Quick Reference

Common lisp

Bert Burgemeister

Common Lisp Quick Reference Revision 145 [2018-01-15]

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

Contents

Typographic Conventions

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

them ▷ Placeholder for actual code.

`me` ▷ Literal text.

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

foo*; {*foo*}* ▷ Zero or more *foo*s.

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

foos ▷ English plural denotes a list argument.

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

$\left\{\left|\begin{matrix} foo \\ bar \\ baz \end{matrix}\right.\right.$ ▷ 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^{P*} ▷ *foo** is evaluated as in *s***progn**; see page 21.

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

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

1 Numbers

1.1 Predicates

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

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

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

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

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

1.2 Numeric Functions

($_f$**+** *a*$_0^*$)
($_f$***** *a*$_1^*$)
▷ Return $\sum a$ or $\prod a$, respectively.

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

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

($\left\{{}_m\textbf{incf} \atop {}_m\textbf{decf}\right\}$ $\widetilde{place}$ [*delta*$_1$])
▷ Increment or decrement the value of *place* by *delta*. Return new value.

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

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

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

($_f$**lcm** *integer*$^*_1$)
($_f$**gcd** *integer**)
▷ Least common multiple or greatest common denominator, respectively, of *integer*s. (**gcd**) returns 0.

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

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

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

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

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

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

(*f*cis *a*) ▷ Return $e^{i\,a} = \cos a + i \sin a$.

(*f*conjugate *a*) ▷ Return complex conjugate of *a*.

(*f*max *num*+)
(*f*min *num*+) ▷ Greatest or least, respectively, of *num*s.

({ {*f*round | *f*round}
  { *f*floor | *f*ffloor}
  { *f*ceiling | *f*fceiling}
  { *f*truncate | *f*ftruncate} } *n* [*d*1])
▷ Return as **integer** or **float**, respectively, *n*/*d* rounded, or rounded towards $-\infty$, $+\infty$, or 0, respectively; and remainder$_2$.

({ *f*mod
  *f*rem } *n* *d*)
▷ Same as *f*floor or *f*truncate, respectively, but return remainder only.

(*f*random *limit* [$\widetilde{state}$ *v**random-state*])
▷ Return non-negative random number less than *limit*, and of the same type.

(*f*make-random-state [{*state* | NIL | T}NIL])
▷ Copy of **random-state** object *state* or of the current random state; or a randomly initialized fresh random state.

*v**random-state* ▷ Current random state.

(*f*float-sign *num-a* [*num-b*1]) ▷ *num-b* with *num-a*'s sign.

(*f*signum *n*)
▷ Number of magnitude 1 representing sign or phase of *n*.

(*f*numerator *rational*)
(*f*denominator *rational*)
▷ Numerator or denominator, respectively, of *rational*'s canonical form.

(*f*realpart *number*)
(*f*imagpart *number*)
▷ Real part or imaginary part, respectively, of *number*.

(*f*complex *real* [*imag*0]) ▷ Make a complex number.

(*f*phase *num*) ▷ Angle of *num*'s polar representation.

(*f*abs *n*) ▷ Return $|n|$.

(*f*rational *real*)
(*f*rationalize *real*)
▷ Convert *real* to rational. Assume complete/limited accuracy for *real*.

(*f*float *real* [*prototype*0.0F0])
▷ Convert *real* into float with type of *prototype*.

1.3 Logic Functions

Negative integers are used in two's complement representation.

(*f* **boole** *operation int-a int-b*)
▷ Return value of bitwise logical *operation*. *operation*s are

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

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

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

(*f* **logandc1** *int-a int-b*) ▷ $\neg$*int-a* $\wedge$ *int-b*.

(*f* **logandc2** *int-a int-b*) ▷ *int-a* $\wedge$ $\neg$*int-b*.

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

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

(*f* **logorc1** *int-a int-b*) ▷ $\neg$*int-a* $\vee$ *int-b*.

(*f* **logorc2** *int-a int-b*) ▷ *int-a* $\vee$ $\neg$*int-b*.

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

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

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

(*f* **logcount** *int*)
▷ Number of 1 bits in *int* ≥ 0, number of 0 bits in *int* < 0.

1.4 Integer Functions

(*f*__integer-length__ *integer*)
▷ Number of bits necessary to represent *integer*.

(*f*__ldb-test__ *byte-spec integer*)
▷ Return T if any bit specified by *byte-spec* in *integer* is set.

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

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

(*f*__deposit-field__ / *f*__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 (*f*__byte-size__ *byte-spec*) bits of *int-a*, respectively.

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

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

(*f*__byte-size__ *byte-spec*)
(*f*__byte-position__ *byte-spec*)
▷ Size or position, respectively, of *byte-spec*.

1.5 Implementation-Dependent

*c*__short-float__, *c*__single-float__, *c*__double-float__, *c*__long-float__ - __epsilon__ / __negative-epsilon__
▷ Smallest possible number making a difference when added or subtracted, respectively.

*c*__least-negative__, *c*__least-negative-normalized__, *c*__least-positive__, *c*__least-positive-normalized__ - __short-float__ / __single-float__ / __double-float__ / __long-float__
▷ Available numbers closest to −0 or +0, respectively.

*c*__most-negative__, *c*__most-positive__ - __short-float__ / __single-float__ / __double-float__ / __long-float__ / __fixnum__
▷ Available numbers closest to $-\infty$ or $+\infty$, respectively.

(*f*__decode-float__ *n*)
(*f*__integer-decode-float__ *n*)
▷ Return significand, exponent, and sign of **float** *n*.

(*f*__scale-float__ *n* [*i*]) ▷ With *n*'s radix *b*, return nb^i.

(*f*__float-radix__ *n*)
(*f*__float-digits__ *n*)
(*f*__float-precision__ *n*)
▷ Radix, number of digits in that radix, or precision in that radix, respectively, of float *n*.

(*f*__upgraded-complex-part-type__ *foo* [*environment*$_{\text{NIL}}$])
▷ Type of most specialized **complex** number able to hold parts of type *foo*.

Index

2 Characters

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

(*f*__characterp__ *foo*)
(*f*__standard-char-p__ *char*) ▷ T if argument is of indicated type.

(*f*__graphic-char-p__ *character*)
(*f*__alpha-char-p__ *character*)
(*f*__alphanumericp__ *character*)
▷ T if *character* is visible, alphabetic, or alphanumeric, respectively.

(*f*__upper-case-p__ *character*)
(*f*__lower-case-p__ *character*)
(*f*__both-case-p__ *character*)
▷ Return T if *character* is uppercase, lowercase, or able to be in another case, respectively.

(*f*__digit-char-p__ *character* [*radix*$_{10}$])
▷ Return its weight if *character* is a digit, or NIL otherwise.

(*f*__char=__ *character*$^+$)
(*f*__char/=__ *character*$^+$)
▷ Return T if all *character*s, or none, respectively, are equal.

(*f*__char-equal__ *character*$^+$)
(*f*__char-not-equal__ *character*$^+$)
▷ Return T if all *character*s, or none, respectively, are equal ignoring case.

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

(*f*__char-greaterp__ *character*$^+$)
(*f*__char-not-lessp__ *character*$^+$)
(*f*__char-lessp__ *character*$^+$)
(*f*__char-not-greaterp__ *character*$^+$)
▷ Return T if *character*s are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.

(*f*__char-upcase__ *character*)
(*f*__char-downcase__ *character*)
▷ Return corresponding uppercase/lowercase character, respectively.

(*f*__digit-char__ *i* [*radix*$_{10}$]) ▷ Character representing digit *i*.

(*f*__char-name__ *char*) ▷ *char*'s name if any, or NIL.

(*f*__name-char__ *foo*) ▷ Character named *foo* if any, or NIL.

(*f*__char-int__ *character*)
(*f*__char-code__ *character*) ▷ Code of *character*.

(*f*__code-char__ *code*) ▷ Character with *code*.

*c*__char-code-limit__ ▷ Upper bound of (*f*__char-code__ *char*); ≥ 96.

(*f*__character__ *c*) ▷ Return #*c*.

3 Strings

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

(*f* **stringp** *foo*)
(*f* **simple-string-p** *foo*) ▷ T if *foo* is of indicated type.

(*f* {**string=** | **string-equal**} *foo bar* {**:start1** *start-foo*[0] | **:start2** *start-bar*[0] | **:end1** *end-foo*[NIL] | **:end2** *end-bar*[NIL]})
▷ Return T if subsequences of *foo* and *bar* are equal. Obey/ignore, respectively, case.

(*f* {**string{/=|-not-equal}** | **string{>|-greaterp}** | **string{>=|-not-lessp}** | **string{<|-lessp}** | **string{<=|-not-greaterp}**} *foo bar* {**:start1** *start-foo*[0] | **:start2** *start-bar*[0] | **:end1** *end-foo*[NIL] | **:end2** *end-bar*[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.

(*f* **make-string** *size* {**:initial-element** *char* | **:element-type** *type*[character]})
▷ Return string of length *size*.

(*f* **string** *x*)
(*f* {**string-capitalize** | **string-upcase** | **string-downcase**} *x* {**:start** *start*[0] | **:end** *end*[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.

(*f* {**nstring-capitalize** | **nstring-upcase** | **nstring-downcase**} *string* {**:start** *start*[0] | **:end** *end*[NIL]})
▷ Convert *string* into a string with capitalized words, an all-uppercase string, or an all-lowercase string, respectively.

(*f* {**string-trim** | **string-left-trim** | **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.

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

(*f* **parse-integer** *string* {**:start** *start*[0] | **:end** *end*[NIL] | **:radix** *int*[10] | **:junk-allowed** *bool*[NIL]})
▷ Return integer parsed from *string* and index[2] of parse end.

4 Conses

4.1 Predicates

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

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

15.4 Declarations

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

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

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

(**dynamic-extent** *variable** (**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*.

(*f* {**ignorable** | **ignore**} {*var* | (**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*[3]) | **debug** | (**debug** *n*[3]) | **safety** | (**safety** *n*[3]) | **space** | (**space** *n*[3]) | **speed** | (**speed** *n*[3])})
▷ Tell compiler how to optimize. $n = 0$ means unimportant, $n = 1$ is neutral, $n = 3$ means important.

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

16 External Environment

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

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

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

(*f* **decode-universal-time** *universal-time* [*time-zone*[current]])
(*f* **get-decoded-time**)
▷ Return second[0], minute[1], hour[2], date[3], month[4], year[5], day[6], daylight-p[7], and zone[8].

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

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

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

15.3 REPL and Debugging

$_v$+ | $_v$++ | $_v$+++
$_v$* | $_v$** | $_v$***
$_v$/ | $_v$// | $_v$///
▷ Last, penultimate, or antepenultimate form evaluated in the REPL, or their respective primary value, or a list of their respective values.

$_v$− ▷ Form currently being evaluated by the REPL.

($_f$**apropos** *string* [*package*NIL])
▷ Print interned symbols containing *string*.

($_f$**apropos-list** *string* [*package*NIL])
▷ List of interned symbols containing *string*.

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

($_f$**ed** [*file-or-function*NIL]) ▷ Invoke editor if possible.

({$_f$**macroexpand-1** | $_f$**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.

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

($_m$**trace** {*function* | (**setf** *function*)}*)
▷ Cause *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.

$_v$***trace-output***
▷ Output stream $_m$**trace** and $_m$**time** send their output to.

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

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

($_m$**time** *form*)
▷ Evaluate *form*s and print timing information to $_v$***trace-output***. Return values of *form*.

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

($_f$**describe** *foo* [*stream*$_{v*standard-output*}$])
▷ Send information about *foo* to *stream*.

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

($_f$**disassemble** *function*)
▷ Send disassembled representation of *function* to $_v$***standard-output***. Return NIL.

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

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

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

($_f$**member** *foo list* {{**:test** *function*$_{\#'eql}$ | **:test-not** *function*} | **:key** *function*})
▷ Return tail of *list* starting with its first element matching *foo*. Return NIL if there is no such element.

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

($_f$**subsetp** *list-a list-b* {{**:test** *function*$_{\#'eql}$ | **:test-not** *function*} | **:key** *function*})
▷ Return T if *list-a* is a subset of *list-b*.

4.2 Lists

($_f$**cons** *foo bar*) ▷ Return new cons (*foo* . *bar*).

($_f$**list** *foo**) ▷ Return list of *foo*s.

($_f$**list*** *foo*$^+$)
▷ Return list of *foo*s with last *foo* becoming cdr of last cons. Return *foo* if only one *foo* given.

($_f$**make-list** *num* [**:initial-element** *foo*NIL])
▷ New list with *num* elements set to *foo*.

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

($_f$**car** *list*) ▷ Car of *list* or NIL if *list* is NIL. **setf**able.

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

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

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

($_f$**nth** *n list*) ▷ Zero-indexed *n*th element of *list*. **setf**able.

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

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

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

({$_f$**rplaca** | $_f$**rplacd**} *cons object*)
▷ Replace car, or cdr, respectively, of *cons* with *object*.

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

($_f$**adjoin** *foo list* {{**:test** *function*$_{\#'eql}$ | **:test-not** *function*} | **:key** *function*})
▷ Return *list* if *foo* is already member of *list*. If not, return ($_f$**cons** *foo list*).

($_m$**pop** *place*) ▷ Set *place* to ($_f$**cdr** *place*), return ($_f$**car** *place*).

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

(*m*pushnew *foo* *place* {:test *function*#'eql | :test-not *function* | :key *function*})
▷ Set *place* to (*f*adjoin *foo* *place*).

(*f*append [*proper-list** *foo*NIL])
(*f*nconc [*non-circular-list** *foo*NIL])
▷ Return concatenated list or, with only one argument, *foo*. *foo* can be of any type.

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

({*f*mapcar | *f*maplist} *function* *list*+)
▷ Return list of return values of *function* successively invoked with corresponding arguments, either cars or cdrs, respectively, from each *list*.

({*f*mapcan | *f*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.

({*f*mapc | *f*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.

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

4.3 Association Lists

(*f*pairlis *keys* *values* [*alist*NIL])
▷ Prepend to *alist* an association list made from lists *keys* and *values*.

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

({*f*assoc | *f*rassoc} *foo* *alist* {:test *test*#'eql | :test-not *test* | :key *function*})
({*f*assoc-if[-not] | *f*rassoc-if[-not]} *test* *alist* [:key *function*])
▷ First cons whose car, or cdr, respectively, satisfies *test*.

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

4.4 Trees

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

({*f*subst *new* *old* *tree* | *f*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*.

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

(*f*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 in case of warnings or errors excluding style-warnings.

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

(*f*load *path* {: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 *f*compile-file/by *f*load.

{v*compile | v*load} {print* | verbose*}
▷ Defaults used by *f*compile-file/by *f*load.

(*s*eval-when ({{:compile-toplevel|compile} | {:load-toplevel|load} | {:execute|eval}}) *form*P*)
▷ Return values of *form*s if *s*eval-when is in the top-level of a file being compiled, in the top-level of a compiled file being loaded, or anywhere, respectively. Return NIL if *form*s are not evaluated. (compile, load and eval deprecated.)

(*s*locally (declare *decl**)* *form*P*)
▷ Evaluate *form*s in a lexical environment with declarations *decl* in effect. Return values of *form*s.

(*m*with-compilation-unit ([:override *bool*NIL]) *form*P*)
▷ Return values of *form*s. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of *form*s.

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

(*s*quote *foo*) ▷ Return unevaluated *foo*.

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

(*f*make-load-form-saving-slots *foo* {:slot-names *slots*all local slots | :environment *environment*})
▷ Return a creation form and an initialization form which on evaluation construct an object equivalent to *foo* with *slots* initialized with the corresponding values from *foo*.

(*f*macro-function *symbol* [*environment*])
(*f*compiler-macro-function {*name* | (setf *name*)} [*environment*])
▷ Return specified macro function, or compiler macro function, respectively, if any. Return NIL otherwise. setfable.

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

(ƒ**gensym** [$s_{\texttt{"G"}}$])
▷ Return fresh, uninterned symbol #:sn with n from $_{v}$***gensym-counter****. Increment $_{v}$***gensym-counter****.

(ƒ**gentemp** [$prefix_{\texttt{T}}$ [$package_{v\text{*package*}}$]])
▷ Intern fresh symbol in package. Deprecated.

(ƒ**copy-symbol** *symbol* [$props_{\texttt{NIL}}$])
▷ Return uninterned copy of *symbol*. If *props* is `T`, give copy the same value, function and property list.

(ƒ**symbol-name** *symbol*)
(ƒ**symbol-package** *symbol*)
(ƒ**symbol-plist** *symbol*)
(ƒ**symbol-value** *symbol*)
(ƒ**symbol-function** *symbol*)
▷ Name, package, property list, value, or function, respectively, of *symbol*. **setf**able.

(ƒ{**documentation** | (**setf** $_{g}$**documentation**) *new-doc*} *foo* {**'variable**|**'function** | **'compiler-macro** | **'method-combination** | **'structure**|**'type**|**'setf**|`T`})
▷ Get/set documentation string of *foo* of given type.

$_{c}$**t**
▷ Truth; the supertype of every type including **t**; the superclass of every class except **t**; $_{v}$***terminal-io****.

$_{c}$**nil**|$_{c}$**()**
▷ Falsity; the empty list; the empty type, subtype of every type; $_{v}$***standard-input****; $_{v}$***standard-output****; the global environment.

14.4 Standard Packages

common-lisp|**cl**
▷ Exports the defined names of Common Lisp except for those in the **keyword** package.

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

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

15 Compiler

15.1 Predicates

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

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

15.2 Compilation

(ƒ**compile** {`NIL` *definition* | {*name* | (**setf** *name*)} [*definition*]})
▷ Return compiled function or replace *name*'s function definition with the compiled function. Return $\frac{\texttt{T}}{2}$ in case of **warning**s or **error**s, and $\frac{\texttt{T}}{3}$ in case of **warning**s or **error**s excluding **style-warning**s.

(ƒ{**sublis** *association-list tree* | **nsublis** *association-list tree*} {{**:test** *function*$_{\#\text{'eql}}$ | **: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.

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

4.5 Sets

(ƒ{**intersection** | **set-difference** | **union** | **set-exclusive-or**} *a b* | ƒ{**nintersection** | **nset-difference**} $\widetilde{a}$ *b* | ƒ{**nunion** | **nset-exclusive-or**} $\widetilde{a}$ $\widetilde{b}$ {{**:test** *function*$_{\#\text{'eql}}$ | **: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

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

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

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

5.2 Array Functions

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

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

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

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

(ƒ**array-dimensions** *array*)
▷ List containing the lengths of *array*'s dimensions.

(ƒ**array-dimension** *array i*) ▷ Length of ith dimension of *array*.

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

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

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

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

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

({*f* **bit-eqv** | *f* **bit-and** | *f* **bit-andc1** | *f* **bit-andc2** | *f* **bit-nand** | *f* **bit-ior** | *f* **bit-orc1** | *f* **bit-orc2** | *f* **bit-xor** | *f* **bit-nor**} *bit-array-a bit-array-b* [*result-bit-array*$_{\texttt{NIL}}$])
▷ Return result of bitwise logical operations (cf. operations of *f* **boole**, page 5) on *bit-array-a* and *bit-array-b*. If *result-bit-array* is `T`, put result in *bit-array-a*; if it is `NIL`, make a new array for result.

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

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

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

5.3 Vector Functions

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

(*f* **vector** *foo**) ▷ Return fresh simple vector of *foo*s.

(*f* **svref** *vector i*) ▷ Element *i* of simple *vector*. **setf**able.

(*f* **vector-push** *foo vector*)
▷ Return `NIL` if *vector*'s fill pointer equals size of *vector*. Otherwise replace element of *vector* pointed to by fill pointer with *foo*; then increment fill pointer.

(*f* **vector-push-extend** *foo vector* [*num*])
▷ Replace element of *vector* pointed to by fill pointer with *foo*, then increment fill pointer. Extend *vector*'s size by $\geq$ *num* if necessary.

(*f* **vector-pop** *vector*)
▷ Return element of *vector* its fillpointer points to after decrementation.

(*f* **fill-pointer** *vector*) ▷ Fill pointer of *vector*. **setf**able.

6 Sequences

6.1 Sequence Predicates

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

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

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

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

(*f* **unintern** *symbol* [*package*$_{v\texttt{*package*}}$])
▷ Remove *symbol* from *package*, return `T` on success.

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

(*f* **shadow** *symbols* [*package*$_{v\texttt{*package*}}$])
▷ Make *symbols* of *package* shadow any otherwise accessible, equally named symbols from other packages. Return `T`.

(*f* **package-shadowing-symbols** *package*)
▷ List of symbols of *package* that shadow any otherwise accessible, equally named symbols from other packages.

(*f* **export** *symbols* [*package*$_{v\texttt{*package*}}$])
▷ Make *symbols* external to *package*. Return `T`.

(*f* **unexport** *symbols* [*package*$_{v\texttt{*package*}}$])
▷ Revert *symbols* to internal status. Return `T`.

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

(*m* **with-package-iterator** (*foo packages* [**:internal**|**:external**|**:inherited**])
(**declare** *decl**)* *form*^{P}*)
▷ Return values of *form*s. In *form*s, successive invocations of (*foo*) return: `T` if a symbol is returned; a symbol from *packages*; accessibility (**:internal**, **:external**, or **:inherited**); and the package the symbol belongs to.

(*f* **require** *module* [*paths*$_{\texttt{NIL}}$])
▷ If not in *v* ***modules***, try *paths* to load *module* from. Signal **error** if unsuccessful. Deprecated.

(*f* **provide** *module*)
▷ If not already there, add *module* to *v* ***modules***. Deprecated.

v ***modules*** ▷ List of names of loaded modules.

14.3 Symbols

A **symbol** has the attributes *name*, home **package**, property list, and optionally value (of global constant or variable *name*) and function (**function**, macro, or special operator *name*).

(*f* **make-symbol** *name*)
▷ Make fresh, uninterned symbol *name*.

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

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

14 Packages and Symbols

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

14.1 Predicates

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

14.2 Packages

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

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

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

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

(*f*make-package *foo* { :nicknames (*nick**)[NIL] | :use (*used-package**) })
▷ Create package *foo*.

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

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

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

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

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

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

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

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

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

({*f*some | *f*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.

(*f*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

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

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

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

(*f*fill ~~*sequence*~~ *foo* { :start *start*[0] | :end *end*[NIL] })
▷ Return *sequence* after setting elements between *start* and *end* to *foo*.

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

(*f*count *foo sequence*
{ :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*.

({*f*count-if | *f*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*.

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

(*f*subseq *sequence start* [*end*[NIL]])
▷ Return subsequence of *sequence* between *start* and *end*. **setf**able.

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

(*f*reverse *sequence*)
(*f*nreverse ~~*sequence*~~) ▷ Return *sequence* in reverse order.

($\{_f$**find** | $_f$**position**$\}$ *foo sequence* {**:from-end** *bool*$_{\texttt{NIL}}$ | $\{$**:test** *function*$_{\#\text{'eql}}$ | **:test-not** *test*$\}$ | **:start** *start*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ | **:key** *function*})

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

($\{_f$**find-if** | $_f$**find-if-not** | $_f$**position-if** | $_f$**position-if-not**$\}$ *test sequence* {**:from-end** *bool*$_{\texttt{NIL}}$ | **:start** *start*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ | **:key** *function*})

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

($_f$**search** *sequence-a sequence-b* {**:from-end** *bool*$_{\texttt{NIL}}$ | $\{$**:test** *function*$_{\#\text{'eql}}$ | **:test-not** *function*$\}$ | **:start1** *start-a*$_{0}$ | **:start2** *start-b*$_{0}$ | **:end1** *end-a*$_{\texttt{NIL}}$ | **:end2** *end-b*$_{\texttt{NIL}}$ | **:key** *function*})

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

($\{_f$**remove** *foo sequence* | $_f$**delete** *foo* $\widetilde{sequence}\}$ {**:from-end** *bool*$_{\texttt{NIL}}$ | $\{$**:test** *function*$_{\#\text{'eql}}$ | **:test-not** *function*$\}$ | **:start** *start*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ | **:key** *function* | **:count** *count*$_{\texttt{NIL}}$})

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

($\{_f$**remove-if** | $_f$**remove-if-not**$\}$ *test sequence* | $\{_f$**delete-if** | $_f$**delete-if-not**$\}$ *test* $\widetilde{sequence}$} {**:from-end** *bool*$_{\texttt{NIL}}$ | **:start** *start*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ | **:key** *function* | **:count** *count*$_{\texttt{NIL}}$})

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

($\{_f$**remove-duplicates** *sequence* | $_f$**delete-duplicates** $\widetilde{sequence}\}$ {**:from-end** *bool*$_{\texttt{NIL}}$ | $\{$**:test** *function*$_{\#\text{'eql}}$ | **:test-not** *function*$\}$ | **:start** *start*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ | **:key** *function*})

▷ Make copy of *sequence* without duplicates.

($\{_f$**substitute** *new old sequence* | $_f$**nsubstitute** *new old* $\widetilde{sequence}\}$ {**:from-end** *bool*$_{\texttt{NIL}}$ | $\{$**:test** *function*$_{\#\text{'eql}}$ | **:test-not** *function*$\}$ | **:start** *start*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ | **:key** *function* | **:count** *count*$_{\texttt{NIL}}$})

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

($\{_f$**substitute-if** | $_f$**substitute-if-not**$\}$ *new test sequence* | $\{_f$**nsubstitute-if** | $_f$**nsubstitute-if-not**$\}$ *new test* $\widetilde{sequence}$} {**:from-end** *bool*$_{\texttt{NIL}}$ | **:start** *start*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ | **:key** *function* | **:count** *count*$_{\texttt{NIL}}$})

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

($_f$**parse-namestring** *foo* [*host* [*default-pathname*$_{v*\text{default-pathname-defaults}*}$ {**:start** *start*$_{0}$ | **:end** *end*$_{\texttt{NIL}}$ | **:junk-allowed** *bool*$_{\texttt{NIL}}$}]])

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

($_f$**merge-pathnames** *path-or-stream* [*default-path-or-stream*$_{v*\text{default-pathname-defaults}*}$ [*default-version*$_{\text{:newest}}$]])

▷ Return pathname made by filling in components missing in *path-or-stream* from *default-path-or-stream*.

$_v$***default-pathname-defaults***

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

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

($_f$**enough-namestring** *path-or-stream* [*root-path*$_{v*\text{default-pathname-defaults}*}$])

▷ Return minimal path string that sufficiently describes the path of *path-or-stream* relative to *root-path.*

($_f$**namestring** *path-or-stream*)
($_f$**file-namestring** *path-or-stream*)
($_f$**directory-namestring** *path-or-stream*)
($_f$**host-namestring** *path-or-stream*)

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

($_f$**translate-pathname** *path-or-stream wildcard-path-a wildcard-path-b*)

▷ Translate the path of *path-or-stream* from *wildcard-path-a* into *wildcard-path-b*. Return new path.

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

($_f$**logical-pathname** *logical-path-or-stream*)

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

($_f$**logical-pathname-translations** *logical-host*)

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

($_f$**load-logical-pathname-translations** *logical-host*)

▷ Load *logical-host*'s translations. Return NIL if already loaded; return T if successful.

($_f$**translate-logical-pathname** *path-or-stream*)

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

($_f$**probe-file** *file*)
($_f$**truename** *file*)

▷ Canonical name of *file*. If *file* does not exist, return NIL/signal **file-error**, respectively.

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

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

($_f$**file-length** *stream*) ▷ Return length of *stream*.

($_f$**rename-file** *foo bar*)

▷ Rename file *foo* to *bar*. Unspecified components of path *bar* default to those of *foo*. Return new pathname, old physical file name$_2$, and new physical file name.$_3$

($_f$**delete-file** *file*) ▷ Delete *file*. Return T.

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

(*m*with-open-file (*stream path open-arg**) (declare *decl**)* *form*P*)
▷ Use *f*open with *open-arg*s to temporarily create *stream* to *path*; return values of *form*s.

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

(*m*with-input-from-string (*foo string* { :index *index* | :start *start*0 | :end *end*NIL }) (declare *decl**)* *form*P*)
▷ Evaluate *form*s with *foo* locally bound to input string-stream from *string*. Return values of *form*s; store next reading position into *index*.

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

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

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

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

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

13.7 Pathnames and Files

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

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

(*f*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*.

(*f*map *type function sequence*+)
▷ Apply *function* successively to corresponding elements of the *sequence*s. Return values as a sequence of *type*. If *type* is NIL, return NIL.

(*f*map-into *result-sequence function sequence**)
▷ Store into *result-sequence* successively values of *function* applied to corresponding elements of the *sequence*s.

(*f*reduce *function sequence* { :initial-value *foo*NIL | :from-end *bool*NIL | :start *start*0 | :end *end*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.

(*f*copy-seq *sequence*)
▷ Copy of *sequence* with shared elements.

7 Hash Tables

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

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

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

(*f*make-hash-table { :test {*f*eq|*f*eql|*f*equal|*f*equalp}#'eql | :size *int* | :rehash-size *num* | :rehash-threshold *num* })
▷ Make a hash table.

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

(*f*hash-table-count *hash-table*)
▷ Number of entries in *hash-table*.

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

(*f*clrhash *hash-table*) ▷ Empty *hash-table*.

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

(*m*with-hash-table-iterator (*foo hash-table*) (declare *decl**)* *form*P*)
▷ Return values of *form*s. In *form*s, invocations of (*foo*) return: T if an entry is returned; its key; its value.

(*f*hash-table-test *hash-table*)
▷ Test function used in *hash-table*.

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

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

8 Structures

(м**defstruct**
foo
(*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 page 18) 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 ɡ**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.

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

9 Control Structure

9.1 Predicates

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

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

(ƒ**equal** *foo bar*)
▷ `T` if *foo* and *bar* are ƒ**eql**, or are equivalent **pathname**s, or are **cons**es with ƒ**equal** cars and cdrs, or are **string**s or **bit-vector**s with ƒ**eql** elements below their fill pointers.

(ƒ**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 ƒ**equalp** elements; or are structures of the same type with ƒ**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 ƒ**equalp** elements.

13.6 Streams

(ƒ**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*.

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

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

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

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

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

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

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

(ƒ**file-position** *stream* [**:start** | **:end** | *position*])
▷ Return position within stream, or set it to *position* and return `T` on success.

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

(ƒ**listen** [*stream* [v***standard-input***]])
▷ `T` if there is a character in input *stream*.

(ƒ**clear-input** [*stream* [v***standard-input***]])
▷ Clear input from *stream*, return `NIL`.

(ƒ**clear-output** | ƒ**force-output** | ƒ**finish-output**) [*stream* [v***standard-output***]])
▷ End output to *stream* and return `NIL` immediately, after initiating flushing of buffers, or after flushing of buffers, respectively.

~ [:] [@] < {[*prefix*$_{\boxed{\text{""}}}$ ~;]|[*per-line-prefix* ~@;]} *body* [~; *suffix*$_{\boxed{\text{""}}}$] ~: [@] >
▷ **Logical Block.** Act like **pprint-logical-block** using *body* as $_f$**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 $_f$**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 $_f$**format** control subclause. With :, use the first *text* if the argument value is NIL, or the second *text* if it is T. With @, do nothing for an argument value of NIL. Use the only *text* and leave the argument to be read again if it is T.

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

~ [*prefix* {,*prefix*}*] [:] [@] /[*package* [:]:$_{\boxed{\text{cl-user:}}}$]*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.

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

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

($_f$**constantp** *foo* [*environment*$_{\boxed{\text{NIL}}}$])
▷ T if *foo* is a constant form.

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

($_f$**fboundp** {*foo* | (**setf** *foo*)}) ▷ T if *foo* is a global function or macro.

9.2 Variables

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

($_m$**defvar** $\widehat{foo}$ [*form* [$\widehat{doc}$]])
▷ 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.

({$_s$**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.

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

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

($_m$**shiftf** $\widetilde{place}^+$ *foo*)
▷ Store value of *foo* in rightmost *place* shifting values of *place*s left, returning first *place*.

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

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

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

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

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

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

(${}_s$**let** | ${}_s$**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.

(${}_m$**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.

(${}_m$**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.

9.3 Functions

Below, ordinary lambda list (*ord-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**]]
[**&aux** *var* | (*var* [*init*NIL])*]).

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

(${}_m$**defun** *foo* (*ord-λ**) | (**setf** *foo*) (*new-value ord-λ**) | ${}_m$**lambda** (*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 ${}_s$**block** named *foo*.

(${}_s$**flet** | ${}_s$**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 ${}_s$**block** around its corresponding *local-form**. Only for ${}_s$**labels**, functions *foo* are visible inside *local-form*s. Return values of *form*s.

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

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

(${}_f$**funcall** *function* arg*) ▷ Values of *function* called with *arg*s.

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

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

(${}_f$**values** *foo**)
▷ Return as multiple values the primary values of the *foo*s. **setf**able.

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

~ [*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*] [,[*dec-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 *dec-digits* after decimal point and *exp-digits* in the signed exponent. With **~G**, choose either **~E** or **~F**. With **@**, always prepend a sign.

~ [*dec-digits*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.

{~**(** *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*1] **%** ▷ **Newline.** Print *n* newlines.

~ [*n*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*1] | ▷ **Page.** Print *n* page separators.

~ [*n*1] ~ ▷ **Tilde.** Print *n* tildes.

~ [*min-col*0] [,[*col-inc*1] [,[*min-pad*0] [,'*pad-char*␣]]] [:] [@] **<**
[*nl-text* ~[*spare*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.

$_v$***print-array*** ▷ If `T`, print arrays $_f$**read**ably.

$_v$***print-base***$_{\boxed{10}}$ ▷ Radix for printing rationals, from 2 to 36.

$_v$***print-case***$_{\boxed{:upcase}}$
▷ Print symbol names all uppercase (**:upcase**), all lowercase (**:downcase**), capitalized (**:capitalize**).

$_v$***print-circle***$_{\boxed{NIL}}$
▷ If `T`, avoid indefinite recursion while printing circular structure.

$_v$***print-escape***$_{\boxed{T}}$
▷ If `NIL`, do not print escape characters and package prefixes.

$_v$***print-gensym***$_{\boxed{T}}$ ▷ If `T`, print **#:** before uninterned symbols.

$_v$***print-length***$_{\boxed{NIL}}$
$_v$***print-level***$_{\boxed{NIL}}$
$_v$***print-lines***$_{\boxed{NIL}}$
▷ If integer, restrict printing of objects to that number of elements per level/to that depth/to that number of lines.

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

$_v$***print-pretty*** ▷ If `T`, print prettily.

$_v$***print-radix***$_{\boxed{NIL}}$ ▷ If `T`, print rationals with a radix indicator.

$_v$***print-readably***$_{\boxed{NIL}}$
▷ If `T`, print $_f$**read**ably or signal error **print-not-readable**.

$_v$***print-right-margin***$_{\boxed{NIL}}$
▷ Right margin width in ems while pretty-printing.

($_f$**set-pprint-dispatch** *type function* [*priority*$_{\boxed{0}}$ [*table*$_{\boxed{v*print-pprint-dispatch*}}$]])
▷ Install entry comprising *function* of arguments stream and object to print; and *priority* as *type* into *table*. If *function* is `NIL`, remove *type* from *table*. Return `NIL`.

($_f$**pprint-dispatch** *foo* [*table*$_{\boxed{v*print-pprint-dispatch*}}$])
▷ Return highest priority *function* associated with type of *foo* and `T` if there was a matching type specifier in *table*.

($_f$**copy-pprint-dispatch** [*table*$_{\boxed{v*print-pprint-dispatch*}}$])
▷ Return copy of *table* or, if *table* is `NIL`, initial value of $_v$***print-pprint-dispatch***.

$_v$***print-pprint-dispatch*** ▷ Current pretty print dispatch table.

13.5 Format

($_m$**formatter** $\widehat{control}$)
▷ Return function of *stream* and *arg** applying $_f$**format** to *stream*, *control*, and *arg** returning `NIL` or any excess *arg*s.

($_f$**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 $_v$***standard-output***. Return `NIL`. If first argument is `NIL`, return formatted output.

`~` [*min-col*$_{\boxed{0}}$] [`,`[*col-inc*$_{\boxed{1}}$] [`,`[*min-pad*$_{\boxed{0}}$] [`,'`*pad-char*$_{\boxed{\ }}$]]]
[`:`] [**@**] {**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.

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

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

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

($_f$**identity** *foo*) ▷ Return *foo*.

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

($_f$**fdefinition** $\begin{Bmatrix} foo \\ (\textbf{setf}\ foo) \end{Bmatrix}$)
▷ Definition of global function *foo*. **setf**able.

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

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

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

9.4 Macros

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

([**&whole** *var*] [*E*] $\begin{Bmatrix} var \\ (macro\text{-}\lambda^*) \end{Bmatrix}^*$ [*E*]
[**&optional** $\left\{ \begin{matrix} var \\ (\begin{Bmatrix} var \\ (macro\text{-}\lambda^*) \end{Bmatrix}\ [init_{\boxed{NIL}}\ [supplied\text{-}p]]) \end{matrix} \right\}^*$] [*E*]
[$\begin{Bmatrix} \textbf{\&rest} \\ \textbf{\&body} \end{Bmatrix}$ $\begin{Bmatrix} rest\text{-}var \\ (macro\text{-}\lambda^*) \end{Bmatrix}$] [*E*]
[**&key** $\left\{ \begin{matrix} var \\ (\begin{Bmatrix} var \\ (\textbf{:}key\ \begin{Bmatrix} var \\ (macro\text{-}\lambda^*) \end{Bmatrix}) \end{Bmatrix}\ [init_{\boxed{NIL}}\ [supplied\text{-}p]]) \end{matrix} \right\}^*$ [*E*]
[**&allow-other-keys**]] [**&aux** $\begin{Bmatrix} var \\ (var\ [init_{\boxed{NIL}}]) \end{Bmatrix}^*$] [*E*])
or
([**&whole** *var*] [*E*] $\begin{Bmatrix} var \\ (macro\text{-}\lambda^*) \end{Bmatrix}^*$ [*E*]
[**&optional** $\left\{ \begin{matrix} var \\ (\begin{Bmatrix} var \\ (macro\text{-}\lambda^*) \end{Bmatrix}\ [init_{\boxed{NIL}}\ [supplied\text{-}p]]) \end{matrix} \right\}^*$] [*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.

($\begin{Bmatrix} _m\textbf{defmacro} \\ _f\textbf{define-compiler-macro} \end{Bmatrix}$ $\begin{Bmatrix} foo \\ (\textbf{setf}\ foo) \end{Bmatrix}$ (*macro*-λ*)
$\left\{ \begin{matrix} (\textbf{declare}\ \widehat{decl^*})^* \\ \widehat{doc} \end{matrix} \right\}$ $form^{P_*}$)
▷ Define macro *foo* which on evaluation as (*foo tree*) applies expanded *form*s to arguments from *tree*, which corresponds to *tree*-shaped *macro*-λs. *form*s are enclosed in an implicit $_s$**block** named *foo*.

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

(*s*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 *s*block**s** of the same name.

(*s*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 *s*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 *f*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 *s*block named *function*.

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

(*m*define-modify-macro *foo* ([&optional {*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.

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

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

&optional *var**
▷ Bind *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.

(*f*write-sequence *sequence* *stream* {:start *start*0 | :end *end*NIL})
▷ Write elements of *sequence* to binary or character *stream*.

({*f*write | *f*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*v*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 *f*write only.)

(*f*pprint-fill *stream foo* [*parenthesis*T [*noop*]])
(*f*pprint-tabular *stream foo* [*parenthesis*T [*noop* [*n*16]]])
(*f*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 *f*format directive ~//.

(*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 *f*write. Return NIL.

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

(*f*pprint-tab {:line | :line-relative | :section | :section-relative} *c i* [*stream*v*standard-output*])
▷ Move cursor forward to column number $c + ki$, $k \geq 0$ being as small as possible.

(*f*pprint-indent {:block | :current} *n* [*stream*v*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.

(*f*pprint-newline {:linear | :fill | :miser | :mandatory} [*stream*v*standard-output*])
▷ Print a conditional newline if *stream* is a pretty printing stream. Return NIL.

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

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

#P$string$ ▷ A pathname.

#:foo ▷ Uninterned symbol foo.

#.$form$ ▷ Read-time value of $form$.

$_v$***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**.

#+$feature$ $when\text{-}feature$
#−$feature$ $unless\text{-}feature$
▷ Means $when\text{-}feature$ if $feature$ is T; means $unless\text{-}feature$ if $feature$ is NIL. $feature$ is a symbol from $_v$***features***, or ({**and**|**or**} $feature^*$), or (**not** $feature$).

$_v$***features***
▷ List of symbols denoting implementation-dependent features.

|c^*|; \c
▷ Treat arbitrary character(s) c as alphabetic preserving case.

13.4 Printer

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

($_f$**prin1-to-string** foo)
($_f$**princ-to-string** foo)
▷ Print foo to $string$ $_f$**read**ably or human-readably, respectively.

($_g$**print-object** $object$ $\widetilde{stream}$)
▷ Print $object$ to $stream$. Called by the Lisp printer.

($_m$**print-unreadable-object** (foo $\widetilde{stream}$ {|**:type** $bool$[NIL]|**:identity** $bool$[NIL]}) $form^{P*}$)
▷ Enclosed in **#<** and **>**, print foo by means of $form$s to $stream$. Return NIL.

($_f$**terpri** [$\widetilde{stream}$[$_v$***standard-output***]])
▷ Output a newline to $stream$. Return NIL.

($_f$**fresh-line** [$\widetilde{stream}$[$_v$***standard-output***]])
▷ Output a newline to $stream$ and return T unless $stream$ is already at the start of a line.

($_f$**write-char** $char$ [$\widetilde{stream}$[$_v$***standard-output***]])
▷ Output $char$ to $stream$.

({$_f$**write-string**|$_f$**write-line**} $string$ [$\widetilde{stream}$[$_v$***standard-output***] [{|**:start** $start$[0]|**:end** end[NIL]}]])
▷ Write $string$ to $stream$ without/with a trailing newline.

($_f$**write-byte** $byte$ $\widetilde{stream}$) ▷ Write $byte$ to binary $stream$.

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

&environment var
▷ Bind var to the lexical compilation environment.

&aux var^* ▷ Bind vars as in $_s$**let***.

9.5 Control Flow

($_s$**if** $test$ $then$ [$else$[NIL]])
▷ Return values of $then$ if $test$ returns T; return values of $else$ otherwise.

($_m$**cond** ($test$ $then^{P*}$[$test$])*)
▷ Return the values of the first $then^*$ whose $test$ returns T; return NIL if all $test$s return NIL.

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

($_m$**case** $test$ ({$\widehat{(key^*)}$|$\widehat{key}$} foo^{P*})* [({**otherwise**|T} bar^{P*})[NIL]])
▷ Return the values of the first foo^* one of whose keys is **eql** $test$. Return values of bars if there is no matching key.

({$_m$**ecase**|$_m$**ccase**} $test$ ({$\widehat{(key^*)}$|$\widehat{key}$} foo^{P*})*)
▷ Return the values of the first foo^* one of whose keys is **eql** $test$. Signal non-correctable/correctable **type-error** 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.

($_s$**progn** $form^*$[NIL])
▷ Evaluate $form$s sequentially. Return values of last $form$.

($_s$**multiple-value-prog1** $form\text{-}r$ $form^*$)
($_m$**prog1** $form\text{-}r$ $form^*$)
($_m$**prog2** $form\text{-}a$ $form\text{-}r$ $form^*$)
▷ Evaluate forms in order. Return values/primary value, respectively, of $form\text{-}r$.

({$_m$**prog**|$_m$**prog***} ({|$name$|($name$ [$value$[NIL]])}*) (**declare** $\widehat{decl^*}$)* {$\widehat{tag}$|$form$}*)
▷ Evaluate $_s$**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 $_s$**block** named NIL.

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

($_s$**block** $name$ $form^{P*}$)
▷ Evaluate $form$s in a lexical environment, and return their values unless interrupted by $_s$**return-from**.

($_s$**return-from** foo [$result$[NIL]])
($_m$**return** [$result$[NIL]])
▷ Have nearest enclosing $_s$**block** named foo/named NIL, respectively, return with values of $result$.

(*s*__tagbody__ {$\widehat{tag}$|*form*}*)
▷ Evaluate *form*s in a lexical environment. *tag*s (symbols or integers) have lexical scope and dynamic extent, and are targets for *s*__go__. Return NIL.

(*s*__go__ $\widehat{tag}$)
▷ Within the innermost possible enclosing *s*__tagbody__, jump to a tag *f*__eql__ *tag*.

(*s*__catch__ *tag* *form*P*)
▷ Evaluate *form*s and return their values unless interrupted by *s*__throw__.

(*s*__throw__ *tag* *form*)
▷ Have the nearest dynamically enclosing *s*__catch__ with a tag *f*__eq__ *tag* return with the values of *form*.

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

9.6 Iteration

({*m*__do__|*m*__do*__} ({*var*|(*var* [*start* [*step*]])}*) (*stop* *result*P*) (__declare__ $\widehat{decl}$*)* {$\widehat{tag}$|*form*}*)
▷ Evaluate *s*__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 *s*__block__ named NIL.

(*m*__dotimes__ (*var* *i* [*result*]) (__declare__ $\widehat{decl}$*)* {$\widehat{tag}$|*form*}*)
▷ Evaluate *s*__tagbody__-like body with *var* successively bound to integers from 0 to $i-1$. Upon evaluation of *result*, *var* is *i*. Implicitly, the whole form is a *s*__block__ named NIL.

(*m*__dolist__ (*var* *list* [*result*]) (__declare__ $\widehat{decl}$*)* {$\widehat{tag}$|*form*}*)
▷ Evaluate *s*__tagbody__-like body with *var* successively bound to the elements of *list*. Upon evaluation of *result*, *var* is NIL. Implicitly, the whole form is a *s*__block__ named NIL.

9.7 Loop Facility

(*m*__loop__ *form**)
▷ **Simple Loop.** If *form*s do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit *s*__block__ named NIL.

(*m*__loop__ *clause**)
▷ **Loop Facility.** For Loop Facility keywords see below and Figure 1.

__named__ *n* ▷ Give *m*__loop__'s implicit *s*__block__ a name.

{__with__ {*var-s*|(*var-s**)} [*d-type*] [= *foo*]}+
{__and__ {*var-p*|(*var-p**)} [*d-type*] [= *bar*]}*
where destructuring type specifier *d-type* has the form
{__fixnum__|__float__|T|NIL|{__of-type__ {*type*|(*type**)}}}
▷ Initialize (possibly trees of) local variables *var-s* sequentially and *var-p* in parallel.

{{__for__|__as__} {*var-s*|(*var-s**)} [*d-type*]}+ {__and__ {*var-p*|(*var-p**)} [*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__.

*v*__*read-base*__ ▷ Radix for reading __integer__s and __ratio__s.

*v*__*read-default-float-format*__
▷ Floating point format to use when not indicated in the number read.

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

(*f*__set-macro-character__ *char* *function* [*non-term-p*] [$\widetilde{rt}$])
▷ Make *char* a macro character associated with *function* of stream and *char*. Return T.

(*f*__get-macro-character__ *char* [*rt*])
▷ Reader macro function associated with *char*, and T if *char* is a non-terminating macro character.

(*f*__make-dispatch-macro-character__ *char* [*non-term-p*] [*rt*])
▷ Make *char* a dispatching macro character. Return T.

(*f*__set-dispatch-macro-character__ *char* *sub-char* *function* [$\widetilde{rt}$])
▷ Make *function* of stream, *n*, *sub-char* a dispatch function of *char* followed by *n*, followed by *sub-char*. Return T.

(*f*__get-dispatch-macro-character__ *char* *sub-char* [*rt*])
▷ Dispatch function associated with *char* followed by *sub-char*.

13.3 Character Syntax

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

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

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

" ▷ Begin and end of a string.

'*foo* ▷ (*s*__quote__ *foo*); *foo* unevaluated.

`([*foo*] [,*bar*] [,@*baz*] [,.$\widetilde{quux}$] [*bing*])
▷ Backquote. *s*__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* ▷ (*f*__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*]|*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*) ▷ (*f*__complex__ *a* *b*), the complex number $a + b\mathrm{i}$.

#'*foo* ▷ (*s*__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.

13.2 Reader

($_f$**y-or-n-p** | $_f$**yes-or-no-p**) [*control arg**])
▷ Ask user a question and return `T` or `NIL` depending on their answer. See page 38, $_f$**format**, for *control* and *arg*s.

($_m$**with-standard-io-syntax** *form*^{P*})
▷ Evaluate *form*s with standard behaviour of reader and printer. Return values of *form*s.

($_f$**read** | $_f$**read-preserving-whitespace**) [*stream*$_{v*standard-input*}$ [*eof-err*$_T$ [*eof-val*$_{NIL}$ [*recursive*$_{NIL}$]]]])
▷ Read printed representation of object.

($_f$**read-from-string** *string* [*eof-error*$_T$ [*eof-val*$_{NIL}$ [{**:start** *start*$_0$ | **:end** *end*$_{NIL}$ | **:preserve-whitespace** *bool*$_{NIL}$}]]])
▷ Return object read from string and zero-indexed position of next character.

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

($_f$**read-char** [*stream*$_{v*standard-input*}$ [*eof-err*$_T$ [*eof-val*$_{NIL}$ [*recursive*$_{NIL}$]]]])
▷ Return next character from *stream*.

($_f$**read-char-no-hang** [*stream*$_{v*standard-input*}$ [*eof-error*$_T$ [*eof-val*$_{NIL}$ [*recursive*$_{NIL}$]]]])
▷ Next character from *stream* or `NIL` if none is available.

($_f$**peek-char** [*mode*$_{NIL}$ [*stream*$_{v*standard-input*}$ [*eof-error*$_T$ [*eof-val*$_{NIL}$ [*recursive*$_{NIL}$]]]]])
▷ Next, or if *mode* is `T`, next non-whitespace character, or if *mode* is a character, next instance of it, from *stream* without removing it there.

($_f$**unread-char** *character* [*stream*$_{v*standard-input*}$])
▷ Put last $_f$**read-char**ed *character* back into *stream*; return `NIL`.

($_f$**read-byte** *stream* [*eof-err*$_T$ [*eof-val*$_{NIL}$]])
▷ Read next byte from binary *stream*.

($_f$**read-line** [*stream*$_{v*standard-input*}$ [*eof-err*$_T$ [*eof-val*$_{NIL}$ [*recursive*$_{NIL}$]]]])
▷ Return a line of text from *stream* and `T` if line has been ended by end of file.

($_f$**read-sequence** *sequence stream* [**:start** *start*$_0$][**:end** *end*$_{NIL}$])
▷ Replace elements of *sequence* between *start* and *end* with elements from binary or character *stream*. Return index of *sequence*'s first unmodified element.

($_f$**readtable-case** *readtable*)$_{:upcase}$
▷ Case sensitivity attribute (one of **:upcase**, **:downcase**, **:preserve**, **:invert**) of *readtable*. **setf**able.

($_f$**copy-readtable** [*from-readtable*$_{v*readtable*}$ [*to-readtable*$_{NIL}$]])
▷ Return copy of *from-readtable*.

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

$_v$***readtable*** ▷ Current readtable.

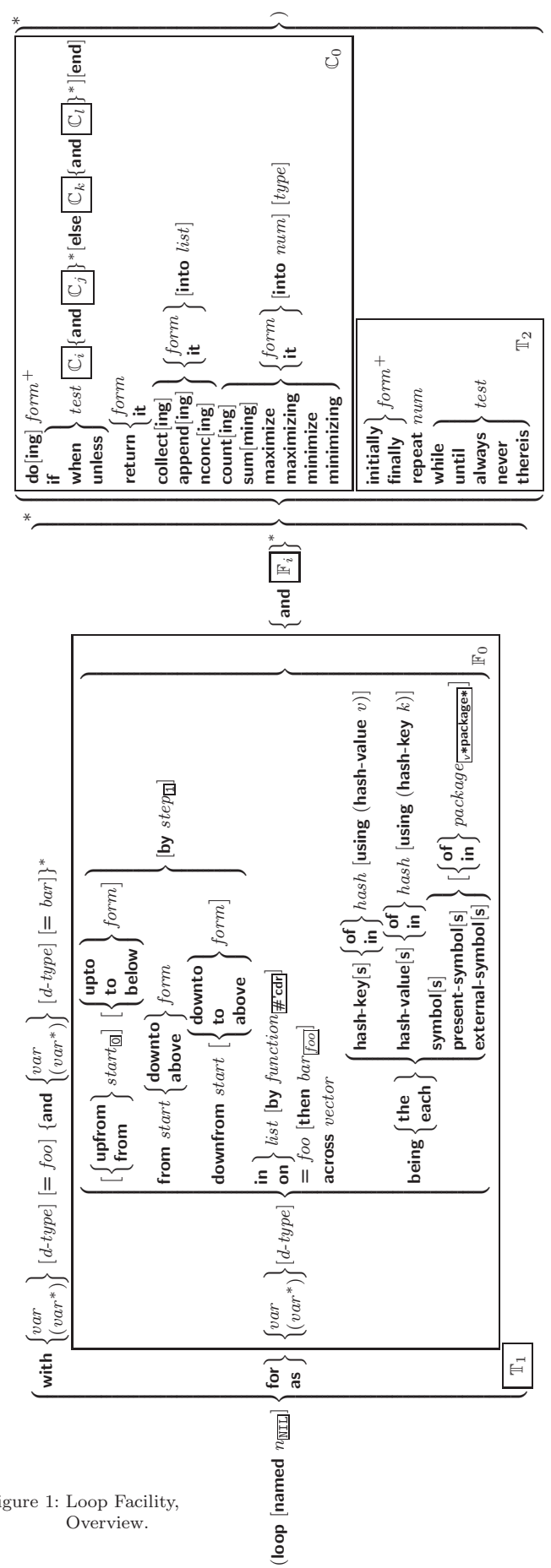

Figure 1: Loop Facility, Overview.

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

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

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

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

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

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

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

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

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

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

{**do**|**doing**} *form*$^+$ ▷ Evaluate *form*s 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-clause*s; otherwise, evaluate *k-clause* and *l-clause*s.

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

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

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

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

($_f$**type-of** *foo*) ▷ Type of *foo*.

($_m$**check-type** *place type* [*string*[{**a**|**an**} *type*]])
▷ Signal correctable **type-error** if *place* is not of *type*. Return `NIL`.

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

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

($_f$**upgraded-array-element-type** *type* [*environment*[`NIL`]])
▷ Element type of most specialized array capable of holding elements of *type*.

($_m$**deftype** *foo* (*macro-λ**) $\left\{\left|\begin{array}{l}(\textbf{declare}\ \widehat{decl^*})^*\\ \widehat{doc}\end{array}\right.\right\}$ *form*^{P_*})
▷ Define type *foo* which when referenced as (*foo* $\widehat{arg^*}$) (or as *foo* if *macro-λ* doesn't contain any required parameters) applies expanded *form*s to *arg*s returning the new type. For (*macro-λ**) see page 19 but with default value of ∗ instead of `NIL`. *form*s are enclosed in an implicit $_s$**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

($_f$**streamp** *foo*)
($_f$**pathnamep** *foo*)
($_f$**readtablep** *foo*) ▷ `T` if *foo* is of indicated type.

($_f$**input-stream-p** *stream*)
($_f$**output-stream-p** *stream*)
($_f$**interactive-stream-p** *stream*)
($_f$**open-stream-p** *stream*)
▷ Return `T` if *stream* is for input, for output, interactive, or open, respectively.

($_f$**pathname-match-p** *path wildcard*)
▷ `T` if *path* matches *wildcard*.

($_f$**wild-pathname-p** *path* [{**:host**|**:device**|**:directory**|**:name**|**:type**|**:version**|`NIL`}])
▷ Return `T` if indicated component in *path* is wildcard. (`NIL` indicates any component.)

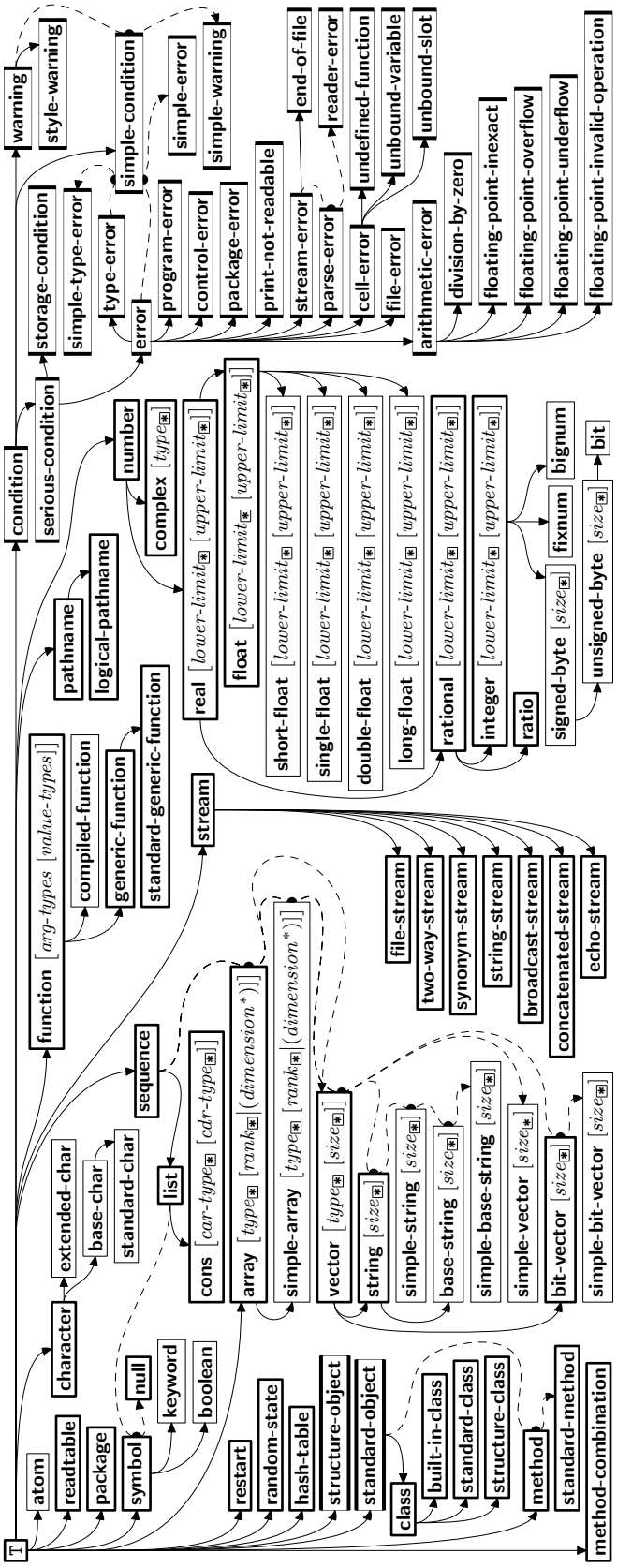

Figure 2: Precedence Order of System Classes (▭), Classes (▬), Types (▭), and Condition Types (▭).
Every type is also a supertype of NIL, the empty type.

{**initially**|**finally**} *form*$^+$
▷ Evaluate *form*s 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

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

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

($_m$**defclass** *foo* (*superclass** **standard-object**)
$$\left(\left\{\begin{array}{l}slot\\ \left(slot\ \left\{\begin{array}{l}\{\textbf{:reader}\ reader\}^*\\ \{\textbf{:writer}\ \left\{\begin{array}{l}writer\\ (\textbf{setf}\ writer)\end{array}\right\}\}^*\\ \{\textbf{:accessor}\ accessor\}^*\\ \textbf{:allocation}\ \left\{\begin{array}{l}\textbf{:instance}\\ \textbf{:class}\end{array}\right\}\ \textbf{:instance}\\ \{\textbf{:initarg}\ :initarg\text{-}name\}^*\\ \textbf{:initform}\ form\\ \textbf{:type}\ type\\ \textbf{:documentation}\ slot\text{-}doc\end{array}\right)\end{array}\right\}^*\right)$$
$$\left(\left\{\begin{array}{l}(\textbf{:default-initargs}\ \{name\ value\}^*)\\ (\textbf{:documentation}\ class\text{-}doc)\\ (\textbf{:metaclass}\ name\ \textbf{standard-class})\end{array}\right\}\right)$$
▷ Define or modify class *foo* as a subclass of *superclass*es. Transform existing instances, if any, by $_g$**make-instances-obsolete**. In a new instance *i* of *foo*, a *slot*'s value defaults to *form* unless set via **:***initarg-name*; it is readable via (*reader i*) or (*accessor i*), and writable via (*writer value i*) or (**setf** (*accessor i*) *value*). *slot*s with **:allocation :class** are shared by all instances of class *foo*.

($_f$**find-class** *symbol* [*errorp*$_{\boxed{T}}$ [*environment*]])
▷ Return class named *symbol*. **setf**able.

($_g$**make-instance** *class* {**:***initarg value*}* *other-keyarg**)
▷ Make new instance of *class*.

($_g$**reinitialize-instance** *instance* {**:***initarg value*}* *other-keyarg**)
▷ Change local slots of *instance* according to *initarg*s by means of $_g$**shared-initialize**.

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

($_f$**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.

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

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

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

(${}_g$**make-instances-obsolete** *class*)
▷ Update all existing instances of *class* using ${}_g$**update-instance-for-redefined-class**.

(${}_g$**initialize-instance** / ${}_g$**update-instance-for-different-class** *previous current*) {**:***initarg value*}* *other-keyarg**)
▷ Set slots on behalf of ${}_g$**make-instance**/of ${}_g$**change-class** by means of ${}_g$**shared-initialize**.

(${}_g$**update-instance-for-redefined-class** *new-instance added-slots discarded-slots discarded-slots-property-list* {**:***initarg value*}* *other-keyarg**)
▷ On behalf of ${}_g$**make-instances-obsolete** and by means of ${}_g$**shared-initialize**, set any *initarg* slots to their corresponding *value*s; set any remaining *added-slots* to the values of their **:initform** forms. Not to be called by user.

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

(${}_g$**shared-initialize** *instance* *initform-slots*/T {**:***initarg-slot value*}* *other-keyarg**)
▷ Fill the *initarg-slot*s of *instance* with the corresponding *value*s, and fill those *initform-slots* that are not *initarg-slot*s with the values of their **:initform** forms.

(${}_g$**slot-missing** *class instance slot* **setf**/**slot-boundp**/**slot-makunbound**/**slot-value** [*value*])
(${}_g$**slot-unbound** *class instance slot*)
▷ Called on attempted access to non-existing or unbound *slot*. Default methods signal **error**/**unbound-slot**, respectively. Not to be called by user.

10.2 Generic Functions

(${}_f$**next-method-p**) ▷ T if enclosing method has a next method.

(${}_m$**defgeneric** *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** *method-selection-optimization*)+)
(**:documentation** *string*)
(**:generic-function-class** *gf-class*[standard-generic-function])
(**:method-class** *method-class*[standard-method])
(**:method-combination** *c-type*[standard] *c-arg**)
(**:method** *defmethod-args*)*
)

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

(${}_m$**with-condition-restarts** *condition restarts form*P*)
▷ Evaluate *form*s with *restarts* dynamically associated with *condition*. Return values of *form*s.

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

(${}_f$**cell-error-name** *condition*)
▷ Name of cell which caused *condition*.

(${}_f$**unbound-slot-instance** *condition*)
▷ Instance with unbound slot which caused *condition*.

(${}_f$**print-not-readable-object** *condition*)
▷ The object not readably printable under *condition*.

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

(${}_f$**type-error-datum** *condition*)
(${}_f$**type-error-expected-type** *condition*)
▷ Object which caused *condition* of type **type-error**, or its expected type, respectively.

(${}_f$**simple-condition-format-control** *condition*)
(${}_f$**simple-condition-format-arguments** *condition*)
▷ Return ${}_f$**format** control or list of ${}_f$**format** arguments, respectively, of *condition*.

${}_v$***break-on-signals***[NIL]
▷ Condition type debugger is to be invoked on.

${}_v$***debugger-hook***[NIL]
▷ Function of condition and function itself. Called before debugger.

12 Types and Classes

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

(${}_f$**typep** *foo type* [*environment*[NIL]]) ▷ T if *foo* is of *type*.

(${}_f$**subtypep** *type-a type-b* [*environment*])
▷ Return T if *type-a* is a recognizable subtype of *type-b*, and NIL if the relationship could not be determined.
2

(${}_s$**the** *type form*) ▷ Declare values of *form* to be of *type*.

(${}_f$**coerce** *object type*) ▷ Coerce *object* into *type*.

(${}_m$**typecase** *foo* (*type* *a-form*P*)* [(**otherwise**/T *b-form*[NIL]P*)])
▷ Return values of the first *a-form** whose *type* is *foo* of. Return values of *b-forms* if no *type* matches.

(${}_m$**etypecase**/${}_m$**ctypecase** *foo* (*type* *form*P*)*)
▷ Return values of the first *form** whose *type* is *foo* of. Signal non-correctable/correctable **type-error** if no *type* matches.

(*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 page 18 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 *f*format *control* and *arg*s (see page 38) and return NIL and T.

(*m*restart-case *form* (*restart* (*ord-λ**)
{:interactive *arg-function* | :report {*report-function* | *string* "restart"} | :test *test-function* T}
(declare $\widehat{decl^*}$)* *restart-form*^{P_*})*)
▷ Return values of *form* or, if during evaluation of *form* one of the dynamically established *restart*s is called, the values of its *restart-form*s. A *restart* is visible under *condition* if (funcall #'*test-function condition*) returns T. If presented in the debugger, *restart*s are described by *string* or by #'*report-function* (of a stream). A *restart* can be called by (invoke-restart *restart arg**), where *arg*s match *ord-λ**, or by (invoke-restart-interactively *restart*) where a list of the respective *arg*s is supplied by #'*arg-function*. See page 18 for *ord-λ**.

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

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

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

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

({*f*abort | *f*muffle-warning | *f*continue | *f*store-value *value* | *f*use-value *value*} [*condition* NIL])

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

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

(*m*defmethod {*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*block *foo*. Applicable *qualifier*s depend on the method-combination type; see section 10.3.

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

(*g*find-method *generic-function qualifiers specializers* [*error*$_T$])
▷ Return suitable method, or signal error.

(*g*compute-applicable-methods *generic-function args*)
▷ List of methods suitable for *args*, most specific first.

(*f*call-next-method *arg** current args)
▷ From within a method, call next method with *arg*s; return its values.

(*g*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. Not to be called by user.

({*f*invalid-method-error *method* | *f*method-combination-error} *control arg**)
▷ Signal error on applicable method with invalid qualifiers, or on method combination. For *control* and *arg*s see format, page 38.

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

(*g*function-keywords *method*)
▷ Return list of keyword parameters of *method* and T if other keys are allowed.

(*g*method-qualifiers *method*) ▷ List of qualifiers of *method*.

10.3 Method Combination Types

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

and|**or**|**append**|**list**|**nconc**|**progn**|**max**|**min**|**+**
▷ Simple built-in **method-combination** types; have the same usage as the *c-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, *f*call-next-method can call less specific **:around** methods if there are any. If not, or if there are no **:around** methods at all, return from the calling **call-next-method** or from the generic function, respectively, the values of (*operator* (*primary-method gen-arg**)*), *gen-arg** being the arguments of the generic function. The *primary-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 page 18, 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 32.

(*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 writable via (*writer value i*) or (**setf** (*accessor i*) *value*). With **:allocation :class**, *slot* is shared by all conditions of type *foo*. A condition is reported by *string* or by *report-function* of arguments condition and stream.

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

({ *f*signal | *f*warn | *f*error } { *condition* | *condition-type* {*:initarg-name value*}* | *control arg** })
▷ Unless handled, signal as **condition**, **warning** or **error**, respectively, *condition* or a new instance of *condition-type* or, with *f*format *control* and *arg*s (see page 38), **simple-condition**, **simple-warning**, or **simple-error**, respectively. From *f*signal and *f*warn, return NIL.

(*f*cerror *continue-control* { *condition continue-arg** | *condition-type* {*:initarg-name value*}* | *control arg** })
▷ Unless handled, signal as correctable **error** *condition* or a new instance of *condition-type* or, with *f*format *control* and *arg*s (see page 38), **simple-error**. In the debugger, use *f*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

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

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