Quick Reference

Common lisp

Bert Burgemeister

Common Lisp Quick Reference Revision 130 [2011-10-12]

LaTeX source: http://clqr.boundp.org

Contents

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.

them ▷ Placeholder for actual code.

`me` ▷ Literal text.

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

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

foo$^+$; {*foo*}$^+$ ▷ One or more *foo*s.

foos ▷ English plural denotes a list argument.

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

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

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

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

foo$^{\text{P}*}$ ▷ *foo** is evaluated as in **sO progn**; see p. 19.

$\underline{foo}$; $\underline{\underset{2}{bar}}$; $\underline{\underset{n}{baz}}$ ▷ Primary, secondary, and nth return value.

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

1 Numbers

1.1 Predicates

(Fu **=** *number*$^+$)
(Fu **/=** *number*$^+$)
▷ T if all *number*s, or none, respectively, are equal in value.

(Fu **>** *number*$^+$)
(Fu **>=** *number*$^+$)
(Fu **<** *number*$^+$)
(Fu **<=** *number*$^+$)
▷ Return T if *number*s are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively.

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

(Fu **evenp** *integer*)
(Fu **oddp** *integer*)
▷ T if *integer* is even or odd, respectively.

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

1.2 Numeric Functions

(Fu **+** *a*[0]*)
(Fu ***** *a*[1]*)
▷ Return $\sum a$ or $\prod a$, respectively.

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

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

({M **incf** | M **decf**} *place* [*delta*[1]])
▷ Increment or decrement the value of *place* by *delta*. Return new value.

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

(Fu **log** *a* [*b*])
▷ Return $\log_b a$ or, without *b*, $\ln a$.

(Fu **sqrt** *n*)
(Fu **isqrt** *n*)
▷ $\sqrt{n}$ in complex or natural numbers, respectively.

(Fu **lcm** *integer**[1])
(Fu **gcd** *integer**)
▷ Least common multiple or greatest common denominator, respectively, of *integer*s. (**gcd**) returns 0.

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

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

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

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

(Fu **sinh** *a*)
(Fu **cosh** *a*)
(Fu **tanh** *a*)
▷ $\sinh a$, $\cosh a$, or $\tanh a$, respectively.

(Fu **asinh** *a*)
(Fu **acosh** *a*)
(Fu **atanh** *a*)
▷ asinh *a*, acosh *a*, or atanh *a*, respectively.

(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 *num*s.

({ Fu **round** | Fu **fround** } / { Fu **floor** | Fu **ffloor** } / { Fu **ceiling** | Fu **fceiling** } / { Fu **truncate** | Fu **ftruncate** } *n* [*d*1])
▷ 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 Fu **floor** or Fu **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}NIL])
▷ 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*1]) ▷ *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*0]) ▷ 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*. *operation*s are

co **boole-1**	▷ *int-a*.
co **boole-2**	▷ *int-b*.
co **boole-c1**	▷ ¬*int-a*.
co **boole-c2**	▷ ¬*int-b*.
co **boole-set**	▷ All bits set.
co **boole-clr**	▷ All bits zero.

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

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

(**logeqv** *integer**)
(**logand** *integer**)
▷ Return value of exclusive-nored or anded *integer*s, respectively. Without any *integer*, return -1.

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

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

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

(**logxor** *integer**)
(**logior** *integer**)
▷ Return value of exclusive-ored or ored *integer*s, respectively. Without any *integer*, return 0.

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

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

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

(**logbitp** *i integer*)
▷ T if zero-indexed ith bit of *integer* is set.

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

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

1.4 Integer Functions

(**integer-length** *integer*)
▷ Number of bits necessary to represent *integer*.

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

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

(**ldb** *byte-spec integer*)
▷ Extract byte denoted by *byte-spec* from *integer*. **setf**able.

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

(**mask-field** *byte-spec integer*)
▷ Return copy of *integer* with all bits unset but those denoted by *byte-spec*. **setf**able.

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

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

1.5 Implementation-Dependent

co **short-float**, co **single-float**, co **double-float**, co **long-float** – **epsilon**, **negative-epsilon**

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

co **least-negative**, co **least-negative-normalized**, co **least-positive**, co **least-positive-normalized** – **short-float**, **single-float**, **double-float**, **long-float**

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

co **most-negative**, co **most-positive** – **short-float**, **single-float**, **double-float**, **long-float**, **fixnum**

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

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

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

(Fu **scale-float** *n* [*i*]) ▷ With *n*'s radix *b*, return nb^i.

(Fu **float-radix** *n*)
(Fu **float-digits** *n*)
(Fu **float-precision** *n*)

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

(Fu **upgraded-complex-part-type** *foo* [*environment*NIL])

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

2 Characters

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

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

(Fu **graphic-char-p** *character*)
(Fu **alpha-char-p** *character*)
(Fu **alphanumericp** *character*)

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

(Fu **upper-case-p** *character*)
(Fu **lower-case-p** *character*)
(Fu **both-case-p** *character*)

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

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

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

(Fu **char=** *character*$^+$)
(Fu **char/=** *character*$^+$)

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

(Fu **char-equal** *character*$^+$)
(Fu **char-not-equal** *character*$^+$)

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

(Fu **char>** *character*$^+$)
(Fu **char>=** *character*$^+$)
(Fu **char<** *character*$^+$)
(Fu **char<=** *character*$^+$)

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

Index

15.4 Declarations

(Fu **proclaim** *decl*)
(M **declaim** $\widehat{decl^*}$)
▷ Globally make declaration(s) *decl*. *decl* can be: **declaration**, **type**, **ftype**, **inline**, **notinline**, **optimize**, or **special**. See below.

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

(**declaration** foo*)
▷ Make *foo*s names of declarations.

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

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

({**ignorable** | **ignore**} { *var* | (sO **function** *function*) }*)
▷ Suppress warnings about used/unused bindings.

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

(**optimize** { **compilation-speed** | (**compilation-speed** $n_{\boxed{3}}$) | **debug** | (**debug** $n_{\boxed{3}}$) | **safety** | (**safety** $n_{\boxed{3}}$) | **space** | (**space** $n_{\boxed{3}}$) | **speed** | (**speed** $n_{\boxed{3}}$) })
▷ Tell compiler how to optimize. $n = 0$ means unimportant, $n = 1$ is neutral, $n = 3$ means important.

(**special** *var**) ▷ Declare *var*s to be dynamic.

16 External Environment

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

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

(Fu **encode-universal-time** *sec min hour date month year* [*zone*$_{\boxed{\text{curr.}}}$])
(Fu **get-universal-time**)
▷ Seconds from 1900-01-01, 00:00, ignoring leap seconds.

(Fu **decode-universal-time** *universal-time* [*time-zone*$_{\boxed{\text{current}}}$])
(Fu **get-decoded-time**)
▷ Return second$_{8}$, minute$_{2}$, hour$_{3}$, date$_{4}$, month$_{5}$, year$_{6}$, day$_{7}$, daylight-p$_{8}$, and zone$_{9}$.

(Fu **room** [{NIL | **:default** | T}])
▷ Print information about internal storage management.

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

({ Fu **lisp-implementation** | Fu **software** | Fu **machine** } - { **type** | **version** })
▷ Name or version of implementation, operating system, or hardware, respectively.

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

(Fu **char-greaterp** *character*$^+$)
(Fu **char-not-lessp** *character*$^+$)
(Fu **char-lessp** *character*$^+$)
(Fu **char-not-greaterp** *character*$^+$)
▷ Return T if *character*s 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*$_{\boxed{10}}$]) ▷ 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*.

co **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=** | Fu **string-equal** } *foo bar* { **:start1** *start-foo*$_{\boxed{0}}$ | **:start2** *start-bar*$_{\boxed{0}}$ | **:end1** *end-foo*$_{\boxed{\text{NIL}}}$ | **:end2** *end-bar*$_{\boxed{\text{NIL}}}$ })
▷ Return T if subsequences of *foo* and *bar* are equal. Obey/ignore, respectively, case.

({ Fu **string{/= | -not-equal}** | Fu **string{> | -greaterp}** | Fu **string{>= | -not-lessp}** | Fu **string{< | -lessp}** | Fu **string{<= | -not-greaterp}** } *foo bar* { **:start1** *start-foo*$_{\boxed{0}}$ | **:start2** *start-bar*$_{\boxed{0}}$ | **:end1** *end-foo*$_{\boxed{\text{NIL}}}$ | **:end2** *end-bar*$_{\boxed{\text{NIL}}}$ })
▷ 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*$_{\boxed{\text{character}}}$ })
▷ Return string of length *size*.

(Fu **string** *x*)
({ Fu **string-capitalize** | Fu **string-upcase** | Fu **string-downcase** } *x* { **:start** *start*$_{\boxed{0}}$ | **:end** *end*$_{\boxed{\text{NIL}}}$ })
▷ 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** | Fu **nstring-upcase** | Fu **nstring-downcase** } $\widetilde{string}$ { **:start** *start*$_{\boxed{0}}$ | **:end** *end*$_{\boxed{\text{NIL}}}$ })
▷ Convert *string* into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

({ Fu **string-trim** | Fu **string-left-trim** | 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. **setf**able.

(Fu **parse-integer** *string* { **:start** *start*[0] | **:end** *end*[NIL] | **:radix** *int*[10] | **:junk-allowed** *bool*[NIL] })
▷ 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* { **:test** *function*[#'eql] | **:test-not** *function* | **:key** *function* })
▷ Return tail of *list* starting with its first element matching *foo*. Return NIL if there is no such element.

({Fu **member-if** | Fu **member-if-not**} *test list* [**:key** *function*])
▷ Return tail of *list* starting with its first element satisfying *test*. Return NIL if there is no such element.

(Fu **subsetp** *list-a list-b* { **:test** *function*[#'eql] | **:test-not** *function* | **:key** *function* })
▷ 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 *foo*s.

(Fu **list*** *foo*$^+$)
▷ Return list of *foo*s 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. **setf**able.

(Fu **cdr** *list*)
(Fu **rest** *list*) ▷ Cdr of *list* or NIL if *list* is NIL. **setf**able.

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

({Fu **first**|Fu **second**|Fu **third**|Fu **fourth**|Fu **fifth**|Fu **sixth**|...|Fu **ninth**|Fu **tenth**} *list*)
▷ Return nth element of *list* if any, or NIL otherwise. **setf**able.

(Fu **nth** *n list*) ▷ Zero-indexed nth element of *list*. **setf**able.

(Fu **c***X***r** *list*)
▷ With *X* being one to four **a**s and **d**s representing Fu **car**s and Fu **cdr**s, e.g. (Fu **cadr** *bar*) is equivalent to (Fu **car** (Fu **cdr** *bar*)). **setf**able.

(Fu **last** *list* [*num*[1]]) ▷ Return list of last *num* conses of *list*.

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** | Fu **macroexpand**} *form* [*environment*[NIL]])
▷ Return macro expansion, once or entirely, respectively, of *form* and T if *form* was a macro form. Return *form* and NIL otherwise.

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 *function*s to be traced. With no arguments, return list of traced functions.

(M **untrace** {*function* | (**setf** *function*)}*)
▷ Stop *function*s, 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 *arg*s.

(M **time** *form*)
▷ Evaluate *form*s 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*.

(gF **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.

(Fu **compile-file** *file* { **:output-file** *out-path* | **:verbose** *bool* [v ***compile-verbose***] | **:print** *bool* [v ***compile-print***] | **:external-format** *file-format* [**:default**] })

▷ Write compiled contents of *file* to *out-path*. Return true output path or NIL, T in case of warnings or errors, T (2) 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* [v ***load-verbose***] | **:print** *bool* [v ***load-print***] | **:if-does-not-exist** *bool* [T] | **:external-format** *file-format* [**:default**] })

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

v ***compile-file*** / v ***load*** - { **pathname*** [NIL] | **truename*** [NIL] }

▷ Input file used by Fu **compile-file**/by Fu **load**.

v ***compile*** / v ***load*** - { **print*** | **verbose*** }

▷ Defaults used by Fu **compile-file**/by Fu **load**.

(sO **eval-when** ({ {**:compile-toplevel** | **compile**} | {**:load-toplevel** | **load**} | {**:execute** | **eval**} }) *form**)

▷ 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 *form*s are not evaluated. (**compile**, **load** and **eval** deprecated.)

(sO **locally** (**declare** *decl**)* *form**)

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

(M **with-compilation-unit** ([**:override** *bool* [NIL]]) *form**)

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

(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 (2) 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 (2) 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. **setf**able.

(Fu **eval** *arg*)

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

(Fu { **butlast** | **nbutlast** } *list* [*num* [1]]) ▷ *list* excluding last *num* conses.

(Fu { **rplaca** | **rplacd** } *cons object*)

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

(Fu **ldiff** *list foo*)

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

(Fu **adjoin** *foo list* { **:test** *function* [#'eql] | **:test-not** *function* | **:key** *function* })

▷ Return *list* if *foo* is already member of *list*. If not, return (Fu **cons** *foo list*).

(M **pop** *place*) ▷ Set *place* to (Fu **cdr** *place*), return (Fu **car** *place*).

(M **push** *foo place*) ▷ Set *place* to (Fu **cons** *foo place*).

(M **pushnew** *foo place* { **:test** *function* [#'eql] | **:test-not** *function* | **:key** *function* })

▷ Set *place* to (Fu **adjoin** *foo place*).

(Fu **append** [*proper-list** *foo* [NIL]])

(Fu **nconc** [*non-circular-list** *foo* [NIL]])

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

(Fu **revappend** *list foo*)

(Fu **nreconc** *list foo*)

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

(Fu { **mapcar** | **maplist** } *function list*+)

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

(Fu { **mapcan** | **mapcon** } *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.

(Fu { **mapc** | **mapl** } *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.

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

4.3 Association Lists

(Fu **pairlis** *keys values* [*alist* [NIL]])

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

(Fu **acons** *key value alist*)

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

(Fu { **assoc** | **rassoc** } *foo alist* { **:test** *test* [#'eql] | **:test-not** *test* | **:key** *function* })

(Fu { **assoc-if**[**-not**] | **rassoc-if**[**-not**] } *test alist* [**:key** *function*])

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

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

4.4 Trees

(Fu **tree-equal** *foo bar* {**:test** *test* #'eql | **:test-not** *test*})
▷ Return T if trees *foo* and *bar* have same shape and leaves satisfying *test*.

({Fu **subst** *new old tree* | Fu **nsubst** *new old tree*} {| **:test** *function* #'eql | **:test-not** *function* | **:key** *function* |})
▷ Make copy of *tree* with each subtree or leaf matching *old* replaced by *new*.

({Fu **subst-if**[**-not**] *new test tree* | Fu **nsubst-if**[**-not**] *new test tree*} [**:key** *function*])
▷ Make copy of *tree* with each subtree or leaf satisfying *test* replaced by *new*.

({Fu **sublis** *association-list tree* | Fu **nsublis** *association-list tree*} {| **:test** *function* #'eql | **:test-not** *function* | **:key** *function* |})
▷ Make copy of *tree* with each subtree or leaf matching a key in *association-list* replaced by that key's value.

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

4.5 Sets

({Fu **intersection** | Fu **set-difference** | Fu **union** | Fu **set-exclusive-or**} *a b* | {Fu **nintersection** | Fu **nset-difference**} $\widetilde{a}$ *b* | {Fu **nunion** | Fu **nset-exclusive-or**} $\widetilde{a}$ $\widetilde{b}$ } {| **:test** *function* #'eql | **:test-not** *function* | **:key** *function* |})
▷ Return $a \cap b$, $a \setminus b$, $a \cup b$, or $a \triangle b$, respectively, of lists *a* and *b*.

5 Arrays

5.1 Predicates

(Fu **arrayp** *foo*)
(Fu **vectorp** *foo*)
(Fu **simple-vector-p** *foo*) ▷ T if *foo* is of indicated type.
(Fu **bit-vector-p** *foo*)
(Fu **simple-bit-vector-p** *foo*)

(Fu **adjustable-array-p** *array*)
(Fu **array-has-fill-pointer-p** *array*)
▷ T if *array* is adjustable/has a fill pointer, respectively.

(Fu **array-in-bounds-p** *array* [*subscripts*])
▷ Return T if *subscripts* are in *array*'s bounds.

5.2 Array Functions

({Fu **make-array** *dimension-sizes* [**:adjustable** *bool* NIL] | Fu **adjust-array** $\widetilde{array}$ *dimension-sizes*}
{| **:element-type** *type* T | **:fill-pointer** {*num* | *bool*} NIL | {| **:initial-element** *obj* | **:initial-contents** *sequence* | **:displaced-to** *array* NIL [**:displaced-index-offset** *i* 0] |} |})
▷ Return fresh, or readjust, respectively, vector or array.

(Fu **aref** *array* [*subscripts*])
▷ Return array element pointed to by *subscripts*. **setf**able.

(Fu **row-major-aref** *array i*)
▷ Return *i*th element of *array* in row-major order. **setf**able.

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* 0])
▷ Return fresh, uninterned symbol #:*sn* with *n* from var ***gensym-counter***. Increment var ***gensym-counter***.

(Fu **gentemp** [*prefix* T [*package* var ***package***]])
▷ Intern fresh symbol in package. Deprecated.

(Fu **copy-symbol** *symbol* [*props* NIL])
▷ 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*. **setf**able.

({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***; 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** {NIL *definition* | {*name* | (**setf** *name*)} [*definition*]})
▷ Return compiled function or replace *name*'s function definition with the compiled function. Return T[2] in case of warnings or errors, and T[3] in case of warnings or errors excluding style warnings.

(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.

(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 **bit-eqv** / Fu **bit-and** / Fu **bit-andc1** / Fu **bit-andc2** / Fu **bit-nand** / Fu **bit-ior** / Fu **bit-orc1** / Fu **bit-orc2** / Fu **bit-xor** / Fu **bit-nor**} *bit-array-a bit-array-b* [*result-bit-array*NIL])
▷ Return result of bitwise logical operations (cf. operations of Fu **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 *foo*s.

(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

(Fu {**every** | **notevery**} *test sequence*$^+$)
▷ Return NIL or T, respectively, as soon as *test* on any set of corresponding elements of *sequence*s returns NIL.

(Fu {**some** | **notany**} *test sequence*$^+$)
▷ Return value of *test* or NIL, respectively, as soon as *test* on any set of corresponding elements of *sequence*s returns non-NIL.

(Fu **mismatch** *sequence-a sequence-b* { **:from-end** *bool*$_{NIL}$ | {**:test** *function*$_{\#'eql}$ | **:test-not** *function*} | **:start1** *start-a*$_0$ | **:start2** *start-b*$_0$ | **:end1** *end-a*$_{NIL}$ | **:end2** *end-b*$_{NIL}$ | **:key** *function* })
▷ Return position in *sequence-a* where *sequence-a* and *sequence-b* begin to mismatch. Return NIL if they match entirely.

6.2 Sequence Functions

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

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

(Fu **merge** *type* $\widetilde{sequence}$*-a* $\widetilde{sequence}$*-b test* [**:key** *function*$_{NIL}$])
▷ Return interleaved sequence of *type*. Merged sequence will be sorted if both *sequence-a* and *sequence-b* are sorted.

(Fu **fill** $\widetilde{sequence}$ *foo* { **:start** *start*$_0$ | **:end** *end*$_{NIL}$ })
▷ Return *sequence* after setting elements between *start* and *end* to *foo*.

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

(Fu **count** *foo sequence* { **:from-end** *bool*$_{NIL}$ | {**:test** *function*$_{\#'eql}$ | **:test-not** *function*} | **:start** *start*$_0$ | **:end** *end*$_{NIL}$ | **:key** *function* })
▷ Return number of elements in *sequence* which match *foo*.

(Fu {**count-if** | **count-if-not**} *test sequence* { **:from-end** *bool*$_{NIL}$ | **:start** *start*$_0$ | **:end** *end*$_{NIL}$ | **:key** *function* })
▷ Return number of elements in *sequence* which satisfy *test*.

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

(Fu **subseq** *sequence start* [*end*$_{NIL}$])
▷ Return subsequence of *sequence* between *start* and *end*. **setf**able.

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

(Fu **reverse** *sequence*)
(Fu **nreverse** $\widetilde{sequence}$) ▷ Return *sequence* in reverse order.

(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$_2$, and new physical file name$_3$.

(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$_2$ 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-symbol*s, symbols from *used-package*s, *imported-symbol*s, and *shd-symbol*s. Add *shd-symbol*s 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** | **unuse-package**} *other-packages* [*package*$_{*package*}$])
▷ Make exported symbols of *other-packages* available in *package*, or remove them from *package*, respectively. Return T.

13.7 Pathnames and Files

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

▷ 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**|**pathname-device**|**pathname-directory**|**pathname-name**|**pathname-type**} *path* [**:case** {**:local**|**:common**}[**:local**]])

(Fu **pathname-version** *path*)

▷ Return pathname component.

(Fu **parse-namestring** *foo* [*host* [*default-pathname*[var ***default-pathname-defaults***] {**:start** *start*[0]|**:end** *end*[NIL]|**:junk-allowed** *bool*[NIL]}]])

▷ Return pathname converted from string, pathname, or stream *foo*; and position$_2$ where parsing stopped.

(Fu **merge-pathnames** *pathname* [*default-pathname*[var ***default-pathname-defaults***] [*default-version*[**:newest**]]])

▷ 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*[var ***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*. **setf**able.

({Fu **find**|**position**} *foo sequence* {**:from-end** *bool*[NIL]|{**:test** *function*[#'eql]|**:test-not** *test*}|**:start** *start*[0]|**:end** *end*[NIL]|**:key** *function*})

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

({Fu **find-if**|**find-if-not**|**position-if**|**position-if-not**} *test sequence* {**:from-end** *bool*[NIL]|**:start** *start*[0]|**:end** *end*[NIL]|**:key** *function*})

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

(Fu **search** *sequence-a sequence-b* {**:from-end** *bool*[NIL]|{**:test** *function*[#'eql]|**:test-not** *function*}|**:start1** *start-a*[0]|**:start2** *start-b*[0]|**:end1** *end-a*[NIL]|**:end2** *end-b*[NIL]|**:key** *function*})

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

({Fu **remove** *foo sequence*|**delete** *foo sequence*} {**:from-end** *bool*[NIL]|{**:test** *function*[#'eql]|**:test-not** *function*}|**:start** *start*[0]|**:end** *end*[NIL]|**:key** *function*|**:count** *count*[NIL]})

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

({Fu **remove-if**|**remove-if-not** *test sequence*|**delete-if**|**delete-if-not** *test sequence*} {**:from-end** *bool*[NIL]|**:start** *start*[0]|**:end** *end*[NIL]|**:key** *function*|**:count** *count*[NIL]})

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

({Fu **remove-duplicates**|**delete-duplicates** *sequence*} {**:from-end** *bool*[NIL]|{**:test** *function*[#'eql]|**:test-not** *function*}|**:start** *start*[0]|**:end** *end*[NIL]|**:key** *function*})

▷ Make copy of *sequence* without duplicates.

({Fu **substitute**|**nsubstitute** *new old sequence*} {**:from-end** *bool*[NIL]|{**:test** *function*[#'eql]|**:test-not** *function*}|**:start** *start*[0]|**:end** *end*[NIL]|**:key** *function*|**:count** *count*[NIL]})

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

({Fu **substitute-if**|**substitute-if-not**|**nsubstitute-if**|**nsubstitute-if-not** *new test sequence*} {**:from-end** *bool*[NIL]|**:start** *start*[0]|**:end** *end*[NIL]|**:key** *function*|**:count** *count*[NIL]})

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

(Fu **replace** *sequence-a sequence-b* {**:start1** *start-a*[0]|**:start2** *start-b*[0]|**:end1** *end-a*[NIL]|**:end2** *end-b*[NIL]})

▷ 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* { **:initial-value** *foo*$_{\texttt{NIL}}$ | **:from-end** *bool*$_{\texttt{NIL}}$ | **:start** *start*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ | **:key** *function* }*)
▷ 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** { **:test** {Fu **eq**|Fu **eql**|Fu **equal**|Fu **equalp**}$_{\#\text{'eql}}$ | **:size** *int* | **:rehash-size** *num* | **:rehash-threshold** *num* }*)
▷ Make a hash table.

(Fu **gethash** *key hash-table* [*default*$_{\texttt{NIL}}$])
▷ Return object with *key* if any or *default* otherwise; and `T` if found, `NIL` otherwise. **setf**able.

(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** *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 Fu **make-hash-table**.

(Fu **sxhash** *foo*)
▷ Hash code unique for any argument Fu **equal** *foo*.

(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*$_{\text{var *standard-input*}}$])
▷ `T` if there is a character in input *stream*.

(Fu **clear-input** [*stream*$_{\text{var *standard-input*}}$])
▷ Clear input from *stream*, return `NIL`.

({ Fu **clear-output** | Fu **force-output** | Fu **finish-output** } [*stream*$_{\text{var *standard-output*}}$])
▷ End output to *stream* and return `NIL` immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.

(Fu **close** *stream* [**:abort** *bool*$_{\texttt{NIL}}$])
▷ 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*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ }) (**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*$_{\texttt{NIL}}$ [**:element-type** *type*$_{\text{character}}$]]) (**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.

~ [**@**] **?**
▷ **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* :[:]`CL-USER:`]*function*/
▷ **Call Function.** Call all-uppercase *package*::*function* with the arguments stream, format-argument, colon-p, at-sign-p and *prefix*es 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

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

)
▷ Open **file-stream** to *path*.

(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*`0` [*end*`NIL`]])
▷ Return a **string-stream** supplying the characters from *string*.

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

8 Structures

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

[*doc*]
- *slot*
- (*slot* [*init* | **:type** *slot-type** | **:read-only** *bool*])

)
▷ Define structure *foo* together with functions `MAKE-`*foo*, `COPY-`*foo* and *foo*`-P`; and **setf**able 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, *arg*s and **:***key*s correspond to the positional and keyword parameters defined in *ord-λ* whose *var*s in turn correspond to *slot*s. **:print-object**/**:print-function** generate a gf **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 **eql** *foo bar*)
▷ `T` if *foo* and *bar* are identical, or the same **character**, or **number**s of the same type and value.

(Fu **equal** *foo bar*)
▷ `T` if *foo* and *bar* are Fu **eql**, or are equivalent **pathname**s, or are **cons**es with Fu **equal** cars and cdrs, or are **string**s or **bit-vector**s with Fu **eql** 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 **number**s of the same value ignoring type; or are equivalent **pathname**s; or are **cons**es or **array**s of the same shape with Fu **equalp** elements; or are structures of the same type with Fu **equalp** elements; or are **hash-table**s of the same size with the same **:test** function, the same keys in terms of **: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** {*foo* | (**setf** *foo*)}) ▷ T if *foo* is a global function or macro.

9.2 Variables

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

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

({M **setf** | M **psetf**} {*place form*}*)
▷ Set *place*s to primary values of *form*s. Return values of last *form*/NIL; work sequentially/in parallel, respectively.

({sO **setq** | M **psetq**} {*symbol form*}*)
▷ Set *symbol*s to primary values of *form*s. Return value of last *form*/NIL; work sequentially/in parallel, respectively.

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

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

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

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

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

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

(Fu **get-properties** *property-list keys*)
▷ Return key and value$_2$ of first entry from *property-list* matching a key from *keys*, and tail of *property-list*$_3$ starting with that key. Return NIL, NIL$_2$, and NIL$_3$ if there was no matching key in *property-list*.

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

9.3 Functions

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

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

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

{**~(** *text* **~)**|**~:(** *text* **~)**|**~@(** *text* **~)**|**~:@(** *text* **~)**}
▷ **Case-Conversion.** Convert *text* to lowercase, convert first letter of each word to uppercase, capitalize first word and convert the rest to lowercase, or convert to uppercase, respectively.

{**~P**|**~:P** |**~@P**|**~:@P**}
▷ **Plural.** If argument **eql** 1 print nothing, otherwise print s; do the same for the previous argument; if argument **eql** 1 print y, otherwise print ies; do the same for the previous argument, respectively.

~ [$n_{\boxed{1}}$] **%** ▷ **Newline.** Print n newlines.

~ [$n_{\boxed{1}}$] **&**
▷ **Fresh-Line.** Print $n-1$ newlines if output stream is at the beginning of a line, or n newlines otherwise.

{**~_**|**~:_**|**~@_**|**~:@_**}
▷ **Conditional Newline.** Print a newline like **pprint-newline** with argument **:linear**, **:fill**, **:miser**, or **:mandatory**, respectively.

{**~:↩**|**~@↩**|**~↩**}
▷ **Ignored Newline.** Ignore newline, or whitespace following newline, or both, respectively.

~ [$n_{\boxed{1}}$] **|** ▷ **Page.** Print n page separators.

~ [$n_{\boxed{1}}$] **~** ▷ **Tilde.** Print n tildes.

~ [*min-col*$_{\boxed{0}}$] [**,**[*col-inc*$_{\boxed{1}}$] [**,**[*min-pad*$_{\boxed{0}}$] [**,***pad-char*$_{\boxed{\ }}$]]]
[**:**] [**@**] **<** [*nl-text* **~**[*spare*$_{\boxed{0}}$ [**,***width*]]**:**;] {*text* **~;**}* *text* **~>**
▷ **Justification.** Justify text produced by *text*s in a field of at least *min-col* columns. With **:**, right justify; with **@**, left justify. If this would leave less than *spare* characters on the current line, output *nl-text* first.

~ [**:**] [**@**] **<** {[*prefix*$_{\boxed{\text{""}}}$ **~;**]|[*per-line-prefix* **~@;**]} *body* [**~;** *suffix*$_{\boxed{\text{""}}}$] **~:** [**@**] **>**
▷ **Logical Block.** Act like **pprint-logical-block** using *body* as (Fu **format**) control string on the elements of the list argument or, with **@**, on the remaining arguments, which are extracted by **pprint-pop**. With **:**, *prefix* and *suffix* default to (and). When closed by **~:@>**, spaces in *body* are replaced with conditional newlines.

{**~** [$n_{\boxed{0}}$] **i**|**~** [$n_{\boxed{0}}$] **:i**}
▷ **Indent.** Set indentation to n relative to leftmost/to current position.

~ [$c_{\boxed{1}}$] [,$i_{\boxed{1}}$] [**:**] [**@**] **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.

{**~** [$m_{\boxed{1}}$] *****|**~** [$m_{\boxed{1}}$] **:***|**~** [$n_{\boxed{0}}$] **@***}
▷ **Go-To.** Jump m arguments forward, or backward, or to argument n.

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

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

~ [*i*] [**:**] [**@**] **[** [{*text* **~;**}* *text*] [**~:;** *default*] **~]**
▷ **Conditional Expression.** Use the zero-indexed argumenth (or ith if given) *text* as a (Fu **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.

(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 2 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 var ***print-pprint-dispatch***.

var ***print-pprint-dispatch*** ▷ Current pretty print dispatch table.

13.5 Format

(M **formatter** *control*)
▷ Return function of stream and a **&rest** argument applying Fu **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 *arg*s. Alternatively, *control* can be a function returned by M **formatter** which is then applied to *out-stream* and *arg**. Output to *out-string*, *out-stream* or, if first argument is T, to var ***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-char*s on the left rather than on the right.

~ [*radix*10] [,[*width*] [,[*pad-char*␣] [,[*comma-char*,] [,*comma-interval*3]]]] [:] [@] **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*3]]] [:] [@] {**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*1] [,[*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.

({M **defun** | M **lambda**} {*foo* (*ord-λ**) | (**setf** *foo*) (*new-value ord-λ**)} (**declare** *decl**)* [*doc*] *form*P*)
▷ Define a function named *foo* or (**setf** *foo*), or an anonymous function, respectively, which applies *form*s to *ord-λ*s. For M **defun**, *form*s are enclosed in an implicit sO **block** named *foo*.

({sO **flet** | sO **labels**} (({*foo* (*ord-λ**) | (**setf** *foo*) (*new-value ord-λ**)} (**declare** *local-decl**)* [*doc*] *local-form*P*)*) (**declare** *decl**)* *form*P*)
▷ Evaluate *form*s with locally defined functions *foo*. Globally defined functions of the same name are shadowed. Each *foo* is also the name of an implicit sO **block** around its corresponding *local-form**. Only for **labels**, functions *foo* are visible inside *local-forms*. Return values of *form*s.

(sO **function** {*foo* | (M **lambda** *form**)})
▷ Return lexically innermost function named *foo* or a lexical closure of the M **lambda** expression.

(Fu **apply** {*function* | (**setf** *function*)} *arg* args*)
▷ Values of *function* called with *arg*s and the list elements of *args*. **setf**able if *function* is one of Fu **aref**, Fu **bit**, and Fu **sbit**.

(Fu **funcall** *function* arg*) ▷ Values of *function* called with *arg*s.

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

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

(Fu **values** *foo**)
▷ Return as multiple values the primary values of the *foo*s. **setf**able.

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

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

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

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

(Fu **identity** *foo*) ▷ Return *foo*.

(Fu **function-lambda-expression** *function*)
▷ If available, return lambda expression of *function*, NIL if *function* was defined in an environment without bindings, 2 and name 3 of *function*.

(Fu **fdefinition** {*foo* | (**setf** *foo*)})
▷ Definition of global function *foo*. **setf**able.

(Fu **fmakunbound** *foo*)
▷ Remove global function or macro definition *foo*.

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

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

9.4 Macros

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

([**&whole** *var*] [*E*] { *var* | (*macro-λ**) }* [*E*]

[**&optional** { *var* | ({ *var* | (*macro-λ**) } [*init*NIL [*supplied-p*]]) }*] [*E*]

[{ **&rest** | **&body** } { *rest-var* | (*macro-λ**) }] [*E*]

[**&key** { *var* | ({ *var* | (:*key* { *var* | (*macro-λ**) }) } [*init*NIL [*supplied-p*]]) }* [*E*]

[**&allow-other-keys**] [**&aux** { *var* | (*var* [*init*NIL]) }*] [*E*])

or

([**&whole** *var*] [*E*] { *var* | (*macro-λ**) }* [*E*] [**&optional**

{ *var* | ({ *var* | (*macro-λ**) } [*init*NIL [*supplied-p*]]) }*] [*E*] **.** *rest-var*).

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

(M **defmacro** | Fu **define-compiler-macro** { *foo* | (**setf** *foo*) } (*macro-λ**) (**declare** *decl**)* [*doc*] *form*P*)

▷ Define macro *foo* which on evaluation as (*foo tree*) applies expanded *form*s to arguments from *tree*, which corresponds to *tree*-shaped *macro-λ*s. *form*s are enclosed in an implicit sO **block** named *foo*.

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

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

(sO **macrolet** ((*foo* (*macro-λ**) (**declare** *local-decl**)* [*doc*] *macro-form*P*)*) (**declare** *decl**)* *form*P*)

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

(sO **symbol-macrolet** ((*foo expansion-form*)*) (**declare** *decl**)* *form*P*)

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

(M **defsetf** *function*

{ *updater* [*doc*] | (*setf-λ**) (*s-var**) (**declare** *decl**)* [*doc*] *form*P* }

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

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

[**&key** { *var* | ({ *var* | (:*key var*) } [*init*NIL [*supplied-p*]]) }*

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

▷ Specify how to **setf** a place accessed by *function*. **Short form:** (**setf** (*function arg**) *value-form*) is replaced by (*updater arg* value-form*); the latter must return *value-form*. **Long form:** on invocation of (**setf** (*function arg**) *value-form*), *form*s 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**. *form*s are enclosed in an implicit sO **block** named *function*.

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

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

(M **pprint-logical-block** (*stream list* { { **:prefix** *string* | **:per-line-prefix** *string* | **:suffix** *string*"" } })

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

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

(M **pprint-pop**)

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

(Fu **pprint-tab** { **:line** | **:line-relative** | **:section** | **:section-relative** } *c i* [*stream*var ***standard-output***])

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

(Fu **pprint-indent** { **:block** | **:current** } *n* [*stream*var ***standard-output***])

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

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

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

(Fu **pprint-newline** { **:linear** | **:fill** | **:miser** | **:mandatory** } [*stream*var ***standard-output***])

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

var ***print-array*** ▷ If T, print arrays Fu **read**ably.

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

var ***print-case***:upcase

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

var ***print-circle***NIL

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

var ***print-escape***T

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

var ***print-gensym***T

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

var ***print-length***NIL
var ***print-level***NIL
var ***print-lines***NIL

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

var ***print-miser-width***

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

var ***print-pretty*** ▷ If T, print pretty.

var ***print-radix***NIL ▷ If T, print rationals with a radix indicator.

var ***print-readably***NIL

▷ If T, print Fu **read**ably or signal error **print-not-readable**.

var ***print-right-margin***NIL

▷ Right margin width in ems while pretty-printing.

13.4 Printer

(Fu {**prin1** | **print** | **pprint** | **princ**} *foo* [*stream*var ***standard-output***])
▷ Print *foo* to *stream* **read**ably, **read**ably between a newline and a space, **read**ably after a newline, or human-readably without any extra characters, respectively. **prin1**, **print** and **princ** return *foo*.

(Fu **prin1-to-string** *foo*)
(Fu **princ-to-string** *foo*)
▷ Print *foo* to *string* **read**ably 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 *form*s 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** | **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** | **write-to-string**} *foo* {**:array** *bool* | **:base** *radix* | **:case** {**:upcase** | **: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 **write** only.)

(Fu **pprint-fill** *stream foo* [*parenthesis*T [*noop*]])
(Fu **pprint-tabular** *stream foo* [*parenthesis*T [*noop* [*n*16]]])
(Fu **pprint-linear** *stream foo* [*parenthesis*T [*noop*]])
▷ Print *foo* to *stream*. If *foo* is a list, print as many elements per line as possible; do the same in a table with a column width of *n* ems; or print either all elements on one line or each on its own line, respectively. Return NIL. Usable with **format** directive **~//**.

(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 *arg*s 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 *var*s to corresponding arguments if any.

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

&key *var**
▷ Bind *var*s 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 *var*s as in **let***.

9.5 Control Flow

(sO **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 *test*s return NIL.

(M {**when** | **unless**} *test foo*P*)
▷ Evaluate *foo*s and return their values if *test* returns T or NIL, respectively. Return NIL otherwise.

(M **case** *test* ({(*key**) | *key*} *foo*P*)* [({**otherwise** | T} *bar*P*)NIL])
▷ Return the values of the first *foo** one of whose *key*s is **eql** *test*. Return values of *bar*s if there is no matching *key*.

(M {**ecase** | **ccase**} *test* ({(*key**) | *key*} *foo*P*)*)
▷ Return the values of the first *foo** one of whose *key*s is **eql** *test*. Signal non-correctable/correctable **type-error** and return NIL if there is no matching *key*.

(M **and** *form**T)
▷ Evaluate *form*s 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 *form*s 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 *form*s 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** *decl**)* *form*P*)
▷ Evaluate *form*s with *name*s lexically bound (in parallel or sequentially, respectively) to *value*s. Return values of *form*s.

(M **prog** | M **prog***) (*name* | (*name* [*value*NIL]))* (**declare** *decl**)* (*tag* | *form*)*)
▷ Evaluate **tagbody**-like body with *name*s lexically bound (in parallel or sequentially, respectively) to *value*s. Return NIL or explicitly M **return**ed values. Implicitly, the whole form is a sO **block** named NIL.

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

(sO **unwind-protect** *protected cleanup**)
▷ Evaluate *protected* and then, no matter how control leaves *protected*, *cleanup*s. Return values of *protected*.

(M **destructuring-bind** *destruct-λ bar* (**declare** *decl**)* *form*P*)
▷ Evaluate *form*s 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** (*var**) *values-form* (**declare** *decl**)* *body-form*P*)
▷ Evaluate *body-form*s with *var*s lexically bound to the return values of *values-form*. Return values of *body-form*s.

(sO **block** *name form*P*)
▷ Evaluate *form*s 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 sO **block** named *foo*/named NIL, respectively, return with values of *result*.

(sO **tagbody** {*tag* | *form*}*)
▷ Evaluate *form*s in a lexical environment. *tag*s (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 **eql** *tag*.

(sO **catch** *tag form*P*)
▷ Evaluate *form*s 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** *decl**)* (*tag* | *form*)*)
▷ Evaluate sO **tagbody**-like body with *var*s successively bound according to the values of the corresponding *start* and *step* forms. *var*s are bound in parallel/sequentially, respectively. Stop iteration when *stop* is T. Return values of *result**. Implicitly, the whole form is a sO **block** named NIL.

13.3 Character Syntax

#| *multi-line-comment** **|#**
; *one-line-comment**
▷ Comments. There are stylistic conventions:

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

(*foo**[**.** *bar*NIL]**)** ▷ List of *foo*s with the terminating cdr *bar*.

" ▷ Begin and end of a string.

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

`([*foo*] [**,***bar*] [**,@***baz*] [**,.***quux*] [*bing*])
▷ Backquote. sO **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* ▷ (Fu **character "***c***"**), the character *c*.

#B*n*; **#O***n*; *n***.**; **#X***n*; **#***r***R***n*
▷ Integer of radix 2, 8, 10, 16, or *r*; $2 \le r \le 36$.

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

{[*m*]*n*[{**S**|**F**|**D**|**L**|**E**}*x*E0] | *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***)** ▷ (Fu **complex** *a b*), the complex number $a + b\mathrm{i}$.

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

#*n***A***sequence* ▷ *n*-dimensional array.

#[*n*]**(***foo**)
▷ Vector of some (or *n*) *foo*s filled with last *foo* if necessary.

#[*n*]*****b*
▷ Bit vector of some (or *n*) *b*s filled with last *b* if necessary.

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

#P*string* ▷ A pathname.

#:*foo* ▷ Uninterned symbol *foo*.

#.*form* ▷ Read-time value of *form*.

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.

(Fu **read-delimited-list** *char* [*stream*$_{\text{var *standard-input*}}$ [*recursive*$_{\texttt{NIL}}$]])
▷ Continue reading until encountering *char*. Return list of objects read. Signal error if no *char* is found in stream.

(Fu **read-char** [*stream*$_{\text{var *standard-input*}}$ [*eof-err*$_{\texttt{T}}$ [*eof-val*$_{\texttt{NIL}}$ [*recursive*$_{\texttt{NIL}}$]]]])
▷ Return next character from *stream*.

(Fu **read-char-no-hang** [*stream*$_{\text{var *standard-input*}}$ [*eof-error*$_{\texttt{T}}$ [*eof-val*$_{\texttt{NIL}}$ [*recursive*$_{\texttt{NIL}}$]]]])
▷ Next character from *stream* or `NIL` if none is available.

(Fu **peek-char** [*mode*$_{\texttt{NIL}}$ [*stream*$_{\text{var *standard-input*}}$ [*eof-error*$_{\texttt{T}}$ [*eof-val*$_{\texttt{NIL}}$ [*recursive*$_{\texttt{NIL}}$]]]]])
▷ 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*$_{\text{var *standard-input*}}$])
▷ Put last **read-char**ed *character* back into *stream*; return `NIL`.

(Fu **read-byte** *stream* [*eof-err*$_{\texttt{T}}$ [*eof-val*$_{\texttt{NIL}}$]])
▷ Read next byte from binary *stream*.

(Fu **read-line** [*stream*$_{\text{var *standard-input*}}$ [*eof-err*$_{\texttt{T}}$ [*eof-val*$_{\texttt{NIL}}$ [*recursive*$_{\texttt{NIL}}$]]]])
▷ Return a line of text from *stream* and $\frac{\texttt{T}}{2}$ if line has been ended by end of file.

(Fu **read-sequence** *sequence* *stream* [**:start** *start*$_{0}$][**:end** *end*$_{\texttt{NIL}}$])
▷ 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*)$_{\text{:upcase}}$
▷ Case sensitivity attribute (one of **:upcase**, **:downcase**, **:preserve**, **:invert**) of *readtable*. **setf**able.

(Fu **copy-readtable** [*from-readtable*$_{\text{var *readtable*}}$ [*to-readtable*$_{\texttt{NIL}}$]])
▷ Return copy of *from-readtable*.

(Fu **set-syntax-from-char** *to-char from-char* [*to-readtable*$_{\text{var *readtable*}}$ [*from-readtable*$_{\text{standard readtable}}$]])
▷ Copy syntax of *from-char* to *to-readtable*. Return `T`.

var ***readtable*** ▷ Current readtable.

var ***read-base***$*_{10}$ ▷ Radix for reading **integer**s and **ratio**s.

var ***read-default-float-format***$*_{\text{single-float}}$
▷ Floating point format to use when not indicated in the number read.

var ***read-suppress***$*_{\texttt{NIL}}$
▷ If `T`, reader is syntactically more tolerant.

(Fu **set-macro-character** *char function* [*non-term-p*$_{\texttt{NIL}}$ [*rt*$_{\text{var *readtable*}}$]])
▷ Make *char* a macro character associated with *function* of stream and *char*. Return `T`.

(Fu **get-macro-character** *char* [*rt*$_{\text{var *readtable*}}$])
▷ Reader macro function associated with *char*, and $\frac{\texttt{T}}{2}$ if *char* is a non-terminating macro character.

(Fu **make-dispatch-macro-character** *char* [*non-term-p*$_{\texttt{NIL}}$ [*rt*$_{\text{var *readtable*}}$]])
▷ Make *char* a dispatching macro character. Return `T`.

(Fu **set-dispatch-macro-character** *char sub-char function* [*rt*$_{\text{var *readtable*}}$])
▷ 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*$_{\text{var *readtable*}}$])
▷ Dispatch function associated with *char* followed by *sub-char*.

(M **dotimes** (*var i* [*result*$_{\texttt{NIL}}$]) (**declare** *decl**)* {*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 sO **block** named `NIL`.

(M **dolist** (*var list* [*result*$_{\texttt{NIL}}$]) (**declare** *decl**)* {*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 sO **block** named `NIL`.

9.7 Loop Facility

(M **loop** *form**)
▷ **Simple Loop.** If *form*s do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit sO **block** named `NIL`.

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

named $n_{\texttt{NIL}}$ ▷ Give M **loop**'s implicit sO **block** a name.

{**with** $\begin{Bmatrix}var\text{-}s\\(var\text{-}s^*)\end{Bmatrix}$ [*d-type*] [= *foo*]}$^+$
{**and** $\begin{Bmatrix}var\text{-}p\\(var\text{-}p^*)\end{Bmatrix}$ [*d-type*] [= *bar*]}*
where destructuring type specifier *d-type* has the form
$\left\{\textbf{fixnum}|\textbf{float}|\texttt{T}|\texttt{NIL}|\{\textbf{of-type}\ \begin{Bmatrix}type\\(type^*)\end{Bmatrix}\}\right\}$
▷ Initialize (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel.

{{**for**|**as**} $\begin{Bmatrix}var\text{-}s\\(var\text{-}s^*)\end{Bmatrix}$ [*d-type*]}$^+$ {**and** $\begin{Bmatrix}var\text{-}p\\(var\text{-}p^*)\end{Bmatrix}$ [*d-type*]}*
▷ Begin of iteration control clauses. Initialize and step (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel. Destructuring type specifier *d-type* as with **with**.

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

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

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

by {*step*$_{1}$|*function*$_{\text{\#'cdr}}$}
▷ Specify the (positive) decrement or increment or the *function* of one argument returning the next part of the list.

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

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

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

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

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

{**symbol**|**symbols**|**present-symbol**|**present-symbols**|**external-symbol**|**external-symbols**} [{**of**|**in**} *package*$_{\text{var *package*}}$]
▷ Bind *var* successively to the accessible symbols, or the present symbols, or the external symbols respectively, of *package*.

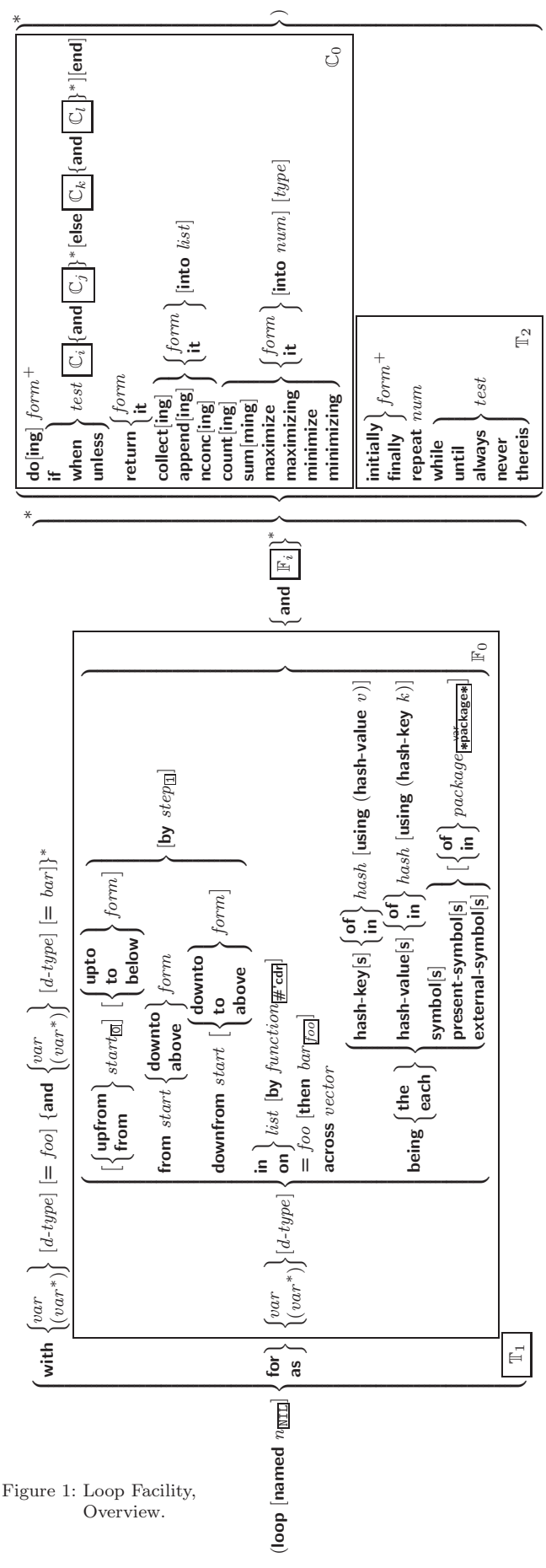

Figure 1: Loop Facility, Overview.

(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** *decl**)* [*doc*] *form*P*)
▷ Define type *foo* which when referenced as (*foo arg**) applies expanded *form*s to *arg*s returning the new type. For (*macro-λ**) see p. 18 but with default value of * instead of NIL. *form*s are enclosed in an implicit **block** named *foo*.

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

(**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 *type*s.

(**or** *type**NIL) ▷ Type specifier for union of *type*s.

(**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 **streamp** *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**|**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 *arg*s.

(M **with-standard-io-syntax** *form*P*)
▷ Evaluate *form*s with standard behaviour of reader and printer. Return values of *form*s.

(Fu **read**|**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*0|**:end** *end*NIL|**:preserve-whitespace** *bool*NIL}]]])
▷ Return object read from string and zero-indexed position of next character.

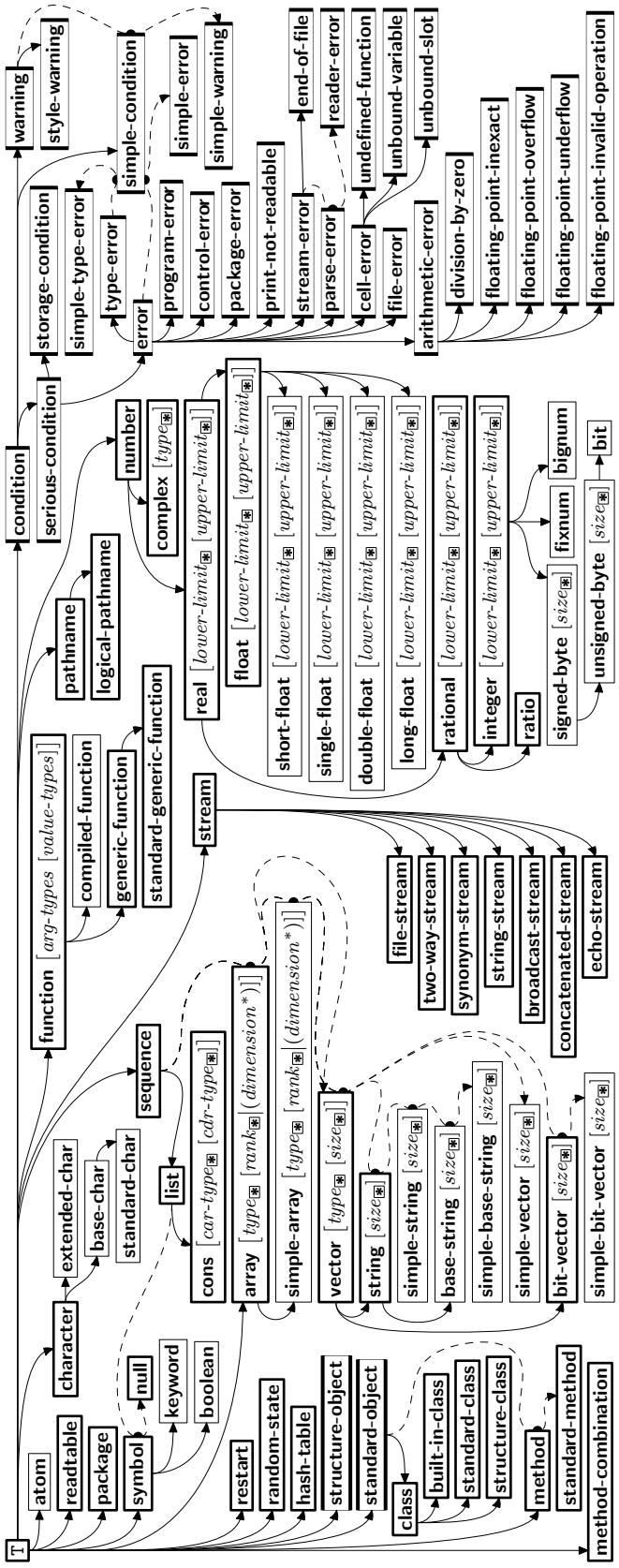

Figure 2: Precedence Order of System Classes (▭), Classes (▬), Types (▭), and Condition Types (▭).

{**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 Fu **append** or Fu **nconc**, respectively. If no *list* is given, collect into an anonymous list which is returned after termination.

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

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

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

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

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

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

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

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

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

10 CLOS

10.1 Classes

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

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

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

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

▷ Define, as a subclass of *superclass*es, 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* T [*environment*]])
▷ Return class named *symbol*. **setf**able.

(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 *initarg*s.

(Fu **slot-value** *foo slot*) ▷ Return value of *slot* in *foo*. **setf**able.

(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*P*)
▷ Return values of *forms* after evaluating them in a lexical environment with slots of *instance* visible as **setf**able *slot*s or *var*s/with *accessor*s of *instance* visible as **setf**able *var*s.

(gF **class-name** *class*)
((**setf** gF **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 gF **make-instance**/of gF **change-class** by means of gF **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 gF **make-instances-obsolete** by means of gF **shared-initialize**.

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

(gF **shared-initialize** *instance* { *slots* | T } {**:***initarg value*}* *other-keyarg**)
▷ Fill *instance*'s *slots* using *initarg*s and **:initform** forms.

(gF **slot-missing** *class object slot* { **setf** | **slot-boundp** | **slot-makunbound** | **slot-value** } [*value*])
▷ Called in case of attempted access to missing *slot*. Its primary method signals **error**.

(M **with-condition-restarts** *condition restarts form*P*)
▷ Evaluate *form*s with *restarts* dynamically associated with *condition*. Return values of *form*s.

(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 Fu **format** control or list of Fu **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.

(sQ **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*P*)* [({ **otherwise** | T } *b-form*P* NIL)])
▷ Return values of the *a-form*s whose *type* is *foo* of. Return values of *b-form*s if no *type* matches.

({ M **ctypecase** | M **etypecase** } *foo* (*type form*P*)*)
▷ Return values of the *form*s 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.

(M **assert** *test* [(*place**) [{*condition continue-arg** | *type* {**:***initarg-name value*}* | *control arg**}]])
▷ If *test*, which may depend on *place*s, returns NIL, signal as correctable **error** *condition* or a new condition of *type* or, with (Fu **format** *control* and *arg*s (see p. 36), **error**. When using the debugger's continue option, *place*s can be altered before re-evaluation of *test*. Return NIL.

(M **handler-case** *foo* (*type* ([*var*]) (**declare** $\widehat{decl^*}$)* *condition-form*P*)* [(**:no-error** (*ord-λ**) (**declare** $\widehat{decl^*}$)* *form*P*)])
▷ If, on evaluation of *foo*, a condition of *type* is signalled, evaluate matching *condition-form*s with *var* bound to the condition, and return their values. Without a condition, bind *ord-λ*s to values of *foo* and return values of *form*s 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 *form*s after evaluating them with *condition-type*s dynamically bound to their respective *handler-function*s of argument condition.

(M **with-simple-restart** ({*restart* | NIL} *control arg**) *form*P*)
▷ Return values of *form*s unless *restart* is called during their evaluation. In this case, describe restart using (Fu **format** *control* and *arg*s (see p. 36) and return NIL and T.

(M **restart-case** *form* (*foo* (*ord-λ**) {**:interactive** *arg-function* | **:report** {*report-function* | *string* "*foo*"} | **:test** *test-function* T}
(**declare** $\widehat{decl^*}$)* *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-form*s. *arg-function* supplies appropriate *arg*s 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** (({$\widehat{restart}$ | NIL} *restart-function*
{**:interactive-function** *function* | **:report-function** *function* | **:test-function** *function*})*) *form*P*)
▷ Return values of *form*s evaluated with *restart*s dynamically bound to *restart-function*s.

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

(gF **slot-unbound** *class instance slot*)
▷ Called by Fu **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** {*foo* | (**setf** *foo*)} (*required-var** [**&optional** {*var* | (*var*)}*] [**&rest** *var*] [**&key** {*var* | (*var* | (**:***key var*))}*] [**&allow-other-keys**])
{(**:argument-precedence-order** *required-var*+) | (**declare** (**optimize** *arg**)+) | (**:documentation** $\widehat{string}$) | (**:generic-function-class** *class* standard-generic-function) | (**:method-class** *class* standard-method) | (**:method-combination** *c-type* standard *c-arg**) | (**:method** *defmethod-args*)*})
▷ Define generic function *foo*. *defmethod-args* resemble those of M **defmethod**. For *c-type* see section 10.3.

(Fu **ensure-generic-function** {*foo* | (**setf** *foo*)}
{**:argument-precedence-order** *required-var*+ | **:declare** (**optimize** *arg**)+ | **:documentation** *string* | **:generic-function-class** *class* | **:method-class** *class* | **:method-combination** *c-type c-arg** | **:lambda-list** *lambda-list* | **:environment** *environment*})
▷ 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** {*foo* | (**setf** *foo*)} [{**:before** | **:after** | **:around** | *qualifier**} primary method]
({*var* | (*spec-var* {*class* | (**eql** *bar*)})}* [**&optional** {*var* | (*var* [*init* [*supplied-p*]])}*] [**&rest** *var*] [**&key** {*var* | ({*var* | (**:***key var*)} [*init* [*supplied-p*]])}* [**&allow-other-keys**]] [**&aux** {*var* | (*var* [*init*])}*]) {(**declare** $\widehat{decl^*}$)* | $\widehat{doc}$} *form*P*)
▷ Define new method for generic function *foo*. *spec-var*s specialize to either being of *class* or being **eql** *bar*, respectively. On invocation, *var*s and *spec-var*s of the new method act like parameters of a function with body *form**. *form*s are enclosed in an implicit S(O) **block** *foo*. Applicable *qualifier*s depend on the **method-combination** type; see section 10.3.

({gF **add-method** | gF **remove-method**} *generic-function method*)
▷ Add (if necessary) or remove (if any) *method* to/from *generic-function*.

(gF **find-method** *generic-function qualifiers specializers* [*error* T])
▷ 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 *arg*s; return its values.

(gF **no-applicable-method** *generic-function arg**)
▷ Called on invocation of *generic-function* on *arg*s if there is no applicable method. Default method signals **error**.

(Fu **invalid-method-error** *method* | Fu **method-combination-error** *method* *control arg**)
▷ Signal **error** on applicable method with invalid qualifiers, or on method combination. For *control* and *arg*s 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 T 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 primary method which supplies the values of the calling Fu **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-type*s defined by the short form of M **define-method-combination**.

(M **define-method-combination** *c-type*
{**:documentation** *string*
| **:identity-with-one-argument** *bool* [NIL]
| **: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-method*s are ordered [{**:most-specific-first** | **:most-specific-last**} [:most-specific-first]] (specified as *c-arg* in M **defgeneric**). Using *c-type* as the *qualifier* in M **defmethod** makes the method primary.

(M **define-method-combination** *c-type* (*ord-λ**) ((*group*
{***** | (*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. M **defgeneric**), with *symbol* bound to the generic function, with *method-combination-λ** bound to the arguments of the generic function, and with *group*s bound to lists of methods. An applicable method becomes a member of the leftmost *group* whose *predicate* or *qualifier*s match. Methods can be called via M **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.

(M **call-method** {*method* | (M **make-method** *form*)} [({*next-method* | (M **make-method** *form*)}*)])
▷ From within an effective method form, call *method* with the arguments of the generic function and with information about its *next-method*s; return its values.

11 Conditions and Errors

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

(M **define-condition** *foo* (*parent-type** [**condition**])
{*slot*
| (*slot* {**:reader** *reader*}*
| {**:writer** {*writer* | (**setf** *writer*)}}*
| {**:accessor** *accessor*}*
| **:allocation** {**:instance** | **:class**} [:instance]
| {**:initarg** *:initarg-name*}*
| **:initform** *form*
| **:type** *type*
| **:documentation** *slot-doc*)}*
{(**:default-initargs** {*name value*}*)
| (**:documentation** *condition-doc*)
| (**:report** {*string* | *report-function*})}*)
▷ Define, as a subtype of *parent-type*s, 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 writeable 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.

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

({Fu **signal** | Fu **warn** | Fu **error**} {*condition* | *type* {*:initarg-name value*}* | *control arg**})
▷ Unless handled, signal as **condition**, **warning** or **error**, respectively, *condition* or a new condition of *type* or, with Fu **format** *control* and *arg*s (see p. 36), **simple-condition**, **simple-warning**, or **simple-error**, respectively. From Eu **signal** and Fu **warn**, return NIL.

(Fu **cerror** *continue-control* {*condition continue-arg** | *type* {*:initarg-name value*}* | *control arg**})
▷ Unless handled, signal as correctable **error** *condition* or a new condition of *type* or, with Fu **format** *control* and *arg*s (see p. 36), **simple-error**. In the debugger, use Fu **format** arguments *continue-control* and *continue-arg*s to tag the continue option. Return NIL.

(M **ignore-errors** *form*P*)
▷ Return values of *form*s or, in case of **error**s, NIL and the condition.
2

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