table or a package.

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

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

$\left\{ \text{symbol|symbols|present-symbol|present-symbols|external-symbol|external-symbols} \right\} \left[\left\{ \text{of|in} \right\} \textit{package} \left[\begin{array}{l} \textit{var} \\ \#\text{package}^* \end{array} \right] \right]$
 ▷ Bind *var* successively to the accessible symbols, or the present symbols, or the external symbols respectively, of *package*.

- {do|doing}** *form*⁺
 ▷ Evaluate *forms* in every iteration.
- {if|when|unless}** *test i-clause {and j-clause}* [else k-clause {and l-clause}*] [end]*
 ▷ If *test* returns T, T, or NIL, respectively, evaluate *i-clause* and *j-clauses*; otherwise, evaluate *k-clause* and *l-clauses*.
- it** ▷ Inside *i-clause* or *k-clause*: value of *test*.
- return** *{form|it}*
 ▷ Return immediately, skipping any **finally** parts, with values of *form* or **it**.
- {collect|collecting}** *{form|it} [into list]*
 ▷ Collect values of *form* or **it** into *list*. If no *list* is given, collect into an anonymous list which is returned after termination.
- {append|appending|nconc|nconcing}** *{form|it} [into list]*
 ▷ Concatenate values of *form* or **it**, which should be lists, into *list* by the means of **append** or **nconc**, respectively. If no *list* is given, collect into an anonymous list which is returned after termination.
- {count|counting}** *{form|it} [into n] [type]*
 ▷ Count the number of times the value of *form* or of **it** is T. If no *n* is given, count into an anonymous variable which is returned after termination.
- {sum|summing}** *{form|it} [into sum] [type]*
 ▷ Calculate the sum of the primary values of *form* or of **it**. If no *sum* is given, sum into an anonymous variable which is returned after termination.
- {maximize|maximizing|minimize|minimizing}** *{form|it} [into max-min] [type]*
 ▷ Determine the maximum or minimum, respectively, of the primary values of *form* or of **it**. If no *max-min* is given, use an anonymous variable which is returned after termination.
- {initially|finally}** *form*⁺
 ▷ Evaluate *forms* before begin, or after end, respectively, of iterations.
- repeat** *num*
 ▷ Terminate **loop**^M after *num* iterations; *num* is evaluated once.
- {while|until}** *test*
 ▷ Continue iteration until *test* returns NIL or T, respectively.
- {always|never}** *test*
 ▷ Terminate **loop**^M returning NIL and skipping any **finally** parts as soon as *test* is NIL or T, respectively. Otherwise continue **loop**^M with its default return value set to T.
- thereis** *test*
 ▷ Terminate **loop**^M when *test* is T and return value of *test*, skipping any **finally** parts. Otherwise continue **loop**^M with its default return value set to NIL.
- (loop-finish)**^M
 ▷ Terminate **loop**^M immediately executing any **finally** clauses and returning any accumulated results.

10 CLOS

10.1 Classes

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

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

$$\left(\begin{array}{l} \text{:default-initargs } \{ \text{name value} \}^* \\ \text{:documentation } \text{class-doc} \\ \text{:metaclass } \text{name} \text{standard-class} \end{array} \right)$$

▷ Define, as a subclass of *superclasses*, class *foo*. In a new instance *i*, a *slot*'s value defaults to *form* unless set via *:initarg-name*; it is readable via (*reader i*) or (*accessor i*), and writeable via (*writer value i*) or (**setf** (*accessor i*) *value*). With **:allocation :class**, *slot* is shared by all instances of class *foo*.

(^{Fu}**find-class** *symbol* [*errorp*] [*environment*])

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

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

▷ Make new instance of *class*.

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

▷ Change local slots of instance according to *initargs*.

(^{Fu}**slot-value** *foo slot*) ▷ Return value of *slot* in *foo*. **setfable**.

(^{Fu}**slot-makunbound** *instance slot*)

▷ Make *slot* in instance unbound.

(^M**with-slots** (*slot* (*var slot*)*) ^M**with-accessors** (*var accessor**) *instance* (**declare** *decl**) *form**)

▷ Return values of forms after evaluating them in a lexical environment with slots of *instance* visible as **setfable slots** or *vars/with accessors* of *instance* visible as **setfable vars**.

(^{gF}**class-name** *class*)

(^{gF}**(setf class-name)** *new-name class*) ▷ Get/set name of *class*.

(^{Fu}**class-of** *foo*) ▷ Class *foo* is a direct instance of.

(^{gF}**change-class** *instance new-class* *:initarg value** *other-keyarg**)

▷ Change class of instance to *new-class*.

(^{gF}**make-instances-obsolete** *class*)

▷ Update instances of *class*.

(^{gF}**initialize-instance** (*instance*) ^{gF}**update-instance-for-different-class** *previous current*)

*:initarg value** *other-keyarg**)
▷ Its primary method sets slots on behalf of **make-instance**/of **change-class** by means of **shared-initialize**.

(^{gF}**update-instance-for-redefined-class** *instances added-slots discarded-slots property-list* *:initarg value** *other-keyarg**)

▷ Its primary method sets slots on behalf of **make-instances-obsolete** by means of **shared-initialize**.

(^{gF}**allocate-instance** *class* *:initarg value** *other-keyarg**)

▷ Return uninitialized instance of *class*. Called by **make-instance**.

(^{gF}**shared-initialize** *instance* $\left\{ \begin{array}{l} \text{slots} \\ \text{T} \end{array} \right\}$ *:initarg value** *other-keyarg**)

▷ Fill *instance*'s *slots* using *initargs* and **:initform** forms.

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

▷ Called in case of attempted access to missing *slot*. Its primary method signals **error**.

(^{gF}**slot-unbound** *class instance slot*)

- ▷ Called by **slot-value** in case of unbound *slot*. Its primary method signals **unbound-slot**.

10.2 Generic Functions

(^{Fu}**next-method-p**)

- ▷ T if enclosing method has a next method.

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

- ▷ Define generic function *foo*. *defmethod-args* resemble those of **defmethod**. For *c-type* see section 10.3.

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

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

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

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

- ▷ 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.

($\left\{ \begin{array}{l} \text{add-method} \\ \text{remove-method} \end{array} \right\}^{\text{gF}}$ *generic-function method*)

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

(^{gF}**find-method** *generic-function qualifiers specializers* [*error* \square])

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

(^{gF}**compute-applicable-methods** *generic-function args*)

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

- (^{Fu}**call-next-method** *arg** current args)
 ▷ From within a method, call next method with *args*; return its values.
- (^{gF}**no-applicable-method** *generic-function arg**)
 ▷ Called on invocation of *generic-function* on *args* if there is no applicable method. Default method signals **error**.
- (^{Fu}**invalid-method-error** *method*)
 (^{Fu}**method-combination-error** *control arg**)
 ▷ Signal **error** on applicable method with invalid qualifiers, or on method combination. For *control* and *args* see **format**, p. 36.
- (^{gF}**no-next-method** *generic-function method arg**)
 ▷ Called on invocation of **call-next-method** when there is no next method. Default method signals **error**.
- (^{gF}**function-keywords** *method*)
 ▷ Return list of keyword parameters of *method* and $\frac{T}{2}$ if other keys are allowed.
- (^{gF}**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, ^{Fu}**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 ^{Fu}primary method which supplies the values of the calling **call-next-method** if any, or of the generic function; and which can call less specific primary methods via ^{Fu}**call-next-method**. After its return, call all **:after** methods, least specific first.

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

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

(^M**define-method-combination** *c-type*

{
 :documentation *string*
 :identity-with-one-argument *bool* NTL
 :operator *operator* *c-type*
 }

▷ **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, ^{Fu}**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 [^M{**:most-specific-first** **:most-specific-last** } **:most-specific-first**] (specified as *c-arg* in **defgeneric**). Using *c-type* as the *qualifier* in ^M**defmethod** 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*^{P*})

▷ **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. **defgeneric**), with *symbol* bound to the generic function, with *method-combination-λ** bound to the arguments of the generic function, and with *groups* bound to lists of methods. An applicable method becomes a member of the leftmost *group* whose *predicate* or *qualifiers* match. Methods can be called via **call-method**. Lambda lists (*ord-λ**) and (*method-combination-λ**) according to *ord-λ* on p. 16, the latter enhanced by an optional **&whole** argument.

$$(\overset{\text{M}}{\text{call-method}} \left\{ \widehat{\text{method}} \left(\overset{\text{M}}{\text{make-method}} \widehat{\text{form}} \right) \right\} \left[\left(\left\{ \widehat{\text{next-method}} \left(\overset{\text{M}}{\text{make-method}} \widehat{\text{form}} \right) \right\}^* \right) \right] \right)$$

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

11 Conditions and Errors

For standardized condition types cf. Figure 2 on page 30.

$$(\overset{\text{M}}{\text{define-condition}} \text{foo} (\text{parent-type}^* \boxed{\text{condition}})$$

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

$$\left(\left\{ \begin{array}{l} (\text{:default-initargs } \{ \text{name value} \}^*) \\ (\text{:documentation } \text{condition-doc}) \\ (\text{:report } \left\{ \begin{array}{l} \text{string} \\ \text{report-function} \end{array} \right\}) \end{array} \right\} \right)$$

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

$$(\overset{\text{Fu}}{\text{make-condition}} \text{type} \{\text{:initarg-name value}\}^*)$$

▷ Return new condition of type.

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

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

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

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

$$(\overset{\text{M}}{\text{ignore-errors}} \text{form}^{\text{P}})$$

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

$$(\overset{\text{Fu}}{\text{invoke-debugger}} \text{condition})$$

▷ Invoke debugger with *condition*.

(^M**assert** *test* [(*place*^{*}) [{ *condition* *continue-arg*^{*} }
 { *type* {*:initarg* *name* *value*}^{*} }
 { *control* *arg*^{*} }]])

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

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

▷ If, on evaluation of *foo*, a condition of *type* is signalled, evaluate matching *condition-forms* with *var* bound to the condition, and return their values. Without a condition, bind *ord-λs* to values of *foo* and return values of forms or, without a **:no-error** clause, return values of *foo*. See p. 16 for (*ord-λ*^{*}).

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

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

(^M**with-simple-restart** ({ *restart* }
 { **NIL** } *control* *arg*^{*}) *form*^{P*})

▷ Return values of forms unless *restart* is called during their evaluation. In this case, describe restart using ^{Fu}**format** *control* and *args* (see p. 36) and return **NIL** and $\frac{T}{2}$.

(^M**restart-case** *form* (*foo* (*ord-λ*^{*}) { **:interactive** *arg-function* }
 { **:report** { *report-function* }
 { **:test** *test-function*_□ }

(**declare** $\widehat{\text{decl}}^*$)^{*} *restart-form*^{P*})^{*}
▷ Evaluate *form* with dynamically established restarts *foo*. Return values of form or, if by (^{Fu}**invoke-restart** *foo* *arg*^{*}) one restart *foo* is called, use *string* or *report-function* (of a stream) to print a description of restart *foo* and return the values of its restart-forms. *arg-function* supplies appropriate *args* if *foo* is called by ^{Fu}**invoke-restart-interactively**. If (*test-function* *condition*) returns **T**, *foo* is made visible under *condition*. *arg*^{*} matches (*ord-λ*^{*}); see p. 16 for the latter.

(^M**restart-bind** (({ $\widehat{\text{restart}}$ }
 { **NIL** } *restart-function*

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

▷ Return values of forms evaluated with *restarts* dynamically bound to *restart-functions*.

(^{Fu}**invoke-restart** *restart* *arg*^{*})

(^{Fu}**invoke-restart-interactively** *restart*)

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

({ ^{Fu}**compute-restarts** }
 { ^{Fu}**find-restart** *name* } [*condition*])

▷ Return list of all restarts, or innermost restart *name*, 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.

(^{Fu}**restart-name** *restart*) ▷ Name of restart.

({ ^{Fu}**abort**
 { ^{Fu}**muffle-warning**
 { ^{Fu}**continue**
 { ^{Fu}**store-value** *value* }
 { ^{Fu}**use-value** *value* } } } } [*condition*_□])

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

- (^M**with-condition-restarts** *condition restarts form^{R*}*)
 ▷ Evaluate *forms* with *restarts* dynamically associated with *condition*. Return values of forms.
- (^{Fu}**arithmetic-error-operation** *condition*)
 (^{Fu}**arithmetic-error-operands** *condition*)
 ▷ List of function or of its operands respectively, used in the operation which caused *condition*.
- (^{Fu}**cell-error-name** *condition*)
 ▷ Name of cell which caused *condition*.
- (^{Fu}**unbound-slot-instance** *condition*)
 ▷ Instance with unbound slot which caused *condition*.
- (^{Fu}**print-not-readable-object** *condition*)
 ▷ The object not readably printable under *condition*.
- (^{Fu}**package-error-package** *condition*)
 (^{Fu}**file-error-pathname** *condition*)
 (^{Fu}**stream-error-stream** *condition*)
 ▷ Package, path, or stream, respectively, which caused the *condition* of indicated type.
- (^{Fu}**type-error-datum** *condition*)
 (^{Fu}**type-error-expected-type** *condition*)
 ▷ Object which caused *condition* of type **type-error**, or its expected type, respectively.
- (^{Fu}**simple-condition-format-control** *condition*)
 (^{Fu}**simple-condition-format-arguments** *condition*)
 ▷ Return format control or list of format arguments, respectively, of *condition*.
- ^{var}***break-on-signals***_{NIL}
 ▷ Condition type debugger is to be invoked on.
- ^{var}***debugger-hook***_{NIL}
 ▷ Function of condition and function itself. Called before debugger.

12 Types and Classes

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

- (^{Fu}**typep** *foo type* [*environment*_{NIL}]) ▷ T if *foo* is of *type*.
- (^{Fu}**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.
- (^{SO}**the** *type form*) ▷ Declare values of form to be of *type*.
- (^{Fu}**coerce** *object type*) ▷ Coerce object into *type*.
- (^M**typecase** *foo* (*type a-form^{R*}*)* [(otherwise _T *b-form^{R*}*_{NIL}*)])
 ▷ Return values of the a-forms whose *type* is *foo* of. Return values of b-forms if no *type* matches.
- (^M**ctypcase** / ^M**etypcase**) *foo* (*type form^{R*}*)*
 ▷ Return values of the forms whose *type* is *foo* of. Signal correctable/non-correctable error, respectively if no *type* matches.
- (^{Fu}**type-of** *foo*) ▷ Type of foo.
- (^M**check-type** *place type* [*string*_{{a|an} type}])
 ▷ Signal correctable **type-error** if *place* is not of *type*. Return NIL.
- (^{Fu}**stream-element-type** *stream*) ▷ Return type of *stream* objects.
- (^{Fu}**array-element-type** *array*) ▷ Element type *array* can hold.

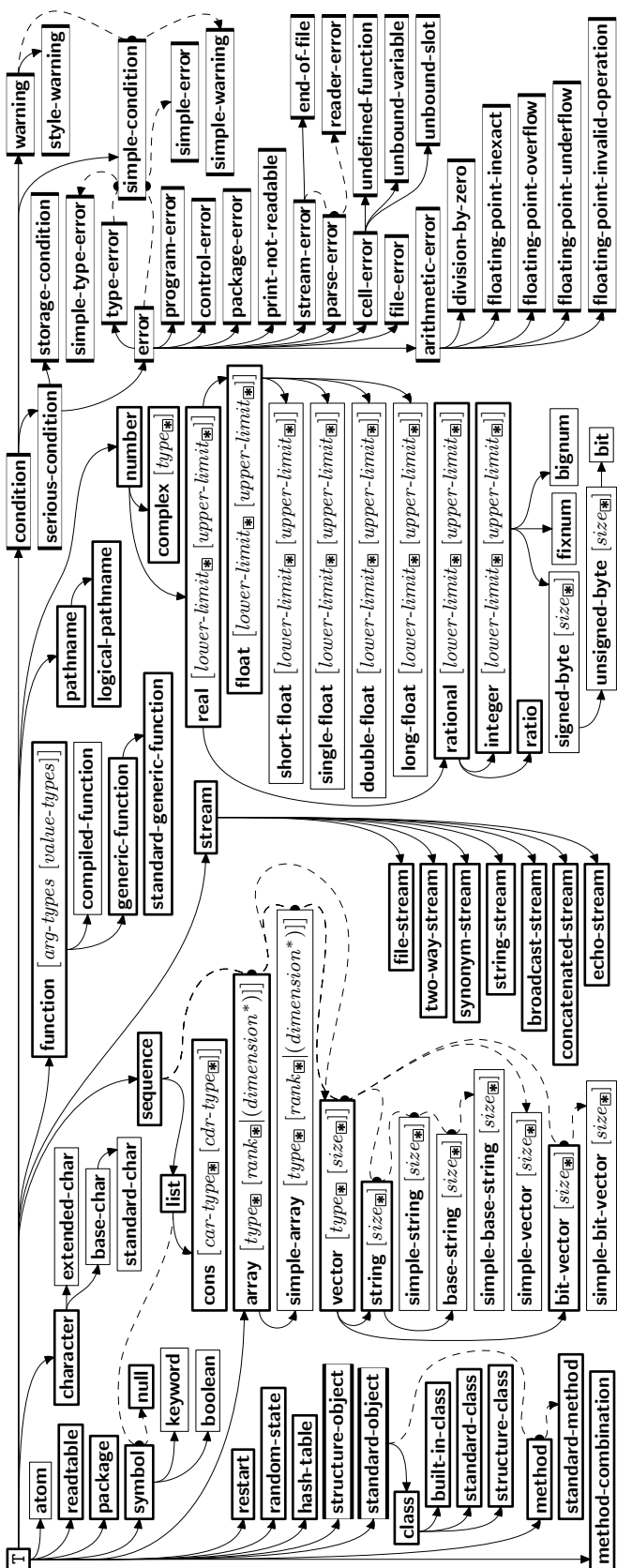


Figure 2: Precedence Order of System Classes (□), Classes (▮), Types (▭), and Condition Types (◡).

- (^{Fu}**upgraded-array-element-type** *type* [*environment*_{NIL}])
 ▷ Element type of most specialized array capable of holding elements of *type*.
- (^M**deftype** *foo* (*macro-λ**) (**declare** \widehat{decl}^*)^{*} [\widehat{doc}] *form*^{P*})
 ▷ Define type foo which when referenced as (*foo* \widehat{arg}^*) applies expanded *forms* to *args* returning the new type. For (*macro-λ**) see p. 18 but with default value of ***** instead of NIL. *forms* are enclosed in an implicit ^{SO}**block** named *foo*.
- (**eql** *foo*)
 (**member** *foo*^{*}) ▷ Specifier for a type comprising *foo* or *foos*.
- (**satisfies** *predicate*)
 ▷ Type specifier for all objects satisfying *predicate*.
- (**mod** *n*) ▷ Type specifier for all non-negative integers < *n*.
- (**not** *type*) ▷ Complement of type.
- (**and** *type*^{*}_T) ▷ Type specifier for intersection of *types*.
- (**or** *type*^{*}_{NIL}) ▷ Type specifier for union of *types*.
- (**values** *type*^{*} [**&optional** *type*^{*} [**&rest** *other-args*]])
 ▷ Type specifier for multiple values.
- *** ▷ As a type argument (cf. Figure 2): no restriction.

13 Input/Output

13.1 Predicates

- (^{Fu}**stream** *foo*)
 (^{Fu}**pathnamep** *foo*) ▷ T if *foo* is of indicated type.
 (^{Fu}**readtablep** *foo*)
- (^{Fu}**input-stream-p** *stream*)
 (^{Fu}**output-stream-p** *stream*)
 (^{Fu}**interactive-stream-p** *stream*)
 (^{Fu}**open-stream-p** *stream*)
 ▷ Return T if *stream* is for input, for output, interactive, or open, respectively.
- (^{Fu}**pathname-match-p** *path* *wildcard*)
 ▷ T if *path* matches *wildcard*.
- (^{Fu}**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

- (^{Fu}**y-or-n-p** / ^{Fu}**yes-or-no-p**) [*control* *arg*^{*}])
 ▷ Ask user a question and return T or NIL depending on their answer. See p. 36, ^{Fu}**format**, for *control* and *args*.
- (^M**with-standard-io-syntax** *form*^{P*})
 ▷ Evaluate *forms* with standard behaviour of reader and printer. Return values of forms.
- (^{Fu}**read** / ^{Fu}**read-preserving-whitespace**) [*stream*_{*var} ***standard-input*** [*eof-err*_T [*eof-val*_{NIL} [*recursive*_{NIL}]]]]
 ▷ Read printed representation of object.
- (^{Fu}**read-from-string** *string* [*eof-error*_T [*eof-val*_{NIL} [**:start** *start*_T [**:end** *end*_{NIL} [**:preserve-whitespace** *bool*_{NIL}]]]]])
 ▷ Return object read from string and zero-indexed ₂position of next character.

- (^{Fu}**read-delimited-list** *char* [*stream* ^{var}***standard-input***] [*recursive* **NIL**]))
 ▷ Continue reading until encountering *char*. Return list of objects read. Signal error if no *char* is found in stream.
- (^{Fu}**read-char** [*stream* ^{var}***standard-input***] [*eof-err* **T**] [*eof-val* **NIL**] [*recursive* **NIL**]]))
 ▷ Return next character from *stream*.
- (^{Fu}**read-char-no-hang** [*stream* ^{var}***standard-input***] [*eof-error* **N**] [*eof-val* **NIL**] [*recursive* **NIL**]]))
 ▷ Next character from *stream* or **NIL** if none is available.
- (^{Fu}**peek-char** [*mode* **NIL**] [*stream* ^{var}***standard-input***] [*eof-error* **T**] [*eof-val* **NIL**] [*recursive* **NIL**]]))
 ▷ Next, or if *mode* is **T**, next non-whitespace character, or if *mode* is a character, next instance of it, from *stream* without removing it there.
- (^{Fu}**unread-char** *character* [*stream* ^{var}***standard-input***]))
 ▷ Put last ^{Fu}**read-chared** *character* back into *stream*; return NIL.
- (^{Fu}**read-byte** *stream* [*eof-err* **N**] [*eof-val* **NIL**]))
 ▷ Read next byte from binary *stream*.
- (^{Fu}**read-line** [*stream* ^{var}***standard-input***] [*eof-err* **N**] [*eof-val* **NIL**] [*recursive* **NIL**]]))
 ▷ Return a line of text from *stream* and **T** if line has been ended by end of file. ₂
- (^{Fu}**read-sequence** *sequence* *stream* [*start* *start* **0**] [*end* *end* **NIL**]))
 ▷ Replace elements of *sequence* between *start* and *end* with elements from binary or character *stream*. Return index of *sequence*'s first unmodified element.
- (^{Fu}**readtable-case** *readtable*)^{supcase}
 ▷ Case sensitivity attribute (one of **:upcase**, **:downcase**, **:preserve**, **:invert**) of *readtable*. **setfable**.
- (^{Fu}**copy-readtable** [*from-readtable* ^{var}***readtable***] [*to-readtable* **NIL**]))
 ▷ Return copy of from-readtable.
- (^{Fu}**set-syntax-from-char** *to-char* *from-char* [*to-readtable* ^{var}***readtable***] [*from-readtable* **standard-readtable**]))
 ▷ Copy syntax of *from-char* to *to-readtable*. Return T.
- ^{var}***readtable*** ▷ Current readtable.
- ^{var}***read-base***₁₀ ▷ Radix for reading **integers** and **ratios**.
- ^{var}***read-default-float-format***^{single-float}
 ▷ Floating point format to use when not indicated in the number read.
- ^{var}***read-suppress***^{NIL}
 ▷ If **T**, reader is syntactically more tolerant.
- (^{Fu}**set-macro-character** *char* *function* [*non-term-p* **NIL**] [*rt* ^{var}***readtable***]))
 ▷ Make *char* a macro character associated with *function* of stream and *char*. Return T.
- (^{Fu}**get-macro-character** *char* [*rt* ^{var}***readtable***]))
 ▷ Reader macro function associated with *char*, and **T** if *char* is a non-terminating macro character. ₂
- (^{Fu}**make-dispatch-macro-character** *char* [*non-term-p* **NIL**] [*rt* ^{var}***readtable***]))
 ▷ Make *char* a dispatching macro character. Return T.
- (^{Fu}**set-dispatch-macro-character** *char* *sub-char* *function* [*rt* ^{var}***readtable***]))
 ▷ Make *function* of stream, *n*, *sub-char* a dispatch function of *char* followed by *n*, followed by *sub-char*. Return T.
- (^{Fu}**get-dispatch-macro-character** *char* *sub-char* [*rt* ^{var}***readtable***]))
 ▷ Dispatch function associated with *char* followed by *sub-char*.

13.3 Character Syntax

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

`; one-line-comment*`

▷ Comments. There are stylistic conventions:

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

`(foo* [. barNIL])` ▷ List of *foos* with the terminating *cdr bar*.

`"` ▷ Begin and end of a string.

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

``([foo] [bar] [,@baz] [.,SOquux] [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` ▷ (`character "c"`), the character *c*.

`#Bn`; `#On`; `n.`; `#Xn`; `#rRn`

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

`n/d` ▷ The **ratio** $\frac{n}{d}$.

`{[m].n[{S|F|D|L|E}xEO]|m[.n]{{S|F|D|L|E}x}}`

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

`#C(a b)` ▷ (`complex 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.

`#[n]*b*`

▷ Bit vector of some (or *n*) *bs* filled with last *b* if necessary.

`#S(type {slot value}*)` ▷ Structure of *type*.

`#Pstring` ▷ A pathname.

`#:foo` ▷ Uninterned symbol *foo*.

`#.form` ▷ Read-time value of *form*.

`var*read-eval*T` ▷ 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 `var*features*`, or (`{and|or} feature*`), or (`not feature`).

`var*features*`

▷ List of symbols denoting implementation-dependent features.

`|c*|; \c`

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

13.4 Printer

(^{Fu}**prin1** ^{Fu}**print** ^{Fu}**pprint** ^{Fu}**princ**) *foo* [*stream* ^{var}***standard-output***])

▷ Print *foo* to *stream* ^{Fu}readably, ^{Fu}readably between a newline and a space, ^{Fu}readably after a newline, or human-readably without any extra characters, respectively. ^{Fu}**prin1**, ^{Fu}**print** and ^{Fu}**princ** return foo.

(^{Fu}**prin1-to-string** *foo*)

(^{Fu}**princ-to-string** *foo*)

▷ Print *foo* to string ^{Fu}readably or human-readably, respectively.

(^{gF}**print-object** *object* *stream*)

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

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

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

(^{Fu}**terpri** [*stream* ^{var}***standard-output***])

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

(^{Fu}**fresh-line**) [*stream* ^{var}***standard-output***]

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

(^{Fu}**write-char** *char* [*stream* ^{var}***standard-output***])

▷ Output char to *stream*.

(^{Fu}**write-string** ^{Fu}**write-line**) *string* [*stream* ^{var}***standard-output*** [{**:start** *start*₀ **:end** *end*_{NIL}}]])

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

(^{Fu}**write-byte** *byte* *stream*)

▷ Write byte to binary *stream*.

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

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

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

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

(^{Fu}**pprint-fill** *stream* *foo* [*parenthesis*_¶ [*noop*]])

(^{Fu}**pprint-tabular** *stream* *foo* [*parenthesis*_¶ [*noop* [*n*_{¶6}]])])

(^{Fu}**pprint-linear** *stream* *foo* [*parenthesis*_¶ [*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 ^{Fu}**format** directive ~//.

- (^{Fu}**set-pprint-dispatch** *type function* [*priority*₀]
 [*table*_{var} ***print-pprint-dispatch***]))
- ▷ Install entry comprising *function* of arguments *stream* and *object* to print; and *priority* as *type* into *table*. If *function* is NIL, remove *type* from *table*. Return NIL.
- (^{Fu}**pprint-dispatch** *foo* [*table*_{var} ***print-pprint-dispatch***]))
- ▷ Return highest priority *function* associated with type of *foo* and T if there was a matching type specifier in *table*.
- (^{Fu}**copy-pprint-dispatch** [*table*_{var} ***print-pprint-dispatch***]))
- ▷ Return *copy* of *table* or, if *table* is NIL, initial value of ***print-pprint-dispatch***.
- *print-pprint-dispatch*** ▷ Current pretty print dispatch table.

13.5 Format

- (^M**formatter** *control*)
- ▷ Return *function* of stream and a **&rest** argument applying **format** to stream, *control*, and the **&rest** argument returning NIL or any excess arguments.
- (^{Fu}**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 **formatter** which is then applied to *out-stream* and *arg**. Output to *out-string*, *out-stream* or, if first argument is T, to ***standard-output***. Return NIL. If first argument is NIL, return *formatted output*.
- ~ [*min-col*₀] [, [*col-inc*₁] [, [*min-pad*₀] [, *pad-char*_□]]] [:] [**@**] {**A**|**S**}
- ▷ **Aesthetic/Standard**. Print argument of any type for consumption by humans/by the reader, respectively. With :, print NIL as () rather than nil; with @, add *pad-chars* on the left rather than on the right.
- ~ [*radix*₀] [, [*width*] [, [*pad-char*_□] [, [*comma-char*_□] [, [*comma-interval*₀]]]]] [:] [**@**] **R**
- ▷ **Radix**. (With one or more prefix arguments.) Print argument as number; with :, group digits *comma-interval* each; with @, always prepend a sign.
- {~**R**|~:**R**|~**@R**|~**@:R**}
- ▷ **Roman**. Take argument as number and print it as English cardinal number, as English ordinal number, as Roman numeral, or as old Roman numeral, respectively.
- ~ [*width*] [, [*pad-char*_□] [, [*comma-char*_□] [, [*comma-interval*₀]]]] [:] [**@**] {**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*] [, [*dec-digits*] [, [*shift*₀] [, [*overflow-char*] [, [*pad-char*_□]]]]] [**@**] **F**
- ▷ **Fixed-Format Floating-Point**. With @, always prepend a sign.
- ~ [*width*] [, [*int-digits*] [, [*exp-digits*] [, [*scale-factor*₀] [, [*overflow-char*] [, [*pad-char*_□] [, [*exp-char*]]]]]]] [**@**] {**E**|**G**}
- ▷ **Exponential/General Floating-Point**. Print argument as floating-point number with *int-digits* before decimal point and *exp-digits* in the signed exponent. With ~**G**, choose either ~**E** or ~**F**. With @, always prepend a sign.
- ~ [*dec-digits*₂] [, [*int-digits*₁] [, [*width*₀] [, [*pad-char*_□]]]] [:] [**@**] **\$**
- ▷ **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.

- $\{ \sim (\text{text} \sim) | \sim : (\text{text} \sim) | \sim \textcircled{\text{C}} (\text{text} \sim) | \sim : \textcircled{\text{C}} (\text{text} \sim) \}$
 ▷ **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.
- $\{ \sim \text{P} | \sim : \text{P} | \sim \textcircled{\text{P}} | \sim : \textcircled{\text{P}} \}$
 ▷ **Plural.** If argument **eq1** print nothing, otherwise print **s**; do the same for the previous argument; if argument **eq1** print **y**, otherwise print **ies**; do the same for the previous argument, respectively.
- $\sim [n_{\square}] \% \quad \triangleright$ **Newline.** Print *n* newlines.
- $\sim [n_{\square}] \&$
 ▷ **Fresh-Line.** Print *n* – 1 newlines if output stream is at the beginning of a line, or *n* newlines otherwise.
- $\{ \sim - | \sim : - | \sim \textcircled{-} | \sim : \textcircled{-} \}$
 ▷ **Conditional Newline.** Print a newline like **pprint-newline** with argument **:linear**, **:fill**, **:miser**, or **:mandatory**, respectively.
- $\{ \sim : \leftarrow | \sim \textcircled{\leftarrow} | \sim \leftarrow \}$
 ▷ **Ignored Newline.** Ignore newline, or whitespace following newline, or both, respectively.
- $\sim [n_{\square}] | \quad \triangleright$ **Page.** Print *n* page separators.
- $\sim [n_{\square}] \sim \quad \triangleright$ **Tilde.** Print *n* tildes.
- $\sim [min-col_{\square}] [, [col-inc_{\square}] [, [min-pad_{\square}] [, [pad-char_{\square}]]]$
 $[: \textcircled{\text{C}} < [nl-text \sim [spare_{\square}] [, [width]]] ; \{ \text{text} \sim ; \}^* \text{text} \sim >$
 ▷ **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.
- $\sim [:] [\textcircled{\text{C}} < \{ [prefix_{\square} \sim ;] | [per-line-prefix \sim \textcircled{\text{C}} ;] \} \text{body} [\sim ;$
 $suffix_{\square}] \sim : [\textcircled{\text{C}} >$
 ▷ **Logical Block.** Act like **pprint-logical-block** using *body* as **format** control string on the elements of the list argument or, with **Ⓞ**, on the remaining arguments, which are extracted by **pprint-pop**. With **:**, *prefix* and *suffix* default to (and). When closed by **~:Ⓞ>**, spaces in *body* are replaced with conditional newlines.
- $\{ \sim [n_{\square}] \text{i} | \sim [n_{\square}] : \text{i} \}$
 ▷ **Indent.** Set indentation to *n* relative to leftmost/to current position.
- $\sim [c_{\square}] [, i_{\square}] [:] [\textcircled{\text{C}} \text{T}$
 ▷ **Tabulate.** Move cursor forward to column number $c + ki$, $k \geq 0$ being as small as possible. With **:**, calculate column numbers relative to the immediately enclosing section. With **Ⓞ**, move to column number $c_0 + c + ki$ where c_0 is the current position.
- $\{ \sim [m_{\square}] * | \sim [m_{\square}] : * | \sim [n_{\square}] \textcircled{*} \}$
 ▷ **Go-To.** Jump *m* arguments forward, or backward, or to argument *n*.
- $\sim [limit] [:] [\textcircled{\text{C}} \{ \text{text} \sim \}$
 ▷ **Iteration.** Use *text* repeatedly, up to *limit*, as control string for the elements of the list argument or (with **Ⓞ**) for the remaining arguments. With **:** or **Ⓞ**, list elements or remaining arguments should be lists of which a new one is used at each iteration step.
- $\sim [x [, y [, z]]] \wedge$
 ▷ **Escape Upward.** Leave immediately $\sim < \sim >$, $\sim < \sim : >$, $\sim \{ \sim \}$, $\sim ?$, or the entire **format** operation. With one to three prefixes, act only if $x = 0$, $x = y$, or $x \leq y \leq z$, respectively.
- $\sim [i] [:] [\textcircled{\text{C}} [[\{ \text{text} \sim ; \}^* \text{text}] [\sim ; \text{default}] \sim]$
 ▷ **Conditional Expression.** Use the zero-indexed argument (or *i*th if given) *text* as a **format** control subclause. With **:**, use the first *text* if the argument value is NIL, or the second *text* if it is T. With **Ⓞ**, do nothing for an argument value of NIL. Use the only *text* and leave the argument to be read again if it is T.

~ [ⓐ] ?

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

~ [prefix {,prefix}*] [:] [ⓐ] / [package ::[c1-user:]]function/

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

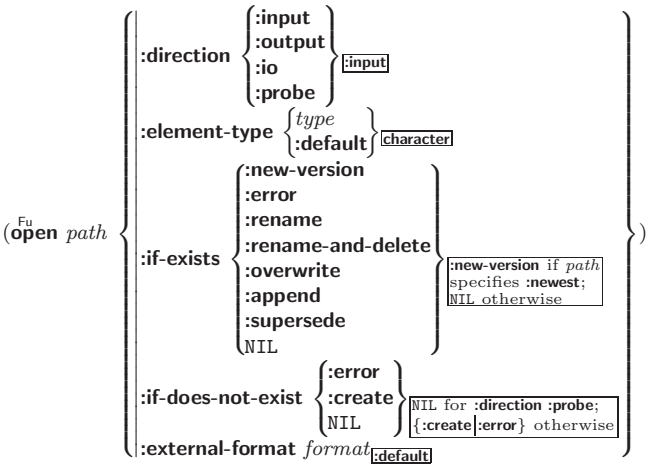
~ [:] [ⓐ] W

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

{V|#}

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

13.6 Streams



▷ Open file-stream to path.

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

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

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

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

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

▷ Return stream of indicated type.

(Fu **make-string-input-stream** *string* [*start*ⓐ [*end*ⓐ]])

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

(Fu **make-string-output-stream** [:element-type *type*ⓐ])

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

(Fu **concatenated-stream-streams** *concatenated-stream*)

(Fu **broadcast-stream-streams** *broadcast-stream*)

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

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

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

(Fu **echo-stream-input-stream** *echo-stream*)

(Fu **echo-stream-output-stream** *echo-stream*)

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

(Fu **synonym-stream-symbol** *synonym-stream*)

▷ Return symbol of *synonym-stream*.

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

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

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

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

(^{Fu}**file-string-length** *stream* *foo*)

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

(^{Fu}**listen** [*stream* ^{var}*standard-input*])

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

(^{Fu}**clear-input** [*stream* ^{var}*standard-input*])

▷ Clear input from *stream*, return NIL.

{ ^{Fu}**clear-output**
^{Fu}**force-output**
^{Fu}**finish-output** } [*stream* ^{var}*standard-output*])

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

(^{Fu}**close** *stream* [:abort *bool* NIL])

▷ Close *stream*. Return T if *stream* had been open. If :abort is T, delete associated file.

(^M**with-open-file** (*stream* *path* *open-arg**) (**declare** decl*)* *form*^{P*})

▷ Use ^{Fu}**open** with *open-args* to temporarily create *stream* to *path*; return values of forms.

(^M**with-open-stream** (*foo* *stream*) (**declare** decl*)* *form*^{P*})

▷ Evaluate *forms* with *foo* locally bound to *stream*. Return values of forms.

(^M**with-input-from-string** (*foo* *string* { :index *index*
:start *start*_Q
:end *end*NIL }) (**declare** decl*)* *form*^{P*})

▷ Evaluate *forms* with *foo* locally bound to input **string-stream** from *string*. Return values of forms; store next reading position into *index*.

(^M**with-output-to-string** (*foo* [*string*NIL [:element-type *type*_{character}]]) (**declare** decl*)* *form*^{P*})

▷ Evaluate *forms* with *foo* locally bound to an output **string-stream**. Append output to *string* and return values of forms if *string* is given. Return string containing output otherwise.

(^{Fu}**stream-external-format** *stream*)

▷ External file format designator.

^{var}***terminal-io*** ▷ Bidirectional stream to user terminal.

^{var}***standard-input***

^{var}***standard-output***

^{var}***error-output***

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

^{var}***debug-io***

^{var}***query-io***

▷ Bidirectional streams for debugging and user interaction.

13.7 Pathnames and Files

^{Fu}(**make-pathname**

}	<pre> :host {host NIL :unspecific} :device {device NIL :unspecific} :directory { {directory :wild NIL :unspecific} { {:absolute} {:relative} } { directory :wild :wild-inferiors :up :back } } :name {file-name :wild NIL :unspecific} :type {file-type :wild NIL :unspecific} :version {:newest version :wild NIL :unspecific} :defaults path_{host from} <u>*default-pathname-defaults*</u> :case {:local :common} :<u>local</u> </pre>	}

▷ Construct 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.

}	^{Fu} pathname-host	}	path [:case { :local :common } : <u>local</u>]
	^{Fu} pathname-device		
	^{Fu} pathname-directory		
	^{Fu} pathname-name		
	^{Fu} pathname-type		

^{Fu}(**pathname-version** *path*)▷ Return pathname component.^{Fu}(**parse-namestring** *foo* [*host*

```

[default-pathnamevar *default-pathname-defaults*
 {
  :start start0
  :end endNIL
  :junk-allowed boolNIL
 } ]])

```

▷ Return pathname converted from string, pathname, or stream *foo*; and position where parsing stopped.

^{Fu}(**merge-pathnames** *pathname*

```

[default-pathnamevar *default-pathname-defaults*
 [default-versionnewest]]])

```

▷ Return pathname after filling in missing components from *default-pathname*.

^{var}***default-pathname-defaults***

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

^{Fu}(**user-homedir-pathname** [*host*])▷ User's home directory.^{Fu}(**enough-namestring** *path* [*root-path* *default-pathname-defaults*])

▷ Return minimal path string to sufficiently describe *path* relative to *root-path*.

^{Fu}(**namestring** *path*)^{Fu}(**file-namestring** *path*)^{Fu}(**directory-namestring** *path*)^{Fu}(**host-namestring** *path*)

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

^{Fu}(**translate-pathname** *path wildcard-path-a wildcard-path-b*)

▷ Translate *path* from *wildcard-path-a* into *wildcard-path-b*. Return new path.

^{Fu}(**pathname** *path*)▷ Pathname of *path*.^{Fu}(**logical-pathname** *logical-path*)

▷ Logical pathname of *logical-path*. Logical pathnames are represented as all-uppercase #P"[host:][:]{{dir|*}}⁺;}*
{name|*}* [. {{type|*}}⁺]_{LISP}] . {version|*|newest|NEWEST}]]".

^{Fu}(**logical-pathname-translations** *logical-host*)

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

- (^{Fu}**load-logical-pathname-translations** *logical-host*)
 ▷ Load *logical-host*'s translations. Return NIL if already loaded; return T if successful.
- (^{Fu}**translate-logical-pathname** *pathname*)
 ▷ Physical pathname corresponding to (possibly logical) *pathname*.
- (^{Fu}**probe-file** *file*)
 (^{Fu}**truename** *file*)
 ▷ Canonical name of *file*. If *file* does not exist, return NIL/signal **file-error**, respectively.
- (^{Fu}**file-write-date** *file*) ▷ Time at which *file* was last written.
- (^{Fu}**file-author** *file*) ▷ Return name of file owner.
- (^{Fu}**file-length** *stream*) ▷ Return length of stream.
- (^{Fu}**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.
- (^{Fu}**delete-file** *file*) ▷ Delete *file*. Return T.
- (^{Fu}**directory** *path*) ▷ List of pathnames matching *path*.
- (^{Fu}**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

14.1 Predicates

- (^{Fu}**symbolp** *foo*)
 (^{Fu}**packagep** *foo*) ▷ T if *foo* is of indicated type.
 (^{Fu}**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.

(^{Fu}**make-package** *foo* {
 (:**nicknames** (*nick**)NIL)
 (:**use** (*used-package**)
 })

▷ Create package *foo*.

(^{Fu}**rename-package** *package new-name* [*new-nicknames*NIL])
 ▷ Rename *package*. Return renamed package.

(^M**in-package** *foo*) ▷ Make package *foo* current.

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

- (^{Fu}**package-use-list** *package*)
 (^{Fu}**package-used-by-list** *package*)
 ▷ List of other packages used by/using *package*.
- (^{Fu}**delete-package** *package*)
 ▷ Delete *package*. Return T if successful.
- ^{var}***package*** common-lisp-user ▷ The current package.
- (^{Fu}**list-all-packages**) ▷ List of registered packages.
- (^{Fu}**package-name** *package*) ▷ Name of *package*.
- (^{Fu}**package-nicknames** *package*) ▷ List of nicknames of *package*.
- (^{Fu}**find-package** *name*) ▷ Package with *name* (case-sensitive).
- (^{Fu}**find-all-symbols** *foo*)
 ▷ List of symbols *foo* from all registered packages.
- (^{Fu}**intern** / ^{Fu}**find-symbol**) *foo* [*package* ^{var}***package***]
- ▷ Intern or find, respectively, symbol *foo* in *package*. Second return value is one of :internal, :external, or :inherited (or NIL if ^{Fu}**intern** created a fresh symbol).
- (^{Fu}**unintern** *symbol* [*package* ^{var}***package***])
 ▷ Remove *symbol* from *package*, return T on success.
- (^{Fu}**import** / ^{Fu}**shadowing-import**) *symbols* [*package* ^{var}***package***]
- ▷ Make *symbols* internal to *package*. Return T. In case of a name conflict signal correctable **package-error** or shadow the old symbol, respectively.
- (^{Fu}**shadow** *symbols* [*package* ^{var}***package***])
 ▷ Make *symbols* of *package* shadow any otherwise accessible, equally named symbols from other packages. Return T.
- (^{Fu}**package-shadowing-symbols** *package*)
 ▷ List of symbols of *package* that shadow any otherwise accessible, equally named symbols from other packages.
- (^{Fu}**export** *symbols* [*package* ^{var}***package***])
 ▷ Make *symbols* external to *package*. Return T.
- (^{Fu}**unexport** *symbols* [*package* ^{var}***package***])
 ▷ Revert *symbols* to internal status. Return T.
- (^M**do-symbols** / ^M**do-external-symbols** / ^M**do-all-symbols**) (*var* [*package* ^{var}***package***] [*result* NIL])
 (**declare** *decl**)* {*tag* / *form*}*
 ▷ Evaluate ^{so}**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 ^{so}**block** named NIL.
- (^M**with-package-iterator** (*foo packages* [:internal|:external|:inherited])
 (**declare** *decl**)* *form*^{P*})
 ▷ Return values of *forms*. In *forms*, successive invocations of (*foo*) return: T if a symbol is returned; a symbol from *packages*; accessibility (:internal, :external, or :inherited); and the package the symbol belongs to.
- (^{Fu}**require** *module* [*paths* NIL])
 ▷ If not in ^{var}***modules***, try *paths* to load *module* from. Signal **error** if unsuccessful. Deprecated.
- (^{Fu}**provide** *module*)
 ▷ If not already there, add *module* to ^{var}***modules***. Deprecated.
- ^{var}***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*).

(^{Fu}**make-symbol** *name*)

▷ Make fresh, uninterned symbol *name*.

(^{Fu}**gensym** [*s*])

▷ Return fresh, uninterned symbol #:sn with *n* from ^{var}*gensym-counter*. Increment ^{var}*gensym-counter*.

(^{Fu}**gentemp** [*prefix*] [*package* [^{var}*package*]])

▷ Intern fresh symbol in package. Deprecated.

(^{Fu}**copy-symbol** *symbol* [*props* ^{NTI}])

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

(^{Fu}**symbol-name** *symbol*)

(^{Fu}**symbol-package** *symbol*)

(^{Fu}**symbol-plist** *symbol*)

(^{Fu}**symbol-value** *symbol*)

(^{Fu}**symbol-function** *symbol*)

▷ Name, package, property list, value, or function, respectively, of *symbol*. **setfable**.

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

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

^{co}**t**

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

^{co}**nil**(^{co})

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

14.4 Standard Packages

common-lisp|**cl**

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

common-lisp-user|**cl-user**

▷ Current package after startup; uses package **common-lisp**.

keyword

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

15 Compiler

15.1 Predicates

(^{Fu}**special-operator-p** *foo*)

▷ T if *foo* is a special operator.

(^{Fu}**compiled-function-p** *foo*)

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

15.2 Compilation

(^{Fu}**compile** {^{NTI}{*name* (^{setf} *name*)} [*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.

(^{Fu}compile-file *file* {
 :output-file *out-path*
 :verbose *bool* ^{var} *compile-verbose*
 :print *bool* ^{var} *compile-print*
 :external-format *file-format* default })

▷ Write compiled contents of *file* to *out-path*. Return true output path or NIL, T in case of warnings or errors, T in case of warnings or errors excluding style warnings.

(^{Fu}compile-file-pathname *file* [:output-file *path*] [*other-keyargs*])

▷ Pathname ^{Fu}compile-file writes to if invoked with the same arguments.

(^{Fu}load *path* {
 :verbose *bool* ^{var} *load-verbose*
 :print *bool* ^{var} *load-print*
 :if-does-not-exist *bool* T
 :external-format *file-format* default })

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

^{var}*compile-file {
^{var}*load } {
 pathname* NIL
 truename* NIL } ^{Fu}compile-file/by ^{Fu}load.

▷ Input file used by ^{Fu}compile-file/by ^{Fu}load.

^{var}*compile {
^{var}*load } {
 print*
 verbose* } ^{Fu}compile-file/by ^{Fu}load.

▷ Defaults used by ^{Fu}compile-file/by ^{Fu}load.

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

▷ Return values of forms if ^{so}eval-when is in the top-level of a file being compiled, in the top-level of a compiled file being loaded, or anywhere, respectively. Return NIL if *forms* are not evaluated. (**compile**, **load** and **eval** deprecated.)

(^{so}locally (declare decl*)^{*} *form*^{P*})

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

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

▷ Return values of forms. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of *forms*.

(^{so}load-time-value *form* [read-only NIL])

▷ Evaluate *form* at compile time and treat its value as literal at run time.

(^{so}quote foo) ▷ Return unevaluated foo.

(^{gF}make-load-form *foo* [*environment*])

▷ Its methods are to return a creation form which on evaluation at ^{Fu}load time returns an object equivalent to *foo*, and an optional initialization form which on evaluation performs some initialization of the object.

(^{Fu}make-load-form-saving-slots *foo* {
 :slot-names *slots* all local slots
 :environment *environment* })

▷ Return a creation form and an initialization form which on evaluation construct an object equivalent to *foo* with *slots* initialized with the corresponding values from *foo*.

(^{Fu}macro-function *symbol* [*environment*])

(^{Fu}compiler-macro-function {
name
 (setf *name*) } [*environment*])

▷ Return specified macro function, or compiler macro function, respectively, if any. Return NIL otherwise. **setfable**.

(^{Fu}eval *arg*)

▷ Return values of value of arg evaluated in global environment.

15.3 REPL and Debugging

```

var | var | var
+ | + | +
var | var | var
* | * | *
var | var | var
/ | / | /

```

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

^{var} - ▷ Form currently being evaluated by the REPL.

(^{Fu}**apropos** *string* [*package*_{NIL}])

▷ Print interned symbols containing *string*.

(^{Fu}**apropos-list** *string* [*package*_{NIL}])

▷ List of interned symbols containing *string*.

(^{Fu}**dribble** [*path*])

▷ Save a record of interactive session to file at *path*. Without *path*, close that file.

(^{Fu}**ed** [*file-or-function*_{NIL}])

▷ Invoke editor if possible.

(^{Fu}**macroexpand-1** } *form* [*environment*_{NIL}])
(^{Fu}**macroexpand** }

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

^{var}***macroexpand-hook***

▷ Function of arguments *expansion function*, macro form, and environment called by ^{Fu}**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.

^{var}***trace-output***

▷ Stream ^M**trace** and ^M**time** print their output on.

(^M**step** *form*)

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

(^{Fu}**break** [*control arg**])

▷ Jump directly into debugger; return NIL. See p. 36, ^{Fu}**format**, for *control* and *args*.

(^M**time** *form*)

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

(^{Fu}**inspect** *foo*)

▷ Interactively give information about *foo*.

(^{Fu}**describe** *foo* [*stream*_{^{var}*standard-output*}])

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

(^{EF}**describe-object** *foo* [*stream*])

▷ Send information about *foo* to *stream*. Not to be called by user.

(^{Fu}**disassemble** *function*)

▷ Send disassembled representation of *function* to ^{var}***standard-output***. Return NIL.

15.4 Declarations

 $(\overset{\text{Fu}}{\text{proclaim}} \text{decl})$ $(\overset{\text{M}}{\text{declaim}} \widehat{\text{decl}}^*)$

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

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

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

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

▷ Make *foos* names of declarations.

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

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

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

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

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

▷ Suppress warnings about used/unused bindings.

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

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

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

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

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

16 External Environment

 $(\overset{\text{Fu}}{\text{get-internal-real-time}})$ $(\overset{\text{Fu}}{\text{get-internal-run-time}})$

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

 $\overset{\text{CO}}{\text{internal-time-units-per-second}}$

▷ Number of clock ticks per second.

 $(\overset{\text{Fu}}{\text{encode-universal-time}} \text{sec } \text{min } \text{hour } \text{date } \text{month } \text{year } [\text{zone}_{\text{current}}])$ $(\overset{\text{Fu}}{\text{get-universal-time}})$

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

 $(\overset{\text{Fu}}{\text{decode-universal-time}} \text{universal-time } [\text{time-zone}_{\text{current}}])$ $(\overset{\text{Fu}}{\text{get-decoded-time}})$

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

 $(\overset{\text{Fu}}{\text{room}} [\{\text{NIL}|\text{default}|T\}])$

▷ Print information about internal storage management.

 $(\overset{\text{Fu}}{\text{short-site-name}})$ $(\overset{\text{Fu}}{\text{long-site-name}})$

▷ String representing physical location of computer.

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

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

 $(\overset{\text{Fu}}{\text{machine-instance}})$ ▷ Computer name.

Index

- " 33
- ' 33
- (33
- () 43
-) 33
- * 3, 30, 31, 40, 45
- ** 40, 45
- *** 45
- BREAK-ON-SIGNALS* 29
- COMPILE-FILE-PATHNAME* 44
- COMPILE-FILE-TRUENAME* 44
- COMPILE-PRINT* 44
- COMPILE-VERBOSE* 44
- DEBUG-IO* 39
- DEBUGGER-HOOK* 29
- DEFAULT-PATHNAME-DEFAULTS* 40
- ERROR-OUTPUT* 39
- FEATURES* 33
- GENSYM-COUNTER* 43
- LOAD-PATHNAME* 44
- LOAD-PRINT* 44
- LOAD-TRUENAME* 44
- LOAD-VERBOSE* 44
- MACROEXPAND-HOOK* 45
- MODULES* 42
- PACKAGE* 42
- PRINT-ARRAY* 35
- PRINT-BASE* 35
- PRINT-CASE* 35
- PRINT-CIRCLE* 35
- PRINT-ESCAPE* 35
- PRINT-GENSYM* 35
- PRINT-LENGTH* 35
- PRINT-LEVEL* 35
- PRINT-LINES* 35
- PRINT-MISER-WIDTH* 35
- PRINT-PPRINT-DISPATCH* 36
- PRINT-PRETTY* 35
- PRINT-RADIX* 35
- PRINT-READABLY* 35
- PRINT-RIGHT-MARGIN* 35
- QUERY-IO* 39
- RANDOM-STATE* 4
- READ-BASE* 32
- READ-DEFAULT-FLOAT-FORMAT* 32
- READ-EVAL* 33
- READ-SUPPRESS* 32
- READTABLE* 32
- STANDARD-INPUT* 39
- STANDARD-OUTPUT* 39
- TERMINAL-IO* 39
- TRACE-OUTPUT* 45
- + 3, 26, 45
- ++ 45
- +++ 45
- , 33
- .. 33
- ,@ 33
- 3, 45
- . 33
- / 3, 33, 45
- // 45
- /// 45
- /= 3
- : 41
- :: 41
- :ALLOW-OTHER-KEYS 19
- ; 33
- < 3
- <= 3
- = 3, 21
- > 3
- >= 3
- \ 33
- # 38
- #\ 33
- #' 33
- #(33
- ## 33
- #+ 33
- #- 33
- #. 33
- #: 33
- #< 33
- #= 33
- #A 33
- #B 33
- #C(33
- #O 33
- #P 33
- #R 33
- #S(33
- #X 33
- ## 33
- #| |# 33
- &ALLOW-OTHER-KEYS 19
- &AUX 19
- &BODY 19
- &ENVIRONMENT 19
- &KEY 19
- &OPTIONAL 19
- &REST 19
- &WHOLE 19
- ~(~) 37
- ~* 37
- ~/ / 38
- ~< ~> 37
- ~< ~> 37
- ~? 38
- ~A 36
- ~B 36
- ~C 36
- ~D 36
- ~E 36
- ~F 36
- ~G 36
- ~I 37
- ~O 36
- ~P 37
- ~R 36
- ~S 36
- ~T 37
- ~W 38
- ~X 36
- ~{ ~} 37
- ~\$ 36
- ~% 37
- ~& 37
- ~^ 37
- ~- 37
- ~| 37
- ~{ ~} 37
- ~* 37
- ~← 37
- ` 33
- || 33
- + 3
- 3
- ABORT 28
- ABOVE 21
- ABS 4
- ACONS 9
- ACOS 3
- ACOSH 4
- ACROSS 21
- ADD-METHOD 25
- ADJOIN 9
- ADJUST-ARRAY 10
- ADJUSTABLE-ARRAY-P 10
- ALLOCATE-INSTANCE 24
- ALPHA-CHAR-P 6
- ALPHANUMERICP 6
- ALWAYS 23
- AND 19, 21, 23, 26, 31, 33
- APPEND 9, 23, 26
- APPENDING 23
- APPLY 17
- APROPPOS 45
- APROPOS-LIST 45
- AREF 10
- ARITHMETIC-ERROR 30
- ARITHMETIC-ERROR-OPERANDS 29
- ARITHMETIC-ERROR-OPERATION 29
- ARRAY 30
- ARRAY-DIMENSION 11
- ARRAY-DIMENSION-LIMIT 11
- ARRAY-DIMENSIONS 11
- ARRAY-ELEMENT-TYPE 29
- ARRAY-DISPLACEMENT 11
- ARRAY-TOTAL-SIZE 11
- ARRAY-TOTAL-SIZE-LIMIT 11
- ARRAYP 10
- AS 21
- ASH 5
- ASIN 3
- ASINH 4
- ASSERT 28
- ASSOC 9
- ASSOC-IF 9
- ASSOC-IF-NOT 9
- ATAN 3
- ATANH 4
- ATOM 8, 30
- BASE-CHAR 30
- BASE-STRING 30
- BEING 21
- BELOW 21
- BIGNUM 30
- BIT 11, 30
- BIT-AND 11
- BIT-ANDC1 11
- BIT-ANDC2 11
- BIT-EQV 11
- BIT-IOR 11
- BIT-NAND 11
- BIT-NOR 11
- BIT-NOT 11
- BIT-ORC1 11
- BIT-ORC2 11
- BIT-VECTOR 30
- BIT-VECTOR-P 10
- BIT-XOR 11
- BLOCK 20
- BOOLE 4
- BOOLE-1 4
- BOOLE-2 4
- BOOLE-AND 5
- BOOLE-ANDC1 5
- BOOLE-ANDC2 5
- BOOLE-C1 4
- BOOLE-C2 4
- BOOLE-CLR 4
- BOOLE-EQV 5
- BOOLE-IOR 5
- BOOLE-NAND 5
- BOOLE-NOR 5
- BOOLE-ORC1 5
- BOOLE-ORC2 5
- BOOLE-SET 4
- BOOLE-XOR 5
- BOOLEAN 30
- BOTH-CASE-P 6
- BOUND-P 15
- BREAK 45
- BROADCAST-STREAM 30
- BROADCAST-STREAM-STREAMS 38
- BUILT-IN-CLASS 30
- BUTLAST 9
- BY 21
- BYTE 5
- BYTE-POSITION 5
- BYTE-SIZE 5
- CAAR 8
- CADR 8
- CALL-ARGUMENTS-LIMIT 17
- CALL-METHOD 27
- CALL-NEXT-METHOD 26
- CAR 8
- CASE 19
- CATCH 20
- CCASE 19
- CDAR 8
- CDDR 8
- CDR 8
- CEILING 4
- CELL-ERROR 30
- CELL-ERROR-NAME 29
- CERROR 27
- CHANGE-CLASS 24
- CHAR 8
- CHAR-CODE 7
- CHAR-CODE-LIMIT 7
- CHAR-DOWNCASE 7
- CHAR-EQUAL 6
- CHAR-GREATERP 7
- CHAR-INT 7
- CHAR-LESSP 7
- CHAR-NAME 7
- CHAR-NOT-EQUAL 6
- CHAR-NOT-GREATERP 7
- CHAR-NOT-LESSP 7
- CHAR-UPCASE 7
- CHAR/= 6
- CHAR< 6
- CHAR<= 6
- CHAR= 6
- CHAR> 6
- CHAR>= 6
- CHARACTER 7, 30, 33
- CHARACTERP 6
- CHECK-TYPE 29
- CIS 4
- CL 43
- CL-USER 43
- CLASS 30
- CLASS-NAME 24
- CLASS-OF 24
- CLEAR-INPUT 39
- CLEAR-OUTPUT 39
- CLOSE 39
- CLQR 1
- CLRHASH 14
- CODE-CHAR 7
- COERCE 29
- COLLECT 23
- COLLECTING 23
- COMMON-LISP 43
- COMMON-LISP-USER 43
- COMPILATION-SPEED 46
- COMPILE 43
- COMPILE-FILE 44
- COMPILE-FILE-PATHNAME 44
- COMPILED-FUNCTION 30
- COMPILED-FUNCTION-P 43
- COMPILER-MACRO 43
- COMPILER-MACRO-FUNCTION 44
- COMPLEMENT 17
- COMPLEX 4, 30, 33
- COMPLEXP 3
- COMPUTE-APPLICABLE-METHODS 25
- COMPUTE-RESTARTS 28
- CONCATENATE 12
- CONCATENATED-STREAM 30
- CONCATENATED-STREAM-STREAMS 38
- COND 19
- CONDITION 30
- CONJUGATE 4
- CONS 8, 30
- CONSP 8
- CONSTANTLY 17
- CONSTANTP 15
- CONTINUE 28
- CONTROL-ERROR 30
- COPY-ALIST 9
- COPY-LIST 9
- COPY-PPRINT-DISPATCH 36
- COPY-READTABLE 32
- COPY-SEQ 14
- COPY-STRUCTURE 15
- COPY-SYMBOL 43
- COPY-TREE 10
- COS 3
- COSH 3
- COUNT 12, 23
- COUNT-IF 12
- COUNT-IF-NOT 12
- COUNTING 23
- CTYPECASE 29
- DEBUG 46
- DECF 3
- DECLAIM 46
- DECLARATION 46
- DECLARE 46
- DECODE-FLOAT 6
- DECODE-UNIVERSAL-TIME 46
- DEFCLASS 23
- DEFCONSTANT 16
- DEFGeneric 25
- DEFINE-COMPILER-MACRO 18
- DEFINE-CONDITION 27
- DEFINE-METHOD-COMBINATION 26
- DEFINE-MODIFY-MACRO 19
- DEFINE-SETF-EXPANDER 18
- DEFINE-SYMBOL-MACRO 18
- DEFMACRO 18
- DEFMETHOD 25
- DEFPACKAGE 41
- DEFPARAMETER 16
- DEFSETF 18
- DEFSTRUCT 15
- DEFTYPE 31
- DEFUN 17
- DEFVAR 16
- DELETE 13
- DELETE-DUPLICATES 13
- DELETE-FILE 41
- DELETE-IF 13
- DELETE-IF-NOT 13
- DELETE-PACKAGE 42
- DENOMINATOR 4
- DEPOSIT-FIELD 5
- DESCRIBE 45
- DESCRIBE-OBJECT 45
- DESTRUCTURING-BIND 20
- DIGIT-CHAR 7
- DIGIT-CHAR-P 6
- DIRECTORY 41
- DIRECTORY-NAMESTRING 40
- DISASSEMBLE 45
- DIVISION-BY-ZERO 30
- DO 20, 23
- DO-ALL-SYMBOLS 42
- DO-EXTERNAL-SYMBOLS 42
- DO-SYMBOLS 42
- DO* 20
- DOCUMENTATION 43
- DOING 23
- DOLIST 21
- DOTIMES 21
- DOUBLE-FLOAT 30, 33
- DOUBLE-FLOAT-EPSILON 6

- DOUBLE-FLOAT-NEGATIVE-EPSILON 6
 DOWNFROM 21
 DOWNTO 21
 DPB 5
 DRIBBLE 45
 DYNAMIC-EXTENT 46
- EACH 21
 ECASE 19
 ECHO-STREAM 30
 ECHO-STREAM-INPUT-STREAM 38
 ECHO-STREAM-OUTPUT-STREAM 38
 ED 45
 EIGHTH 8
 ELSE 23
 ELT 12
 ENCODE-UNIVERSAL-TIME 46
 END 23
 END-OF-FILE 30
 ENDP 8
 ENOUGH-NAMESTRING 40
 ENSURE-DIRECTORIES-EXIST 41
 ENSURE-GENERIC-FUNCTION 25
 EQ 15
 EQL 15, 31
 EQUAL 15
 EQUALP 15
 ERROR 27, 30
 ETYPESCASE 29
 EVAL 44
 EVAL-WHEN 44
 EVENP 3
 EVERY 12
 EXP 3
 EXPORT 42
 EXPT 3
 EXTENDED-CHAR 30
 EXTERNAL-SYMBOL 21
 EXTERNAL-SYMBOLS 21
- FBOUNDP 16
 FCEILING 4
 FDEFINITION 17
 FFLOOR 4
 FIFTH 8
 FILE-AUTHOR 41
 FILE-ERROR 30
 FILE-ERROR-PATHNAME 29
 FILE-LENGTH 41
 FILE-NAMESTRING 40
 FILE-POSITION 39
 FILE-STREAM 30
 FILE-STRING-LENGTH 39
 FILE-WRITE-DATE 41
 FILL 12
 FILL-POINTER 11
 FINALLY 23
 FIND 13
 FIND-ALL-SYMBOLS 42
 FIND-CLASS 24
 FIND-IF 13
 FIND-IF-NOT 13
 FIND-METHOD 25
 FIND-PACKAGE 42
 FIND-RESTART 28
 FIND-SYMBOL 42
 FINISH-OUTPUT 39
 FIRST 8
 FIXNUM 30
 FLET 17
 FLOAT 4, 30
 FLOAT-DIGITS 6
 FLOAT-PRECISION 6
 FLOAT-RADIX 6
 FLOAT-SIGN 4
 FLOATING-POINT-INEXACT 30
 FLOATING-POINT-INVALID-OPERATION 30
 FLOATING-POINT-OVERFLOW 30
 FLOATING-POINT-UNDERFLOW 30
 FLOATP 3
 FLOOR 4
 FMAKUNBOUND 17
 FOR 21
 FORCE-OUTPUT 39
 FORMAT 36
 FORMATTER 36
 FOURTH 8
 FRESH-LINE 34
 FROM 21
 FROUND 4
 FTRUNCATE 4
 FTYPE 46
 FUNCCALL 17
 FUNCTION 17, 30, 33, 43
 FUNCTION-KEYWORDS 26
- FUNCTION-LAMBDA-EXPRESSION 17
 FUNCTIONP 15
- GCD 3
 GENERIC-FUNCTION 30
 GENSYM 43
 GENTEMP 43
 GET 16
 GET-DECODED-TIME 46
 GET-DISPATCH-MACRO-CHARACTER 32
 GET-INTERNAL-REAL-TIME 46
 GET-INTERNAL-RUN-TIME 46
 GET-MACRO-CHARACTER 32
 GET-OUTPUT-STREAM-STRING 38
 GET-PROPERTIES 16
 GET-SETF-EXPANSION 19
 GET-UNIVERSAL-TIME 46
 GETF 16
 GETHASH 14
 GO 20
 GRAPHIC-CHAR-P 6
- HANDLER-BIND 28
 HANDLER-CASE 28
 HASH-KEY 21
 HASH-KEYS 21
 HASH-TABLE 30
 HASH-TABLE-COUNT 14
 HASH-TABLE-P 14
 HASH-TABLE-REHASH-SIZE 14
 HASH-TABLE-REHASH-THRESHOLD 14
 HASH-TABLE-SIZE 14
 HASH-TABLE-TEST 14
 HASH-VALUE 21
 HASH-VALUES 21
 HOST-NAMESTRING 40
- IDENTITY 17
 IF 19, 23
 IGNORABLE 46
 IGNORE 46
 IGNORE-ERRORS 27
 IMAGPART 4
 IMPORT 42
 IN 21
 IN-PACKAGE 41
 INCF 3
 INITIALIZE-INSTANCE 24
 INITIALLY 23
 INLINE 46
 INPUT-STREAM-P 31
 INSPECT 45
 INTEGER 30
 INTEGER-DECODE-FLOAT 6
 INTEGER-LENGTH 5
 INTEGERP 3
 INTERACTIVE-STREAM-P 31
 INTERN 42
 INTERNAL-TIME-UNITS-PER-SECOND 46
 INTERSECTION 10
 INTO 23
 INVALID-METHOD-ERROR 26
 INVOKE-DEBUGGER 27
 INVOKE-RESTART 28
 INVOKE-RESTART-INTERACTIVELY 28
 ISQRT 3
 IT 23
- KEYWORD 30, 41, 43
 KEYWORDP 41
- LABELS 17
 LAMBDA 17
 LAMBDA-LIST-KEYWORDS 19
 LAMBDA-PARAMETERS-LIMIT 17
 LAST 8
 LCM 3
 LDB 5
 LDB-TEST 5
 LDIF 9
 LEAST-NEGATIVE-DOUBLE-FLOAT 6
 LEAST-NEGATIVE-LONG-FLOAT 6
 LEAST-NEGATIVE-NORMALIZED-DOUBLE-FLOAT 6
 LEAST-NEGATIVE-NORMALIZED-LONG-FLOAT 6
 LEAST-NEGATIVE-SHORT-FLOAT 6
 LEAST-POSITIVE-DOUBLE-FLOAT 6
 LEAST-POSITIVE-LONG-FLOAT 6
 LEAST-POSITIVE-NORMALIZED-DOUBLE-FLOAT 6
 LEAST-POSITIVE-NORMALIZED-LONG-FLOAT 6
 LEAST-POSITIVE-NORMALIZED-SHORT-FLOAT 6
 LEAST-POSITIVE-SINGLE-FLOAT 6
 LEAST-POSITIVE-SINGLE-FLOAT 6
 LENGTH 12
 LET 20
 LET* 20
 LISP-IMPLEMENTATION-TYPE 46
 LISP-IMPLEMENTATION-VERSION 46
 LIST 8, 26, 30
 LIST-ALL-PACKAGES 42
 LIST-LENGTH 8
 LIST* 8
 LISTEN 39
 LISTP 8
 LOAD 44
 LOAD-LOGICAL-PATHNAME-TRANSLATIONS 41
 LOAD-TIME-VALUE 44
 LOCALLY 44
 LOG 3
 LOGAND 5
 LOGANDC1 5
 LOGANDC2 5
 LOGBITP 5
 LOGCOUNT 5
 LOGEQV 5
 LOGICAL-PATHNAME 30, 40
 LOGICAL-PATHNAME-TRANSLATIONS 40
 LOGIOR 5
 LOGNAND 5
 LOGNOR 5
 LOGNOT 5
 LOGORC1 5
 LOGORC2 5
 LOGTEST 5
 LOGXOR 5
 LONG-FLOAT 30, 33
 LONG-FLOAT-EPSILON 6
 LONG-FLOAT-NEGATIVE-EPSILON 6
 LONG-SITE-NAME 46
 LOOP 21
 LOOP-FINISH 23
 LOWER-CASE-P 6
- MACHINE-INSTANCE 46
 MACHINE-TYPE 46
 MACHINE-VERSION 46
 MACRO-FUNCTION 44
 MACROEXPAND 45
 MACROEXPAND-1 45
 MACROLET 18
 MAKE-ARRAY 10
 MAKE-BROADCAST-STREAM 38
 MAKE-CONCATENATED-STREAM 38
 MAKE-CONDITION 27
 MAKE-DISPATCH-MACRO-CHARACTER 32
 MAKE-ECHO-STREAM 38
 MAKE-HASH-TABLE 14
 MAKE-INSTANCE 24
 MAKE-INSTANCES-OBSOLETE 24
 MAKE-LIST 8
 MAKE-LOAD-FORM 44
 MAKE-LOAD-FORM-SAVING-SLOTS 44
 MAKE-METHOD 27
 MAKE-PACKAGE 41
 MAKE-PATHNAME 40
 MAKE-RANDOM-STATE 4
 MAKE-SEQUENCE 12
 MAKE-STRING 7
- MAKE-STRING-INPUT-STREAM 38
 MAKE-STRING-OUTPUT-STREAM 38
 MAKE-SYMBOL 43
 MAKE-SYNONYM-STREAM 38
 MAKE-TWO-WAY-STREAM 38
 MAKUNBOUND 16
 MAP 14
 MAP-INTO 14
 MAPC 9
 MAPCAN 9
 MAPCAR 9
 MAPCON 9
 MAPHASH 14
 MAPL 9
 MAPLIST 9
 MASK-FIELD 5
 MAX 4, 26
 MAXIMIZE 23
 MAXIMIZING 23
 MEMBER 8, 31
 MEMBER-IF 8
 MEMBER-IF-NOT 8
 MERGE 12
 MERGE-PATHNAMES 40
 METHOD 30
 METHOD-COMBINATION 30, 43
 METHOD-COMBINATION-ERROR 26
 METHOD-QUALIFIERS 26
 MIN 4, 26
 MINIMIZE 23
 MINIMIZING 23
 MINUSP 3
 MISMATCH 12
 MOD 4, 31
 MOST-NEGATIVE-DOUBLE-FLOAT 6
 MOST-NEGATIVE-FIXNUM 6
 MOST-NEGATIVE-LONG-FLOAT 6
 MOST-NEGATIVE-SHORT-FLOAT 6
 MOST-NEGATIVE-SINGLE-FLOAT 6
 MOST-POSITIVE-DOUBLE-FLOAT 6
 MOST-POSITIVE-FIXNUM 6
 MOST-POSITIVE-LONG-FLOAT 6
 MOST-POSITIVE-SHORT-FLOAT 6
 MOST-POSITIVE-SINGLE-FLOAT 6
 MUFFLE-WARNING 28
 MULTIPLE-VALUE-BIND 20
 MULTIPLE-VALUE-CALL 17
 MULTIPLE-VALUE-LIST 17
 MULTIPLE-VALUE-PROG1 20
 MULTIPLE-VALUE-SETQ 16
 MULTIPLE-VALUES-LIMIT 17
- NAME-CHAR 7
 NAMED 21
 NAMESTRING 40
 NBUFLAST 9
 NCONC 9, 23, 26
 NCONCING 23
 NEVER 23
 NEWLINE 6
 NEXT-METHOD-P 25
 NIL 2, 43
 NINTERSECTION 10
 NINTH 8
 NO-APPLICABLE-METHOD 26
 NO-NEXT-METHOD 26
 NOT 15, 31, 33
 NOTANY 12
 NOTEVERY 12
 NOTINLINE 46
 NRECONC 9
 NREVERSE 12
 NSET-DIFFERENCE 10
 NSET-EXCLUSIVE-OR 10
 NSTRING-CAPITALIZE 7
 NSTRING-DOWNCASE 7
 NSTRING-UPCASE 7
 NSUBLIS 10
 NSUBST 10
 NSUBST-IF 10
 NSUBST-IF-NOT 10
 NSUBSTITUTE 13
 NSUBSTITUTE-IF 13
 NSUBSTITUTE-IF-NOT 13
 NTH 8
 NTH-VALUE 17

- NTHCDR 8
 NULL 8, 30
 NUMBER 30
 NUMBERP 3
 NUMERATOR 4
 NUNION 10
- ODDP 3
 OF 21
 OF-TYPE 21
 ON 21
 OPEN 38
 OPEN-STREAM-P 31
 OPTIMIZE 46
 OR 19, 26, 31, 33
 OTHERWISE 19, 29
 OUTPUT-STREAM-P 31
- PACKAGE 30
 PACKAGE-ERROR 30
 PACKAGE-ERROR-PACKAGE 29
 PACKAGE-NAME 42
 PACKAGE-NICKNAMES 42
 PACKAGE-SHADOWING-SYMBOLS 42
 PACKAGE-USE-LIST 42
 PACKAGE-USED-BY-LIST 42
 PACKAGEP 41
 PAIRLIS 9
 PARSE-ERROR 30
 PARSE-INTEGER 8
 PARSE-NAMESTRING 40
 PATHNAME 30, 40
 PATHNAME-DEVICE 40
 PATHNAME-DIRECTORY 40
 PATHNAME-HOST 40
 PATHNAME-MATCH-P 31
 PATHNAME-NAME 40
 PATHNAME-TYPE 40
 PATHNAME-VERSION 40
 PATHNAMEP 31
 PEEK-CHAR 32
 PHASE 4
 PI 3
 PLUSP 3
 POP 9
 POSITION 13
 POSITION-IF 13
 POSITION-IF-NOT 13
 PPRINT 34
 PPRINT-DISPATCH 36
 PPRINT-EXIT-IF-LIST-EXHAUSTED 35
 PPRINT-FILL 34
 PPRINT-INDENT 35
 PPRINT-LINEAR 34
 PPRINT-LOGICAL-BLOCK 35
 PPRINT-NEWLINE 35
 PPRINT-POP 35
 PPRINT-TAB 35
 PPRINT-TABULAR 34
 PRESENT-SYMBOL 21
 PRESENT-SYMBOLS 21
 PRIN1 34
 PRIN1-TO-STRING 34
 PRINC 34
 PRINC-TO-STRING 34
 PRINT 34
 PRINT-NOT-READABLE 30
 PRINT-NOT-READABLE-OBJECT 29
 PRINT-OBJECT 34
 PRINT-UNREADABLE-OBJECT 34
 PROBE-FILE 41
 PROCLAIM 46
 PROG 20
 PROG1 20
 PROG2 20
 PROG* 20
 PROG* 19, 26
 PROGRAM-ERROR 30
 PROGV 20
 PROVIDE 42
 PSETF 16
 PSETQ 16
 PUSH 9
 PUSHNEW 9
- QUOTE 33, 44
- RANDOM 4
 RANDOM-STATE 30
 RANDOM-STATE-P 3
 RASSOC 9
 RASSOC-IF 9
 RASSOC-IF-NOT 9
 RATIO 30, 33
 RATIONAL 4, 30
 RATIONALIZE 4
 RATIONALP 3
 READ 31
 READ-BYTE 32
 READ-CHAR 32
- READ-CHAR-NO-HANG 32
 READ-DELIMITED-LIST 32
 READ-FROM-STRING 31
 READ-LINE 32
 READ-PRESERVING-WHITESPACE 31
 READ-SEQUENCE 32
 READER-ERROR 30
 READTABLE 30
 READTABLE-CASE 32
 READTABLEP 31
 REAL 30
 REALP 3
 REALPART 4
 REDUCE 14
 REINITIALIZE-INSTANCE 24
 REM 4
 REMF 16
 REMHASH 14
 REMOVE 13
 REMOVE-DUPPLICATES 13
 REMOVE-IF 13
 REMOVE-IF-NOT 13
 REMOVE-METHOD 25
 REMPROP 16
 RENAME-FILE 41
 RENAME-PACKAGE 41
 REPEAT 23
 REPLACE 13
 REQUIRE 42
 REST 8
 RESTART 30
 RESTART-BIND 28
 RESTART-CASE 28
 RESTART-NAME 28
 RETURN 20, 23
 RETURN-FROM 20
 REVAPPEND 9
 REVERSE 12
 ROOM 46
 ROTATEF 16
 ROUND 4
 ROW-MAJOR-AREF 10
 RPLACA 9
 RPLACD 9
- SAFETY 46
 SATISFIES 31
 SBIT 11
 SCALE-FLOAT 6
 SCHAR 8
 SEARCH 13
 SECOND 8
 SEQUENCE 30
 SERIOUS-CONDITION 30
 SET 16
 SET-DIFFERENCE 10
 SET-DISPATCH-MACRO-CHARACTER 32
 SET-EXCLUSIVE-OR 10
 SET-MACRO-CHARACTER 32
 SET-PPRINT-DISPATCH 36
 SET-SYNTAX-FROM-CHAR 32
 SETF 16, 43
 SETQ 16
 SEVENTH 8
 SHADOW 42
 SHADOWING-IMPORT 42
 SHARED-INITIALIZE 24
 SHIFTF 16
 SHORT-FLOAT 30, 33
 SHORT-FLOAT-EPSILON 6
 SHORT-FLOAT-NEGATIVE-EPSILON 6
 SHORT-SITE-NAME 46
 SIGNAL 27
 SIGNED-BYTE 30
 SIGNUM 4
 SIMPLE-ARRAY 30
 SIMPLE-BASE-STRING 30
 SIMPLE-BIT-VECTOR 30
 SIMPLE-BIT-VECTOR-P 10
 SIMPLE-CONDITION 30
 SIMPLE-CONDITION-FORMAT-ARGUMENTS 29
 SIMPLE-CONDITION-FORMAT-CONTROL 29
 SIMPLE-ERROR 30
 SIMPLE-STRING 30
 SIMPLE-STRING-P 7
 SIMPLE-TYPE-ERROR 30
 SIMPLE-VECTOR 30
 SIMPLE-VECTOR-P 10
 SIMPLE-WARNING 30
 SIN 3
 SINGLE-FLOAT 30, 33
 SINGLE-FLOAT-EPSILON 6
- SINGLE-FLOAT-NEGATIVE-EPSILON 6
 SINH 3
 SIXTH 8
 SLEEP 20
 SLOT-BOUNDP 23
 SLOT-EXISTS-P 23
 SLOT-MAKUNBOUND 24
 SLOT-MISSING 24
 SLOT-UNBOUND 25
 SLOT-VALUE 24
 SOFTWARE-TYPE 46
 SOFTWARE-VERSION 46
 SOME 12
 SORT 12
 SPACE 6, 46
 SPECIAL-OPERATOR-P 43
 SPEED 46
 SQRT 3
 STABLE-SORT 12
 STANDARD 26
 STANDARD-CHAR 6, 30
 STANDARD-CHAR-P 6
 STANDARD-CLASS 30
 STANDARD-GENERIC-FUNCTION 30
 STANDARD-METHOD 30
 STANDARD-OBJECT 30
 STEP 45
 STORAGE-CONDITION 30
 STORE-VALUE 28
 STREAM 30
 STREAM-ELEMENT-TYPE 29
 STREAM-ERROR 30
 STREAM-ERROR-STREAM 29
 STREAM-EXTERNAL-FORMAT 39
 STREAMP 31
 STRING 7, 30
 STRING-CAPITALIZE 7
 STRING-DOWNCASE 7
 STRING-EQUAL 7
 STRING-GREATERP 7
 STRING-LEFT-TRIM 7
 STRING-LESSP 7
 STRING-NOT-EQUAL 7
 STRING-NOT-GREATERP 7
 STRING-NOT-LESSP 7
 STRING-RIGHT-TRIM 7
 STRING-STREAM 30
 STRING-TRIM 7
 STRING-UPCASE 7
 STRING/= 7
 STRING< 7
 STRING<= 7
 STRING= 7
 STRING> 7
 STRING>= 7
 STRINGP 7
 STRUCTURE 43
 STRUCTURE-CLASS 30
 STRUCTURE-OBJECT 30
 STYLE-WARNING 30
 SUBLIS 10
 SUBSEQ 12
 SUBSETP 8
 SUBST 10
 SUBST-IF 10
 SUBST-IF-NOT 10
 SUBSTITUTE 13
 SUBSTITUTE-IF 13
 SUBSTITUTE-IF-NOT 13
 SUBTYPEP 29
 SUM 23
 SUMMING 23
 SVREF 11
 SXHASH 14
 SYMBOL 21, 30, 43
 SYMBOL-FUNCTION 43
 SYMBOL-MACROLET 18
 SYMBOL-NAME 43
 SYMBOL-PACKAGE 43
 SYMBOL-PLIST 43
 SYMBOL-VALUE 43
 SYMBOLP 41
 SYMBOLS 21
 SYNONYM-STREAM 30
 SYNONYM-STREAM-SYMBOL 38
- T 2, 30, 43
 TAGBODY 20
 TAILP 8
 TAN 3
 TANH 3
 TENTH 8
 TERP1 34
 THE 21, 29
 THEN 21
 THEREIS 23
 THIRD 8
- THROW 20
 TIME 45
 TO 21
 TRACE 45
 TRANSLATE-LOGICAL-PATHNAME 41
 TRANSLATE-PATHNAME 40
 TREE-EQUAL 10
 TRUENAME 41
 TRUNCATE 4
 TWO-WAY-STREAM 30
 TWO-WAY-STREAM-INPUT-STREAM 38
 TWO-WAY-STREAM-OUTPUT-STREAM 38
 TYPE 43, 46
 TYPE-ERROR 30
 TYPE-ERROR-DATUM 29
 TYPE-ERROR-EXPECTED-TYPE 29
 TYPE-OF 29
 TYPECASE 29
 TYPEP 29
- UNBOUND-SLOT 30
 UNBOUND-SLOT-INSTANCE 29
 UNBOUND-VARIABLE 30
 UNDEFINED-FUNCTION 30
 UNEXPORT 42
 UNINTERN 42
 UNION 10
 UNLESS 19, 23
 UNREAD-CHAR 32
 UNSIGNED-BYTE 30
 UNTIL 23
 UNTRACE 45
 UNUSE-PACKAGE 41
 UNWIND-PROTECT 20
 UPDATE-INSTANCE-FOR-DIFFERENT-CLASS 24
 UPDATE-INSTANCE-FOR-REDEFINED-CLASS 24
 UPFROM 21
 UPGRADED-ARRAY-ELEMENT-TYPE 31
 UPGRADED-COMPLEX-PART-TYPE 6
 UPPER-CASE-P 6
 UPTO 21
 USE-PACKAGE 41
 USE-VALUE 28
 USER-HOMEDIR-PATHNAME 40
 USING 21
- V 38
 VALUES 17, 31
 VALUES-LIST 17
 VARIABLE 43
 VECTOR 11, 30
 VECTOR-POP 11
 VECTOR-PUSH 11
 VECTOR-PUSH-EXTEND 11
 VECTORP 10
- WARN 27
 WARNING 30
 WHEN 19, 23
 WHILE 23
 WILD-PATHNAME-P 31
 WITH 21
 WITH-ACCESSORS 24
 WITH-COMPILED-UNIT 44
 WITH-CONDITION-RESTARTS 29
 WITH-HASH-TABLE-ITERATOR 14
 WITH-INPUT-FROM-STRING 39
 WITH-OPEN-FILE 39
 WITH-OPEN-STREAM 39
 WITH-OUTPUT-TO-STRING 39
 WITH-PACKAGE-ITERATOR 42
 WITH-SIMPLE-RESTART 28
 WITH-SLOTS 24
 WITH-STANDARD-IO-SYNTAX 31
 WRITE 34
 WRITE-BYTE 34
 WRITE-CHAR 34
 WRITE-LINE 34
 WRITE-SEQUENCE 34
 WRITE-STRING 34
 WRITE-TO-STRING 34
- Y-OR-N-P 31
 YES-OR-NO-P 31
- ZEROP 3

