Quick Reference

Common
lisp

Bert Burgemeister

Common Lisp Quick Reference Revision 107 [2009-10-25]
Copyright © 2008, 2009 Bert Burgemeister
LaTeX source: http://clqr.berlios.de

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts and no Back-Cover Texts. http://www.gnu.org/licenses/fdl.html

Contents

1 Numbers 3
- 1.1 Predicates 3
- 1.2 Numeric Functns . 3
- 1.3 Logic Functions . 5
- 1.4 Integer Functions . 5
- 1.5 Implementation-Dependent 6

2 Characters 6

3 Strings 7

4 Conses 8
- 4.1 Predicates 8
- 4.2 Lists 9
- 4.3 Association Lists . 10
- 4.4 Trees 10
- 4.5 Sets 11

5 Arrays 11
- 5.1 Predicates 11
- 5.2 Array Functions . 11
- 5.3 Vector Functions . 12

6 Sequences 12
- 6.1 Seq. Predicates . . 12
- 6.2 Seq. Functions . . 13

7 Hash Tables 15

8 Structures 15

9 Control Structure 16
- 9.1 Predicates 16
- 9.2 Variables 16
- 9.3 Functions 17
- 9.4 Macros 18
- 9.5 Control Flow . . . 20
- 9.6 Iteration 21
- 9.7 Loop Facility . . . 22

10 CLOS 24
- 10.1 Classes 24
- 10.2 Generic Functns . 26
- 10.3 Method Combination Types . . . 27

11 Conditions and Errors 28

12 Input/Output 31
- 12.1 Predicates 31
- 12.2 Reader 31
- 12.3 Macro Chars . . . 32
- 12.4 Printer 33
- 12.5 Format 35
- 12.6 Streams 38
- 12.7 Files 39

13 Types and Classes 40

14 Packages and Symbols 42
- 14.1 Predicates 42
- 14.2 Packages 43
- 14.3 Symbols 44
- 14.4 Std Packages . . 45

15 Compiler 45
- 15.1 Predicates 45
- 15.2 Compilation . . . 45
- 15.3 REPL & Debug . 46
- 15.4 Declarations . . . 47

16 External Environment 48

Typographic Conventions

name; $\overset{\text{Fu}}{\textbf{name}}$; $\overset{\text{M}}{\textbf{name}}$; $\overset{\text{sO}}{\textbf{name}}$; $\overset{\text{gF}}{\textbf{name}}$; $\overset{\text{var}}{\textbf{*name*}}$; $\overset{\text{co}}{\textbf{name}}$

▷ Symbol defined in Common Lisp; esp. function, macro, special operator, generic function, variable, constant.

them ▷ Placeholder for actual code.

`me` ▷ Literal text.

[*foo*$_{\texttt{bar}}$] ▷ Either one *foo* or nothing; defaults to `bar`.

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

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

foos ▷ English plural denotes a list argument.

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

$\left\{\begin{array}{|l} foo \\ bar \\ baz \end{array}\right.$ ▷ 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 $\overset{\text{sO}}{\textbf{progn}}$; see p. 20.

$\underline{foo}$; $\underline{bar}_2$; $\underline{baz}_n$ ▷ Primary, secondary, and *n*th return value.

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

Common Lisp Quick Reference

REALPART 4
REDUCE 14
REINITIALIZE-INSTANCE 25
REM 4
REMF 17
REMHASH 15
REMOVE 14
REMOVE-DUPLICATES 14
REMOVE-IF 14
REMOVE-IF-NOT 14
REMOVE-METHOD 26
REMPROP 17
RENAME-FILE 40
RENAME-PACKAGE 43
REPEAT 24
REPLACE 14
REQUIRE 44
REST 9
RESTART 41
RESTART-BIND 29
RESTART-CASE 29
RESTART-NAME 29
RETURN 21, 22
RETURN-FROM 21
REVAPPEND 10
REVERSE 13
ROOM 48
ROTATEF 17
ROUND 4
ROW-MAJOR-AREF 11
RPLACA 9
RPLACD 9

SAFETY 48
SATISFIES 42
SBIT 11
SCALE-FLOAT 6
SCHAR 8
SEARCH 13
SECOND 9
SEQUENCE 41
SERIOUS-CONDITION 30
SET 17
SET-DIFFERENCE 11
SET-DISPATCH-MACRO-CHARACTER 32
SET-EXCLUSIVE-OR 11
SET-MACRO-CHARACTER 32
SET-PPRINT-DISPATCH 35
SET-SYNTAX-FROM-CHAR 32
SETF 16, 44
SETQ 17
SEVENTH 9
SHADOW 44
SHADOWING-IMPORT 43
SHARED-INITIALIZE 25
SHIFTF 17
SHORT-FLOAT 41
SHORT-FLOAT-EPSILON 6
SHORT-FLOAT-NEGATIVE-EPSILON 6
SHORT-SITE-NAME 48
SIGNAL 28
SIGNED-BYTE 41
SIGNUM 4
SIMPLE-ARRAY 41
SIMPLE-BASE-STRING 41
SIMPLE-BIT-VECTOR 41
SIMPLE-BIT-VECTOR-P 11
SIMPLE-CONDITION 30
SIMPLE-CONDITION-FORMAT-ARGUMENTS 30
SIMPLE-CONDITION-FORMAT-CONTROL 30
SIMPLE-ERROR 30
SIMPLE-STRING 41
SIMPLE-STRING-P 7
SIMPLE-TYPE-ERROR 30
SIMPLE-VECTOR 41
SIMPLE-VECTOR-P 11
SIMPLE-WARNING 30
SIN 3
SINGLE-FLOAT 41
SINGLE-FLOAT-EPSILON 6
SINGLE-FLOAT-NEGATIVE-EPSILON 6
SINH 4
SIXTH 9
SLEEP 21
SLOT-BOUNDP 24
SLOT-EXISTS-P 24
SLOT-MAKUNBOUND 25
SLOT-MISSING 25
SLOT-UNBOUND 25
SLOT-VALUE 25
SOFTWARE-TYPE 48
SOFTWARE-VERSION 48
SOME 12
SORT 13
SPACE 48
SPECIAL 48
SPECIAL-OPERATOR-P 45
SPEED 48
SQRT 3
STABLE-SORT 13
STANDARD 27
STANDARD-CHAR 41
STANDARD-CHAR-P 6
STANDARD-CLASS 41
STANDARD-GENERIC-FUNCTION 41
STANDARD-METHOD 41
STANDARD-OBJECT 41
STEP 47
STORAGE-CONDITION 30
STORE-VALUE 29
STREAM 41
STREAM-ELEMENT-TYPE 42
STREAM-ERROR 30
STREAM-ERROR-STREAM 30
STREAM-EXTERNAL-FORMAT 39
STREAMP 31
STRING 8, 41
STRING-CAPITALIZE 8
STRING-DOWNCASE 8
STRING-EQUAL 7
STRING-GREATERP 8
STRING-LEFT-TRIM 8
STRING-LESSP 8
STRING-NOT-EQUAL 8
STRING-NOT-GREATERP 8
STRING-NOT-LESSP 8
STRING-RIGHT-TRIM 8
STRING-STREAM 41
STRING-TRIM 8
STRING-UPCASE 8
STRING/= 8
STRING< 8
STRING<= 8
STRING= 7
STRING> 8
STRING>= 8
STRINGP 7
STRUCTURE 44
STRUCTURE-CLASS 41
STRUCTURE-OBJECT 41
STYLE-WARNING 30
SUBLIS 10
SUBSEQ 13
SUBSETP 9
SUBST 10
SUBST-IF 10
SUBST-IF-NOT 10
SUBSTITUTE 14
SUBSTITUTE-IF 14
SUBSTITUTE-IF-NOT 14
SUBTYPEP 42
SUM 24
SUMMING 24
SVREF 12
SXHASH 15
SYMBOL 22, 41, 44
SYMBOL-FUNCTION 44
SYMBOL-MACROLET 19
SYMBOL-NAME 44
SYMBOL-PACKAGE 44
SYMBOL-PLIST 44
SYMBOL-VALUE 44
SYMBOLP 42
SYMBOLS 22
SYNONYM-STREAM 41
SYNONYM-STREAM-SYMBOL 38

T 2, 30, 41, 45
TAGBODY 21
TAILP 8
TAN 3
TANH 4
TENTH 9
TERPRI 33
THE 22, 42
THEN 22
THEREIS 24
THIRD 9
THROW 21
TIME 47
TO 22
TRACE 47
TRANSLATE-LOGICAL-PATHNAME 40
TRANSLATE-PATHNAME 40
TREE-EQUAL 10
TRUENAME 40
TRUNCATE 4
TWO-WAY-STREAM 41
TWO-WAY-STREAM-INPUT-STREAM 38
TWO-WAY-STREAM-OUTPUT-STREAM 38
TYPE 44, 47
TYPE-ERROR 30
TYPE-ERROR-DATUM 30
TYPE-ERROR-EXPECTED-TYPE 30
TYPE-OF 42
TYPECASE 42
TYPEP 40

UNBOUND-SLOT 30
UNBOUND-SLOT-INSTANCE 30
UNBOUND-VARIABLE 30
UNDEFINED-FUNCTION 30
UNEXPORT 44
UNINTERN 43
UNION 11
UNLESS 20, 22
UNREAD-CHAR 31
UNSIGNED-BYTE 41
UNTIL 24
UNTRACE 47
UNUSE-PACKAGE 43
UNWIND-PROTECT 21
UPDATE-INSTANCE-FOR-DIFFERENT-CLASS 25
UPDATE-INSTANCE-FOR-REDEFINED-CLASS 25
UPFROM 22
UPGRADED-ARRAY-ELEMENT-TYPE 42
UPGRADED-COMPLEX-PART-TYPE 6
UPPER-CASE-P 7
UPTO 22
USE-PACKAGE 43
USE-VALUE 29
USER-HOMEDIR-PATHNAME 40
USING 22

V 37
VALUES 18, 42
VALUES-LIST 18
VARIABLE 44
VECTOR 12, 41
VECTOR-POP 12
VECTOR-PUSH 12
VECTOR-PUSH-EXTEND 12
VECTORP 11

WARN 28
WARNING 30
WHEN 20, 22
WHILE 24
WILD-PATHNAME-P 31
WITH 22
WITH-ACCESSORS 25
WITH-COMPILATION-UNIT 46
WITH-CONDITION-RESTARTS 29
WITH-HASH-TABLE-ITERATOR 15
WITH-INPUT-FROM-STRING 39
WITH-OPEN-FILE 40
WITH-OPEN-STREAM 38
WITH-OUTPUT-TO-STRING 39
WITH-PACKAGE-ITERATOR 44
WITH-SIMPLE-RESTART 29
WITH-SLOTS 25
WITH-STANDARD-IO-SYNTAX 31
WRITE 34
WRITE-BYTE 34
WRITE-CHAR 34
WRITE-LINE 34
WRITE-SEQUENCE 34
WRITE-STRING 34
WRITE-TO-STRING 34

Y-OR-N-P 31
YES-OR-NO-P 31

ZEROP 3

HANDLER-CASE 29
HASH-KEY 22
HASH-KEYS 22
HASH-TABLE 41
HASH-TABLE-COUNT 15
HASH-TABLE-P 15
HASH-TABLE-REHASH-SIZE 15
HASH-TABLE-REHASH-THRESHOLD 15
HASH-TABLE-SIZE 15
HASH-TABLE-TEST 15
HASH-VALUE 22
HASH-VALUES 22
HOST-NAMESTRING 39

IDENTITY 18
IF 20, 22
IGNORABLE 47
IGNORE 47
IGNORE-ERRORS 28
IMAGPART 4
IMPORT 43
IN 22
IN-PACKAGE 43
INCF 3
INITIALIZE-INSTANCE 25
INITIALLY 24
INLINE 47
INPUT-STREAM-P 31
INSPECT 47
INTEGER 41
INTEGER-DECODE-FLOAT 6
INTEGER-LENGTH 5
INTEGERP 3
INTERACTIVE-STREAM-P 31
INTERN 43
INTERNAL-TIME-UNITS-PER-SECOND 48
INTERSECTION 11
INTO 24
INVALID-METHOD-ERROR 27
INVOKE-DEBUGGER 28
INVOKE-RESTART 29
INVOKE-RESTART-INTERACTIVELY 29
ISQRT 3
IT 22, 24

KEYWORD 41, 43, 45
KEYWORDP 42

LABELS 17
LAMBDA 17
LAMBDA-LIST-KEYWORDS 20
LAMBDA-PARAMETERS-LIMIT 18
LAST 9
LCM 3
LDB 6
LDB-TEST 5
LDIFF 9
LEAST-NEGATIVE-DOUBLE-FLOAT 6
LEAST-NEGATIVE-LONG-FLOAT 6
LEAST-NEGATIVE-NORMALIZED-DOUBLE-FLOAT 6
LEAST-NEGATIVE-NORMALIZED-LONG-FLOAT 6
LEAST-NEGATIVE-NORMALIZED-SHORT-FLOAT 6
LEAST-NEGATIVE-NORMALIZED-SINGLE-FLOAT 6
LEAST-NEGATIVE-SHORT-FLOAT 6
LEAST-NEGATIVE-SINGLE-FLOAT 6
LEAST-POSITIVE-DOUBLE-FLOAT 6
LEAST-POSITIVE-LONG-FLOAT 6
LEAST-POSITIVE-NORMALIZED-DOUBLE-FLOAT 6
LEAST-POSITIVE-NORMALIZED-LONG-FLOAT 6
LEAST-POSITIVE-NORMALIZED-SHORT-FLOAT 6
LEAST-POSITIVE-NORMALIZED-SINGLE-FLOAT 6
LEAST-POSITIVE-SHORT-FLOAT 6
LEAST-POSITIVE-SINGLE-FLOAT 6
LENGTH 13
LET 20
LET* 20
LISP-IMPLEMENTATION-TYPE 48
LISP-IMPLEMENTATION-VERSION 48
LIST 9, 27, 41
LIST-ALL-PACKAGES 43
LIST-LENGTH 9
LIST* 9
LISTEN 38
LISTP 8
LOAD 45
LOAD-LOGICAL-PATHNAME-TRANSLATIONS 40
LOAD-TIME-VALUE 46
LOCALLY 46
LOG 3
LOGAND 5
LOGANDC1 5
LOGANDC2 5
LOGBITP 5
LOGCOUNT 5
LOGEQV 5
LOGICAL-PATHNAME 40, 41
LOGICAL-PATHNAME-TRANSLATIONS 40
LOGIOR 5
LOGNAND 5
LOGNOR 5
LOGNOT 5
LOGORC1 5
LOGORC2 5
LOGTEST 5
LOGXOR 5
LONG-FLOAT 41
LONG-FLOAT-EPSILON 6
LONG-FLOAT-NEGATIVE-EPSILON 6
LONG-SITE-NAME 48
LOOP 22
LOOP-FINISH 24
LOWER-CASE-P 7

MACHINE-INSTANCE 48
MACHINE-TYPE 48
MACHINE-VERSION 48
MACRO-FUNCTION 46
MACROEXPAND 46
MACROEXPAND-1 46
MACROLET 19
MAKE-ARRAY 11
MAKE-BROADCAST-STREAM 38
MAKE-CONCATENATED-STREAM 38
MAKE-CONDITION 28
MAKE-DISPATCH-MACRO-CHARACTER 32
MAKE-ECHO-STREAM 38
MAKE-HASH-TABLE 15
MAKE-INSTANCE 25
MAKE-INSTANCES-OBSOLETE 25
MAKE-LIST 9
MAKE-LOAD-FORM 46
MAKE-LOAD-FORM-SAVING-SLOTS 46
MAKE-METHOD 28
MAKE-PACKAGE 43
MAKE-PATHNAME 39
MAKE-RANDOM-STATE 4
MAKE-SEQUENCE 13
MAKE-STRING 8
MAKE-STRING-INPUT-STREAM 38
MAKE-STRING-OUTPUT-STREAM 38
MAKE-SYMBOL 44
MAKE-SYNONYM-STREAM 38
MAKE-TWO-WAY-STREAM 38
MAKUNBOUND 17
MAP 14
MAP-INTO 14
MAPC 10
MAPCAN 10
MAPCAR 10
MAPCON 10
MAPHASH 15
MAPL 10
MAPLIST 10
MASK-FIELD 6
MAX 4, 27
MAXIMIZE 24
MAXIMIZING 24
MEMBER 8, 42
MEMBER-IF 9
MEMBER-IF-NOT 9
MERGE 13
MERGE-PATHNAMES 39
METHOD 41
METHOD-COMBINATION 41, 44
METHOD-COMBINATION-ERROR 27
METHOD-QUALIFIERS 27
MIN 4, 27
MINIMIZE 24
MINIMIZING 24
MINUSP 3
MISMATCH 12
MOD 4, 42
MOST-NEGATIVE-DOUBLE-FLOAT 6
MOST-NEGATIVE-FIXNUM 6
MOST-NEGATIVE-LONG-FLOAT 6
MOST-NEGATIVE-SHORT-FLOAT 6
MOST-NEGATIVE-SINGLE-FLOAT 6
MOST-POSITIVE-DOUBLE-FLOAT 6
MOST-POSITIVE-FIXNUM 6
MOST-POSITIVE-LONG-FLOAT 6
MOST-POSITIVE-SHORT-FLOAT 6
MOST-POSITIVE-SINGLE-FLOAT 6
MUFFLE-WARNING 29
MULTIPLE-VALUE-BIND 21
MULTIPLE-VALUE-CALL 18
MULTIPLE-VALUE-LIST 18
MULTIPLE-VALUE-PROG1 20
MULTIPLE-VALUE-SETQ 17
MULTIPLE-VALUES-LIMIT 18

NAME-CHAR 7
NAMED 22
NAMESTRING 39
NBUTLAST 9
NCONC 9, 24, 27
NCONCING 24
NEVER 24
NEXT-METHOD-P 26
NIL 2, 45
NINTERSECTION 11
NINTH 9
NO-APPLICABLE-METHOD 26
NO-NEXT-METHOD 27
NOT 16, 42
NOTANY 12
NOTEVERY 12
NOTINLINE 47
NRECONC 10
NREVERSE 13
NSET-DIFFERENCE 11
NSET-EXCLUSIVE-OR 11
NSTRING-CAPITALIZE 8
NSTRING-DOWNCASE 8
NSTRING-UPCASE 8
NSUBLIS 10
NSUBST 10
NSUBST-IF 10
NSUBST-IF-NOT 10
NSUBSTITUTE 14
NSUBSTITUTE-IF 14
NSUBSTITUTE-IF-NOT 14
NTH 9
NTH-VALUE 18
NTHCDR 9
NULL 8, 41
NUMBER 41
NUMBERP 3
NUMERATOR 4
NUNION 11

ODDP 3
OF 22
OF-TYPE 22
ON 22
OPEN 38
OPEN-STREAM-P 31
OPTIMIZE 48
OR 20, 27, 42
OTHERWISE 20, 42
OUTPUT-STREAM-P 31

PACKAGE 41
PACKAGE-ERROR 30
PACKAGE-ERROR-PACKAGE 30
PACKAGE-NAME 43
PACKAGE-NICKNAMES 43
PACKAGE-SHADOWING-SYMBOLS 44
PACKAGE-USE-LIST 43
PACKAGE-USED-BY-LIST 43
PACKAGEP 42
PAIRLIS 10
PARSE-ERROR 30
PARSE-INTEGER 8
PARSE-NAMESTRING 39
PATHNAME 39, 41
PATHNAME-DEVICE 40
PATHNAME-DIRECTORY 40
PATHNAME-HOST 40
PATHNAME-MATCH-P 31
PATHNAME-NAME 40
PATHNAME-TYPE 40
PATHNAME-VERSION 40
PATHNAMEP 31
PEEK-CHAR 31
PHASE 4
PI 3
PLUSP 3
POP 9
POSITION 13
POSITION-IF 13
POSITION-IF-NOT 13
PPRINT 33
PPRINT-DISPATCH 35
PPRINT-EXIT-IF-LIST-EXHAUSTED 35
PPRINT-FILL 34
PPRINT-INDENT 34
PPRINT-LINEAR 34
PPRINT-LOGICAL-BLOCK 34
PPRINT-NEWLINE 35
PPRINT-POP 34
PPRINT-TAB 34
PPRINT-TABULAR 34
PRESENT-SYMBOL 22
PRESENT-SYMBOLS 22
PRIN1 33
PRIN1-TO-STRING 33
PRINC 33
PRINC-TO-STRING 33
PRINT 33
PRINT-NOT-READABLE 30
PRINT-NOT-READABLE-OBJECT 30
PRINT-OBJECT 33
PRINT-UNREADABLE-OBJECT 33
PROBE-FILE 40
PROCLAIM 47
PROG 21
PROG1 20
PROG2 20
PROG* 21
PROGN 20, 27
PROGRAM-ERROR 30
PROGV 21
PROVIDE 44
PSETF 16
PSETQ 17
PUSH 9
PUSHNEW 9

QUOTE 46

RANDOM 4
RANDOM-STATE 41
RANDOM-STATE-P 3
RASSOC 10
RASSOC-IF 10
RASSOC-IF-NOT 10
RATIO 41
RATIONAL 4, 41
RATIONALIZE 4
RATIONALP 3
READ 31
READ-BYTE 31
READ-CHAR 31
READ-CHAR-NO-HANG 31
READ-DELIMITED-LIST 31
READ-FROM-STRING 31
READ-LINE 32
READ-PRESERVING-WHITESPACE 31
READ-SEQUENCE 32
READER-ERROR 30
READTABLE 41
READTABLE-CASE 32
READTABLEP 31
REAL 41
REALP 3

Common Lisp Quick Reference

1 Numbers

1.1 Predicates

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

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

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

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

($\overset{\text{Fu}}{\textbf{minusp}}$ *a*)
($\overset{\text{Fu}}{\textbf{zerop}}$ *a*)
($\overset{\text{Fu}}{\textbf{plusp}}$ *a*)

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

($\overset{\text{Fu}}{\textbf{evenp}}$ *integer*)
($\overset{\text{Fu}}{\textbf{oddp}}$ *integer*)

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

($\overset{\text{Fu}}{\textbf{numberp}}$ *foo*)
($\overset{\text{Fu}}{\textbf{realp}}$ *foo*)
($\overset{\text{Fu}}{\textbf{rationalp}}$ *foo*)
($\overset{\text{Fu}}{\textbf{floatp}}$ *foo*)
($\overset{\text{Fu}}{\textbf{integerp}}$ *foo*)
($\overset{\text{Fu}}{\textbf{complexp}}$ *foo*)
($\overset{\text{Fu}}{\textbf{random-state-p}}$ *foo*)

▷ T if *foo* is of indicated type.

1.2 Numeric Functions

($\overset{\text{Fu}}{+}$ $a_{\boxed{0}}{}^*$)
($\overset{\text{Fu}}{*}$ $a_{\boxed{1}}{}^*$)

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

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

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

($\overset{\text{Fu}}{\textbf{1+}}$ *a*)
($\overset{\text{Fu}}{\textbf{1-}}$ *a*)

▷ Return $a + 1$ or $a - 1$, respectively.

($\left\{\begin{array}{l}\overset{\text{M}}{\textbf{incf}}\\ \overset{\text{M}}{\textbf{decf}}\end{array}\right\}$ $\widetilde{place}$ [$delta_{\boxed{1}}$])

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

($\overset{\text{Fu}}{\textbf{exp}}$ *p*)
($\overset{\text{Fu}}{\textbf{expt}}$ *b* *p*)

▷ Return e^p or b^p, respectively.

($\overset{\text{Fu}}{\textbf{log}}$ *a* [*b*]) ▷ Return $\log_b a$ or, without *b*, $\ln a$.

($\overset{\text{Fu}}{\textbf{sqrt}}$ *n*)
($\overset{\text{Fu}}{\textbf{isqrt}}$ *n*)

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

($\overset{\text{Fu}}{\textbf{lcm}}$ $integer^*_{\boxed{1}}$)
($\overset{\text{Fu}}{\textbf{gcd}}$ *integer**)

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

$\overset{\text{co}}{\textbf{pi}}$ ▷ **long-float** approximation of π, Ludolph's number.

($\overset{\text{Fu}}{\textbf{sin}}$ *a*)
($\overset{\text{Fu}}{\textbf{cos}}$ *a*)
($\overset{\text{Fu}}{\textbf{tan}}$ *a*)

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

($\overset{\text{Fu}}{\textbf{asin}}$ *a*)
($\overset{\text{Fu}}{\textbf{acos}}$ *a*)

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

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

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

(**asinh** *a*)
(**acosh** *a*)
(**atanh** *a*)
▷ $\underline{\operatorname{asinh} a}$, $\underline{\operatorname{acosh} a}$, or $\underline{\operatorname{atanh} a}$, respectively.

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

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

(**max** *num*$^+$)
(**min** *num*$^+$)
▷ Greatest or least, respectively, of *nums*.

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

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

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

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

random-state ▷ Current random state.

(**float-sign** *num-a* [*num-b*$_{\boxed{1}}$])
▷ *num-b* with the sign of *num-a*.

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

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

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

(**complex** *real* [*imag*$_{\boxed{0}}$]) ▷ Make a complex number.

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

(**abs** *n*) ▷ Return $\underline{|n|}$.

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

(**float** *real* [*prototype*$_{\boxed{\textbf{single-float}}}$])
▷ Convert *real* into float with type of *prototype*.

` 32
| | 33
1+ 3
1– 3

ABORT 29
ABOVE 22
ABS 4
ACONS 10
ACOS 3
ACOSH 4
ACROSS 22
ADD-METHOD 26
ADJOIN 9
ADJUST-ARRAY 11
ADJUSTABLE-ARRAY-P 11
ALLOCATE-INSTANCE 25
ALPHA-CHAR-P 6
ALPHANUMERICP 6
ALWAYS 24
AND 20, 22, 27, 42
APPEND 9, 24, 27
APPENDING 24
APPLY 18
APROPOS 46
APROPOS-LIST 46
AREF 11
ARITHMETIC-ERROR 30
ARITHMETIC-ERROR-OPERANDS 30
ARITHMETIC-ERROR-OPERATION 30
ARRAY 41
ARRAY-DIMENSION 11
ARRAY-DIMENSION-LIMIT 12
ARRAY-DIMENSIONS 11
ARRAY-DISPLACEMENT 11
ARRAY-ELEMENT-TYPE 42
ARRAY-HAS-FILL-POINTER-P 11
ARRAY-IN-BOUNDS-P 11
ARRAY-RANK 11
ARRAY-RANK-LIMIT 12
ARRAY-ROW-MAJOR-INDEX 11
ARRAY-TOTAL-SIZE 11
ARRAY-TOTAL-SIZE-LIMIT 12
ARRAYP 11
AS 22
ASH 5
ASIN 3
ASINH 4
ASSERT 28
ASSOC 10
ASSOC-IF 10
ASSOC-IF-NOT 10
ATAN 3
ATANH 4
ATOM 8, 41

BASE-CHAR 41
BASE-STRING 41
BEING 22
BELOW 22
BIGNUM 41
BIT 11, 41
BIT-AND 12
BIT-ANDC1 12
BIT-ANDC2 12
BIT-EQV 12
BIT-IOR 12
BIT-NAND 12
BIT-NOR 12
BIT-NOT 11
BIT-ORC1 12
BIT-ORC2 12
BIT-VECTOR 41
BIT-VECTOR-P 11
BIT-XOR 12
BLOCK 21
BOOLE 5
BOOLE-1 5
BOOLE-2 5
BOOLE-AND 5
BOOLE-ANDC1 5
BOOLE-ANDC2 5
BOOLE-C1 5
BOOLE-C2 5
BOOLE-CLR 5
BOOLE-EQV 5
BOOLE-IOR 5
BOOLE-NAND 5
BOOLE-NOR 5
BOOLE-ORC1 5
BOOLE-ORC2 5
BOOLE-SET 5
BOOLE-XOR 5
BOOLEAN 41
BOTH-CASE-P 7
BOUNDP 16
BREAK 47
BROADCAST-STREAM 41
BROADCAST-STREAM-STREAMS 38
BUILT-IN-CLASS 41
BUTLAST 9
BY 22
BYTE 6
BYTE-POSITION 6
BYTE-SIZE 6

CAAR 9
CADR 9
CALL-ARGUMENTS-LIMIT 18
CALL-METHOD 28
CALL-NEXT-METHOD 26
CAR 9
CASE 20
CATCH 21
CCASE 20
CDAR 9
CDDR 9
CDR 9
CEILING 4
CELL-ERROR 30
CELL-ERROR-NAME 30
CERROR 28
CHANGE-CLASS 25
CHAR 8
CHAR-CODE 7
CHAR-CODE-LIMIT 7
CHAR-DOWNCASE 7
CHAR-EQUAL 7
CHAR-GREATERP 7
CHAR-INT 7
CHAR-LESSP 7
CHAR-NAME 7
CHAR-NOT-EQUAL 7
CHAR-NOT-GREATERP 7
CHAR-NOT-LESSP 7
CHAR-UPCASE 7
CHAR/= 7
CHAR< 7
CHAR<= 7
CHAR= 7
CHAR> 7
CHAR>= 7
CHARACTER 7, 41
CHARACTERP 6
CHECK-TYPE 42
CIS 4
CL 45
CL-USER 45
CLASS 41
CLASS-NAME 25
CLASS-OF 25
CLEAR-INPUT 38
CLEAR-OUTPUT 38
CLOSE 38
CLRHASH 15
CODE-CHAR 7
COERCE 42
COLLECT 24
COLLECTING 24
COMMON-LISP 45
COMMON-LISP-USER 45
COMPILATION-SPEED 48
COMPILE 45
COMPILE-FILE 45
COMPILE-FILE-PATHNAME 45
COMPILED-FUNCTION 41
COMPILED-FUNCTION-P 45
COMPILER-MACRO 44
COMPILER-MACRO-FUNCTION 46
COMPLEMENT 18
COMPLEX 4, 41
COMPLEXP 3
COMPUTE-APPLICABLE-METHODS 26
COMPUTE-RESTARTS 29
CONCATENATE 13
CONCATENATED-STREAM 41
CONCATENATED-STREAM-STREAMS 38
COND 20
CONDITION 30
CONJUGATE 4
CONS 9, 41
CONSP 8
CONSTANTLY 18
CONSTANTP 16
CONTINUE 29
CONTROL-ERROR 30
COPY-ALIST 10
COPY-LIST 10
COPY-PPRINT-DISPATCH 35
COPY-READTABLE 32
COPY-SEQ 14
COPY-STRUCTURE 16
COPY-SYMBOL 44
COPY-TREE 10
COS 3
COSH 4
COUNT 13, 24
COUNT-IF 13
COUNT-IF-NOT 13
COUNTING 24
CTYPECASE 42

DEBUG 48
DECF 3
DECLAIM 47
DECLARATION 47
DECLARE 47
DECODE-FLOAT 6
DECODE-UNIVERSAL-TIME 48
DEFCLASS 24
DEFCONSTANT 16
DEFGENERIC 26
DEFINE-COMPILER-MACRO 19
DEFINE-CONDITION 28
DEFINE-METHOD-COMBINATION 27
DEFINE-MODIFY-MACRO 19
DEFINE-SETF-EXPANDER 19
DEFINE-SYMBOL-MACRO 19
DEFMACRO 19
DEFMETHOD 26
DEFPACKAGE 43
DEFPARAMETER 16
DEFSETF 19
DEFSTRUCT 15
DEFTYPE 42
DEFUN 17
DEFVAR 16
DELETE 14
DELETE-DUPLICATES 14
DELETE-FILE 40
DELETE-IF 14
DELETE-IF-NOT 14
DELETE-PACKAGE 43
DENOMINATOR 4
DEPOSIT-FIELD 6
DESCRIBE 47
DESCRIBE-OBJECT 47
DESTRUCTURING-BIND 21
DIGIT-CHAR 7
DIGIT-CHAR-P 7
DIRECTORY 40
DIRECTORY-NAMESTRING 39
DISASSEMBLE 47
DIVISION-BY-ZERO 30
DO 21, 22
DO-ALL-SYMBOLS 44
DO-EXTERNAL-SYMBOLS 44
DO-SYMBOLS 44
DO* 21
DOCUMENTATION 44
DOING 22
DOLIST 21
DOTIMES 21
DOUBLE-FLOAT 41
DOUBLE-FLOAT-EPSILON 6
DOUBLE-FLOAT-NEGATIVE-EPSILON 6
DOWNFROM 22
DOWNTO 22
DPB 6
DRIBBLE 46
DYNAMIC-EXTENT 47

EACH 22
ECASE 20
ECHO-STREAM 41
ECHO-STREAM-INPUT-STREAM 38
ECHO-STREAM-OUTPUT-STREAM 38
ED 46
EIGHTH 9
ELSE 22
ELT 13
ENCODE-UNIVERSAL-TIME 48
END 22
END-OF-FILE 30
ENDP 8
ENOUGH-NAMESTRING 39
ENSURE-DIRECTORIES-EXIST 40
ENSURE-GENERIC-FUNCTION 26
EQ 16
EQL 16, 42
EQUAL 16
EQUALP 16
ERROR 28, 30
ETYPECASE 42
EVAL 46
EVAL-WHEN 46
EVENP 3
EVERY 12
EXP 3
EXPORT 44
EXPT 3
EXTENDED-CHAR 41
EXTERNAL-SYMBOL 22
EXTERNAL-SYMBOLS 22

FBOUNDP 16
FCEILING 4
FDEFINITION 18
FFLOOR 4
FIFTH 9
FILE-AUTHOR 40
FILE-ERROR 30
FILE-ERROR-PATHNAME 30
FILE-LENGTH 40
FILE-NAMESTRING 39
FILE-POSITION 40
FILE-STREAM 41
FILE-STRING-LENGTH 40
FILE-WRITE-DATE 40
FILL 13
FILL-POINTER 12
FINALLY 24
FIND 13
FIND-ALL-SYMBOLS 43
FIND-CLASS 25
FIND-IF 13
FIND-IF-NOT 13
FIND-METHOD 26
FIND-PACKAGE 43
FIND-RESTART 29
FIND-SYMBOL 43
FINISH-OUTPUT 38
FIRST 9
FIXNUM 41
FLET 17
FLOAT 4, 41
FLOAT-DIGITS 6
FLOAT-PRECISION 6
FLOAT-RADIX 6
FLOAT-SIGN 4
FLOATING-POINT-INEXACT 30
FLOATING-POINT-INVALID-OPERATION 30
FLOATING-POINT-OVERFLOW 30
FLOATING-POINT-UNDERFLOW 30
FLOATP 3
FLOOR 4
FMAKUNBOUND 18
FOR 22
FORCE-OUTPUT 38
FORMAT 36
FORMATTER 35
FOURTH 9
FRESH-LINE 33
FROM 22
FROUND 4
FTRUNCATE 4
FTYPE 47
FUNCALL 18
FUNCTION 18, 41, 44
FUNCTION-KEYWORDS 27
FUNCTION-LAMBDA-EXPRESSION 18
FUNCTIONP 16

GCD 3
GENERIC-FUNCTION 41
GENSYM 44
GENTEMP 44
GET 17
GET-DECODED-TIME 48
GET-DISPATCH-MACRO-CHARACTER 32
GET-INTERNAL-REAL-TIME 48
GET-INTERNAL-RUN-TIME 48
GET-MACRO-CHARACTER 32
GET-OUTPUT-STREAM-STRING 38
GET-PROPERTIES 17
GET-SETF-EXPANSION 19
GET-UNIVERSAL-TIME 48
GETF 17
GETHASH 15
GO 21
GRAPHIC-CHAR-P 6

HANDLER-BIND 29

(**optimize** { **compilation-speed** | (**compilation-speed** $n_{\boxed{3}}$) ; **debug** | (**debug** $n_{\boxed{3}}$) ; **safety** | (**safety** $n_{\boxed{3}}$) ; **space** | (**space** $n_{\boxed{3}}$) ; **speed** | (**speed** $n_{\boxed{3}}$) })

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

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

16 External Environment

(**get-internal-real-time**)[Fu]
(**get-internal-run-time**)[Fu]

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

internal-time-units-per-second[co]

▷ Number of clock ticks per second.

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

▷ Seconds from 1900-01-01, 00:00.

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

▷ Return second, minute$_2$, hour$_3$, date$_4$, month$_5$, year$_6$, day$_7$, daylight-p$_8$, and zone$_9$.

(**room**[Fu] [{NIL | **:default** | T}])

▷ Print information about internal storage management.

(**short-site-name**[Fu])
(**long-site-name**[Fu])

▷ String representing physical location of computer.

({ **lisp-implementation**[Fu] | **software**[Fu] | **machine**[Fu] }-{ **type** | **version** })

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

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

Index

" 32
' 32
(32
() 45
* 41
* 3, 46
** 46
*** 46
BREAK-ON-SIGNALS 30
COMPILE-FILE-PATHNAME 45
COMPILE-FILE-TRUENAME 45
COMPILE-PRINT 45
COMPILE-VERBOSE 45
DEBUG-IO 39
DEBUGGER-HOOK 30
DEFAULT-PATHNAME-DEFAULTS 39
ERROR-OUTPUT 39
FEATURES 33
GENSYM-COUNTER 44
LOAD-PATHNAME 45
LOAD-PRINT 45
LOAD-TRUENAME 45
LOAD-VERBOSE 45
MACROEXPAND-HOOK 47
MODULES 44
PACKAGE 43
PRINT-ARRAY 35
PRINT-BASE 35
PRINT-CASE 35
PRINT-CIRCLE 35
PRINT-ESCAPE 35
PRINT-GENSYM 35
PRINT-LENGTH 35
PRINT-LEVEL 35
PRINT-LINES 35
PRINT-MISER-WIDTH 35
PRINT-PPRINT-DISPATCH 35
PRINT-PRETTY 35
PRINT-RADIX 35
PRINT-READABLY 35
PRINT-RIGHT-MARGIN 35
QUERY-IO 39
RANDOM-STATE 4
READ-BASE 32
READ-DEFAULT-FLOAT-FORMAT 32
READ-EVAL 33
READ-SUPPRESS 32
READTABLE 32
STANDARD-INPUT 39
STANDARD-OUTPUT 39
TERMINAL-IO 39
TRACE-OUTPUT 47
+ 3, 27, 46
++ 46
+++ 46
, 32
,. 32
,@ 32
– 3, 46
/ 3, 46
// 46
/// 46
/= 3
: 43
:: 43
:ALLOW-OTHER-KEYS 20
; 32
< 3
<= 3
= 3, 22
> 3
>= 3
\ 33
37
#\ 33
#' 33
#(33
#* 33
#+ 33
#– 33
#. 33
#: 33
#< 33
#= 33
#A 33
#B 33
#C(33
#O 33
#P 33
#R 33
#S(33
#X 33
33
#| |# 32
&ALLOW-OTHER-KEYS 20
&AUX 20
&BODY 20
&ENVIRONMENT 20
&KEY 20
&OPTIONAL 20
&REST 20
&WHOLE 20
~(~) 36
~* 37
~/ / 37
~< ~:> 37
~< ~> 37
~? 37
~A 36
~B 36
~C 36
~D 36
~E 36
~F 36
~G 36
~I 37
~O 36
~P 36
~R 36
~S 36
~T 37
~W 37
~X 36
~[~] 37
~$ 36
~% 36
~& 36
~^ 37
~_ 36
~| 37
~{ ~} 37
~~ 37

1.3 Logic Functions

Negative integers are used in two's complement representation.

(**boole**[Fu] *operation int-a int-b*)

▷ Return value of bitwise logical *operation*. *operation*s are

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

(**lognot**[Fu] *integer*) ▷ $\neg$*integer*.

(**logeqv**[Fu] *integer**)
(**logand**[Fu] *integer**)

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

(**logandc1**[Fu] *int-a int-b*) ▷ $\neg$*int-a* $\wedge$ *int-b*.

(**logandc2**[Fu] *int-a int-b*) ▷ *int-a* $\wedge$ $\neg$*int-b*.

(**lognand**[Fu] *int-a int-b*) ▷ $\neg$(*int-a* $\wedge$ *int-b*).

(**logxor**[Fu] *integer**)
(**logior**[Fu] *integer**)

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

(**logorc1**[Fu] *int-a int-b*) ▷ $\neg$*int-a* $\vee$ *int-b*.

(**logorc2**[Fu] *int-a int-b*) ▷ *int-a* $\vee$ $\neg$*int-b*.

(**lognor**[Fu] *int-a int-b*) ▷ $\neg$(*int-a* $\vee$ *int-b*).

(**logbitp**[Fu] *i integer*)

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

(**logtest**[Fu] *int-a int-b*)

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

(**logcount**[Fu] *int*)

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

1.4 Integer Functions

(**integer-length**[Fu] *integer*)

▷ Number of bits necessary to represent *integer*.

(**ldb-test**[Fu] *byte-spec integer*)

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

(**ash**[Fu] *integer count*)

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

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

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

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

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

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

1.5 Implementation-Dependent

{**short-float** / **single-float** / **double-float** / **long-float**}-{**epsilon** / **negative-epsilon**}
▷ Smallest possible number making a difference when added or subtracted, respectively.

{**least-negative** / **least-negative-normalized** / **least-positive** / **least-positive-normalized**}-{**short-float** / **single-float** / **double-float** / **long-float**}
▷ Available numbers closest to −0 or +0, respectively.

{**most-negative** / **most-positive**}-{**short-float** / **single-float** / **double-float** / **long-float** / **fixnum**}
▷ Available numbers closest to −∞ or +∞, respectively.

(**decode-float** *n*)
(**integer-decode-float** *n*)
▷ Return significand, exponent, and sign of **float** *n*.

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

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

(**upgraded-complex-part-type** *foo* [*environment*NIL])
▷ Type of most specialized **complex** number able to hold parts of type *foo*.

2 Characters

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

(**graphic-char-p** *character*)
(**alpha-char-p** *character*)
(**alphanumericp** *character*)
▷ T if *character* is visible, alphabetic, or alphanumeric, respectively.

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

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

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

trace-output
▷ Stream **trace** and **time** print their output on.

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

(**break** [*control arg**])
▷ Jump directly into debugger; return NIL. See p. 35, **format**, for *control* and *args*.

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

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

(**describe** *foo* [*stream****standard-output***])
▷ Send information about *foo* to *stream*.

(**describe-object** *foo* [*stream*])
▷ Send information about *foo* to *stream*. Not to be called by user.

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

15.4 Declarations

(**proclaim** *decl*)
(**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 *variables* and/or *functions* to end when control leaves enclosing block.

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

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

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

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

(**locally** (**declare** *decl**)* *form*^{P_*})
▷ Evaluate *forms* in a lexical environment with declarations *decl* in effect. Return values of *forms*.

(**with-compilation-unit** ([:override *bool*NIL]) *form*^{P_*})
▷ Return values of *forms*. Warnings deferred by the compiler until end of compilation are deferred until the end of evaluation of *forms*.

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

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

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

(**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*.

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

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

15.3 REPL and Debugging

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

\- ▷ Form currently being evaluated by the REPL.

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

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

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

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

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

(**upper-case-p** *character*)
(**lower-case-p** *character*)
(**both-case-p** *character*)
▷ Return T if *character* is uppercase, lowercase, or able to be in another case, respectively.

(**digit-char-p** *character* [*radix*10])
▷ Return its weight if *character* is a digit, or NIL otherwise.

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

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

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

(**char-greaterp** *character*$^+$)
(**char-not-lessp** *character*$^+$)
(**char-lessp** *character*$^+$)
(**char-not-greaterp** *character*$^+$)
▷ Return T if *character*s are monotonically decreasing, monotonically non-increasing, monotonically increasing, or monotonically non-decreasing, respectively, ignoring case.

(**char-upcase** *character*)
(**char-downcase** *character*)
▷ Return corresponding uppercase/lowercase character, respectively.

(**digit-char** *i* [*radix*10]) ▷ Character representing digit *i*.

(**char-name** *character*)
▷ Name of *character* if there is one, or NIL.

(**name-char** *name*)
▷ Character with *name* if there is one, or NIL.

(**char-int** *character*)
(**char-code** *character*) ▷ Code of *character*.

(**code-char** *code*) ▷ Character with *code*.

char-code-limit ▷ Upper bound of (**char-code** *char*); ≥ 96.

(**character** *c*) ▷ Return #*c*.

3 Strings

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

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

({ **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.

(**string/=** | **string>** | **string>=** | **string<** | **string<=** *foo bar* {**:start1** *start-foo*$_{\boxed{0}}$ | **:start2** *start-bar*$_{\boxed{0}}$ | **:end1** *end-foo*$_{\boxed{NIL}}$ | **:end2** *end-bar*$_{\boxed{NIL}}$})

▷ If *foo* is lexicographically not equal, greater, not less, less, or not greater, respectively, then return character number from beginning of *foo* where they begin to differ. Otherwise return `NIL`.

(**string-not-equal** | **string-greaterp** | **string-not-lessp** | **string-lessp** | **string-not-greaterp** *foo bar* {**:start1** *start-foo*$_{\boxed{0}}$ | **:start2** *start-bar*$_{\boxed{0}}$ | **:end1** *end-foo*$_{\boxed{NIL}}$ | **:end2** *end-bar*$_{\boxed{NIL}}$})

▷ If *foo* is lexicographically not equal, greater, not less, less, or not greater, respectively, ignoring case, then return character number from beginning of *foo* where they begin to differ. Otherwise return `NIL`.

(**string** *x*)

▷ Convert *x* (**symbol**, **string**, or **character**) into a string.

(**make-string** *size* {**:initial-element** *char* | **:element-type** *type*$_{\boxed{character}}$})

▷ Return string of length *size*.

({**string** | **nstring**}-{**capitalize** | **upcase** | **downcase**} *string* {**:start** *start*$_{\boxed{0}}$ | **:end** *end*$_{\boxed{NIL}}$})

▷ Return *string* (not modified or modified, respectively) with first letter of every word turned into uppercase, letters all uppercase, or letters all lowercase, respectively.

({**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.

(**char** *string i*)
(**schar** *string i*)

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

(**parse-integer** *string* {**:start** *start*$_{\boxed{0}}$ | **:end** *end*$_{\boxed{NIL}}$ | **:radix** *int*$_{\boxed{10}}$ | **:junk-allowed** *bool*$_{\boxed{NIL}}$})

▷ Return integer parsed from *string* and $\underset{2}{\text{index}}$ of parse end.

4 Conses

4.1 Predicates

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

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

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

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

(**member** *foo list* {{**:test** *function*$_{\boxed{\#'eql}}$ | **:test-not** *function*} | **:key** *function*})

▷ Return tail of *list* starting with its first element matching *foo*. Return `NIL` if there is no such element.

t

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

nil|**()**

▷ Falsity; the empty list; the empty type, subtype of every type; ***standard-input***; ***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

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

(**compiled-function-p** *foo*)

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

15.2 Compilation

(**compile** {`NIL` *definition* | {*name* | (**setf** *name*)} [*definition*]})

▷ Return compiled function or replace *name*'s function definition with the compiled function. Return $\underset{2}{\texttt{T}}$ in case of warnings or errors, and $\underset{3}{\texttt{T}}$ in case of warnings or errors excluding style warnings.

(**compile-file** *file* {**:output-file** *out-path* | **:verbose** *bool*$_{\boxed{*compile-verbose*}}$ | **:print** *bool*$_{\boxed{*compile-print*}}$ | **:external-format** *file-format*$_{\boxed{:default}}$})

▷ Write compiled contents of *file* to *out-path*. Return true output path or `NIL`, $\underset{2}{\texttt{T}}$ in case of warnings or errors, $\underset{3}{\texttt{T}}$ in case of warnings or errors excluding style warnings.

(**compile-file-pathname** *file* [**:output-file** *path*] [*other-keyargs*])

▷ Pathname **compile-file** writes to if invoked with the same arguments.

(**load** *path* {**:verbose** *bool*$_{\boxed{*load-verbose*}}$ | **:print** *bool*$_{\boxed{*load-print*}}$ | **:if-does-not-exist** *bool*$_{\boxed{T}}$ | **:external-format** *file-format*$_{\boxed{:default}}$})

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

{***compile-file** | ***load**}-{**pathname***$_{\boxed{NIL}}$ | **truename***$_{\boxed{NIL}}$}

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

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

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

(**shadow** *symbols* [*package*$_{\text{*package*}}$])
▷ Add *symbols* to shadowing list of *package* making equally named inherited symbols shadowed. Return T.

(**package-shadowing-symbols** *package*)
▷ List of shadowing symbols of *package*.

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

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

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

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

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

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

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*).

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

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

(**gentemp** [*prefix*$_{\text{T}}$ [*package*$_{\text{*package*}}$]])
▷ Intern fresh symbol in package. Deprecated.

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

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

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

({**member-if** | **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.

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

4.2 Lists

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

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

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

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

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

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

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

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

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

(**nth** *n list*)
▷ Return zero-indexed *n*th element of *list*. **setf**able.

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

(**last** *list* [*num*$_{\text{1}}$]) ▷ Return list of last *num* conses of *list*.

({**butlast** *list* | **nbutlast** $\widetilde{list}$} [*num*$_{\text{1}}$])
▷ Return *list* excluding last *num* conses.

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

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

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

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

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

(**pushnew** *foo* $\widetilde{place}$ {|{**:test** *function*$_{\text{#'eql}}$ | **:test-not** *function*} | **:key** *function*})
▷ Set *place* to (**adjoin** *foo place*).

(**append** [*list** *foo*])
(**nconc** [$\widetilde{list}$* *foo*])
▷ Return concatenated list. *foo* can be of any type.

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

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

({**mapcan**, **mapcon**} *function list*+)
▷ Return list of concatenated return values of *function* successively invoked with corresponding arguments, either cars or cdrs, respectively, from each *list*. *function* should return a list.

({**mapc**, **mapl**} *function list*+)
▷ Return first *list* after successively applying *function* to corresponding arguments, either cars or cdrs, respectively, from each *list*. *function* should have some side effects.

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

4.3 Association Lists

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

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

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

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

4.4 Trees

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

({**subst** *new old tree*, **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*.

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

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

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

14.2 Packages

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

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

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

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

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

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

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

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

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

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

package~common-lisp-user~ ▷ The current package.

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

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

(**package-nicknames** *package*) ▷ List of nicknames of *package*.

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

(**find-all-symbols** *name*)
▷ Return list of symbols with *name* from all registered packages.

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

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

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

(**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.

(**the** *type form*)
▷ Return values of *form* which are declared to be of *type*.

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

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

({**ctypecase**|**etypecase**} *foo* (*type form**)*)
▷ Return values of the *forms* whose *type* is *foo* of. Signal correctable/non-correctable error, respectively if no *type* matches.

(**type-of** *foo*) ▷ Type of *foo*.

(**check-type** *place type* [*string*])
▷ Return NIL and signal correctable **type-error** if *place* is not of *type*.

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

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

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

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

14 Packages and Symbols

14.1 Predicates

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

4.5 Sets

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

5 Arrays

5.1 Predicates

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

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

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

5.2 Array Functions

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

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

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

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

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

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

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

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

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

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

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

Common Lisp Quick Reference

({ **bit-eqv**[Fu] | **bit-and**[Fu] | **bit-andc1**[Fu] | **bit-andc2**[Fu] | **bit-nand**[Fu] | **bit-ior**[Fu] | **bit-orc1**[Fu] | **bit-orc2**[Fu] | **bit-xor**[Fu] | **bit-nor**[Fu] } *bit-array-a bit-array-b* [*result-bit-array*$_{\text{NIL}}$])

▷ Return <u>result</u> of bitwise logical operations (cf. operations of **boole**[Fu], p. 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.

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

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

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

5.3 Vector Functions

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

(**vector**[Fu] *foo**) ▷ Return fresh <u>simple vector of *foo*s</u>.

(**svref**[Fu] *vector i*) ▷ Return <u>element *i*</u> of simple *vector*. **setf**able.

(**vector-push**[Fu] *foo* $\widetilde{vector}$)
▷ Return <u>`NIL`</u> if *vector*'s fill pointer equals size of *vector*. Otherwise replace element of *vector* pointed to by <u>fill pointer</u> with *foo*; then increment fill pointer.

(**vector-push-extend**[Fu] *foo* $\widetilde{vector}$ [*num*])
▷ Replace element of *vector* pointed to by <u>fill pointer</u> with *foo*, then increment fill pointer. Extend *vector*'s size by $\geq$ *num* if necessary.

(**vector-pop**[Fu] $\widetilde{vector}$)
▷ Return <u>element of *vector*</u> its fillpointer points to after decrementation.

(**fill-pointer**[Fu] *vector*) ▷ <u>Fill pointer</u> of *vector*. **setf**able.

6 Sequences

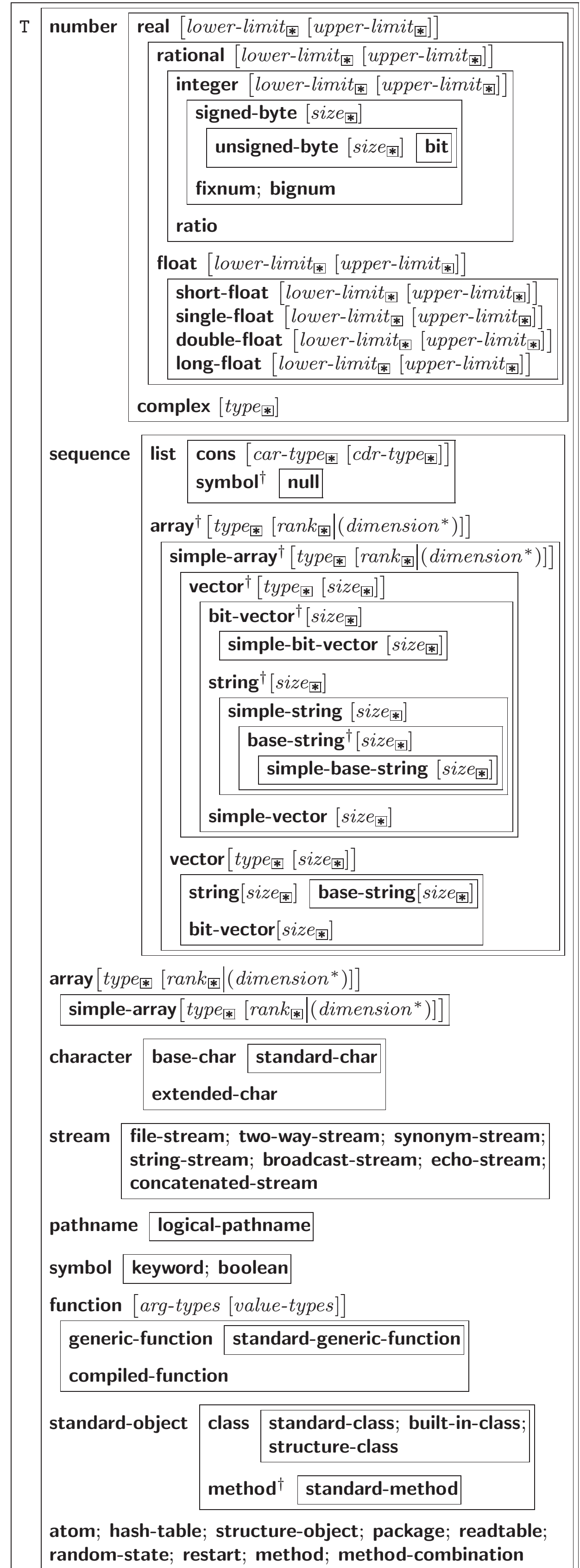
6.1 Sequence Predicates

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

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

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

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



†For supertypes of this type look for the instance without a †.
As a type argument, * means no restriction.

Figure 3: Data Types.

({pathname-host | pathname-device | pathname-directory | pathname-name | pathname-type} *path* [:case {:local | :common}$_{\text{:local}}$])
(pathname-version *path*)
▷ Return pathname component.

(logical-pathname *path*) ▷ Logical name of *path*.

(translate-pathname *path-a path-b path-c*)
▷ Translate *path-a* from wildcard *path-b* into wildcard *path-c*. Return new path.

(logical-pathname-translations *host*)
▷ *host*'s list of translations. **setf**able.

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

(translate-logical-pathname *path*)
▷ Physical pathname of *path*.

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

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

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

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

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

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

(rename-file *foo bar*)
▷ Rename file *foo* to *bar*. Unspecified parts of path *bar* default to those of *foo*. Return new pathname, old file name$_2$, and new file name$_3$.

(delete-file *file*) ▷ Delete *file*, return T.

(directory *path*) ▷ Return list of pathnames.

(ensure-directories-exist *path* [:verbose *bool*])
▷ Create parts of *path* if necessary. Second return value is T$_2$ if something has been created.

(with-open-file (*stream path open-arg**) (declare $\widehat{decl}$*)* *form*^{P_*})
▷ Use **open** with *open-args* (cf. page 38) to temporarily create *stream* to *path*; return values of *forms*.

(cf. page 38)

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

13 Types and Classes

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

(typep *foo type* [*environment*$_{\text{NIL}}$])
▷ Return T if *foo* is of *type*.

6.2 Sequence Functions

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

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

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

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

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

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

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

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

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

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

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

({find | position} *foo sequence* {| :from-end *bool*$_{\text{NIL}}$ | {:test *test*$_{\text{#'eql}}$ | :test-not *test*} | :start *start*$_0$ | :end *end*$_{\text{NIL}}$ | :key *function*})
▷ Return first element in *sequence* which satisfies *test*, or its position relative to the begin of *sequence*, respectively.

({find-if | find-if-not | position-if | position-if-not} *test sequence* {| :from-end *bool*$_{\text{NIL}}$ | :start *start*$_0$ | :end *end*$_{\text{NIL}}$ | :key *function*})
▷ Return first element in *sequence* which satisfies *test*, or its position relative to the begin of *sequence*, respectively.

(search *sequence-a sequence-b* {| :from-end *bool*$_{\text{NIL}}$ | {:test *function*$_{\text{#'eql}}$ | :test-not *function*} | :start1 *start-a*$_0$ | :start2 *start-b*$_0$ | :end1 *end-a*$_{\text{NIL}}$ | :end2 *end-b*$_{\text{NIL}}$ | :key *function*})

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

({**remove** *foo sequence* | **delete** *foo sequence*} {| :from-end *bool*_NIL | {:test *function*_#'eql | :test-not *function*} | :start *start*_0 | :end *end*_NIL | :key *function* | :count *count*_NIL})

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

({{**remove-if** | **remove-if-not**} *test sequence* | {**delete-if** | **delete-if-not**} *test sequence*} {| :from-end *bool*_NIL | :start *start*_0 | :end *end*_NIL | :key *function* | :count *count*_NIL})

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

({**remove-duplicates** *sequence* | **delete-duplicates** *sequence*} {| :from-end *bool*_NIL | {:test *function*_#'eql | :test-not *function*} | :start *start*_0 | :end *end*_NIL | :key *function*})

▷ Make copy of *sequence* without duplicates.

({**substitute** *new old sequence* | **nsubstitute** *new old sequence*} {| :from-end *bool*_NIL | {:test *function*_#'eql | :test-not *function*} | :start *start*_0 | :end *end*_NIL | :key *function* | :count *count*_NIL})

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

({{**substitute-if** | **substitute-if-not**} *new test sequence* | {**nsubstitute-if** | **nsubstitute-if-not**} *new test sequence*} {| :from-end *bool*_NIL | :start *start*_0 | :end *end*_NIL | :key *function* | :count *count*_NIL})

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

(**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*.

(**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.

(**map-into** *result-sequence function sequence**)

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

(**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.

(**copy-seq** *sequence*)

▷ Return copy of *sequence* with shared elements.

(**with-input-from-string** (*foo string* {| :index *index* | :start *start*_0 | :end *end*_NIL}) (**declare** *decl**)* *form*^P*)

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

(**with-output-to-string** (*foo* [*string*_NIL] [:element-type *type*_character]) (**declare** *decl**)* *form*^P*)

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

(**stream-external-format** *stream*)

▷ External file format designator.

terminal-io ▷ Bidirectional stream to user terminal.

standard-input
standard-output
error-output

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

debug-io
query-io

▷ Bidirectional streams for debugging and user interaction.

12.7 Files

(**make-pathname** {| :host *host* | :device *dev* | :directory *dir* | :name *name* | :type *type* | :version *ver* | :defaults *path* | :case {:local | :common}_:local})

▷ Construct pathname.

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

▷ Return *pathname* after filling in missing parts from defaults.

default-pathname-defaults

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

(**pathname** *path*) ▷ Pathname of *path*.

(**enough-namestring** *path* [*root-path*_*default-pathname-defaults*])

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

(**namestring** *path*)
(**file-namestring** *path*)
(**directory-namestring** *path*)
(**host-namestring** *path*)

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

(**parse-namestring** *foo* [*host* [*default-pathname*_*default-pathname-defaults* {| :start *start*_0 | :end *end*_NIL | :junk-allowed *bool*_NIL}]])

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

12.6 Streams

(open *path* {:direction {:input | :output | :io | :probe}[:input] | :element-type *type*[character] | :if-exists {:new-version | :error | :rename | :rename-and-delete | :overwrite | :append | :supersede | NIL} | :if-does-not exist {:error | :create | NIL} | :external-format *format*[:default]})
▷ Open **file-stream** to *path*.

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

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

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

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

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

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

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

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

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

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

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

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

7 Hash Tables

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

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

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

(gethash *key hash-table* [*default*[NIL]])
▷ Return object with *key* if any or *default* otherwise; and T (2) if found, NIL (2) otherwise. **setf**able.

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

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

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

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

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

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

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

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

8 Structures

(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 *type* | :read-only *bool*}])}*) | {(:type {list | vector | (vector *size*)}) {:named | (:initial-offset *n*)}} | {(:print-object [*o-printer*]) | (:print-function [*f-printer*])} | {:predicate | (:predicate [*p-name*[*foo*-P]])}})}

[$\widehat{doc}$] {*slot* | (*slot* [*init* {|**:type** $\widehat{type}$ | |**:read-only** $\widehat{bool}$}])}*)

▷ Define structure type foo together with functions MAKE-*foo*, COPY-*foo* and (unless **:type** without **:named** is used) *foo*-P; and **setf**able accessors *foo*-*slot*. Instances of type *foo* can be created by (MAKE-*foo* {:*slot value*}*) or, if *ord*-λ (see p. 17) is given, by (*maker arg** {:*key value*}*). In the latter case, *args* and :*keys* correspond to the positional and keyword parameters defined in *ord*-λ whose *vars* in turn correspond to *slots*. **:print-object**/**:print-function** generate a **print-object** method for an instance *bar* of *foo* calling (*o-printer bar stream*) or (*f-printer bar stream print-level*), respectively.

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

9 Control Structure

9.1 Predicates

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

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

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

(**equalp** *foo bar*)
▷ T if *foo* and *bar* are identical; or are the same **character** ignoring case; or are **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 **equalp** elements; or are structures of the same type with **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 **equalp** elements.

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

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

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

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

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

9.2 Variables

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

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

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

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

~[$n_{\boxed{1}}$]~ ▷ **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 *texts* in a field of at least *min-col* columns. With :, right justify; with @, left justify. If this would leave less than *spare* characters on the current line, output *nl-text* first.

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

{~[$n_{\boxed{0}}$]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.** *text* is used repeatedly, up to *limit*, as control string for the elements of the list argument or (with @) for the remaining arguments. With : or :@, list elements or remaining arguments should be lists of which a new one is used at each iteration step.

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

~[*i*][:][@][[{*text*~;}* *text*][~:;*default*]~]
▷ **Conditional Expression.** The *texts* are format control subclauses the zero-indexed argumenth (or the *i*th if given) of which is chosen. With :, the argument is boolean and takes first *text* for NIL and second *text* for T. With @, the argument is boolean and if T, takes the only *text* and remains to be read; no *text* is chosen and the argument is used up if it is NIL.

~[@]?
▷ **Recursive Processing.** Process two arguments as **format** string and argument list. With @, take one argument as **format** string and use then the rest of the original arguments.

~[*prefix* {, *prefix*}*][:][@]/*function*/
▷ **Call Function.** Call *function* with the arguments stream, format-argument, colon-p, at-sign-p and *prefixes* for printing format-argument.

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

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

(**format** {T|NIL|*out-string*|*out-stream*} *control arg**)
▷ Output string *control* which may contain ~ directives possibly taking some *args*. Alternatively, *control* can be a function returned by **formatter** which is then applied to *out-stream* and *arg**. Output to *out-string*, *out-stream* or, if first argument is T, to ***standard-output***. Return NIL. If first argument is NIL, return formatted output.

~[*min-col*₀] [,[*col-inc*₁] [,[*min-pad*₀] [,'*pad-char*␣]]] [:][@]{A|S}
▷ **Aesthetic/Standard.** Print argument of any type for consumption by humans/by the reader, respectively. With :, print NIL as () rather than nil; with @, add *pad-char*s on the left rather than on the right.

~[*radix*₁₀] [,[*width*] [,['*pad-char*␣] [,['*comma-char*,] [,*comma-interval*₃]]]] [:][@]R
▷ **Radix.** (With one or more prefix arguments.) Print argument as number; with :, group digits *comma-interval* each; with @, always prepend a sign.

{~R|~:R|~@R|~@:R}
▷ **Roman.** Take argument as number and print it as English cardinal number, as English ordinal number, as Roman numeral, or as old Roman numeral, respectively.

~[*width*] [,['*pad-char*␣] [,['*comma-char*,] [,*comma-interval*₃]]] [:][@]{D|B|O|X}
▷ **Decimal/Binary/Octal/Hexadecimal.** Print integer argument as number. With : group digits *comma-interval* each; with @, always prepend a sign.

~[*width*] [,[*dec-digits*] [,[*shift*₀] [,['*overflow-char*] [,'*pad-char*␣]]]] [@]F
▷ **Fixed-Format Floating-Point.** With @, always prepend a sign.

~[*width*] [,[*int-digits*] [,[*exp-digits*] [,[*scale-factor*₁] [,['*overflow-char*] [,['*pad-char*␣] [,'*exp-char*]]]]]] [@]{E|G}
▷ **Exponential/General Floating-Point.** Print argument as floating-point number with *int-digits* before decimal point and *exp-digits* in the signed exponent. With ~G, choose either ~E or ~F. With @, always prepend a sign.

~[*dec-digits*₂] [,[*int-digits*₁] [,[*width*₀] [,'*pad-char*␣]]] [:][@]$
▷ **Monetary Floating-Point.** Print argument as fixed-format floating-point number. With :, put sign before any padding; with @, always prepend a sign.

{~C|~:C|~@C|~@:C}
▷ **Character.** Print, spell out, print in #\ syntax, or tell how to type, respectively, argument as (possibly non-printing) character.

{~(*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 and following whitespace. With :, ignore only newline; with @, ignore only following whitespace.

Common Lisp Quick Reference

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

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

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

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

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

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

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

(**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*.

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

9.3 Functions

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

(*var** [**&optional** {*var*|(*var* [*init*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.

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

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

(**function**[sO] $\begin{Bmatrix} foo \\ (\textbf{lambda}[M]\ form^*) \end{Bmatrix}$)
▷ Return lexically innermost function named *foo* or a lexical closure of the **lambda**[M] expression.

(**apply**[Fu] $\begin{Bmatrix} function \\ (\textbf{setf}\ function) \end{Bmatrix}$ *arg*$^+$)
▷ Return values of *function* called on *args*. Last *arg* must be a list. **setf**able if *function* is one of **aref**[Fu], **bit**[Fu], and **sbit**[Fu].

(**funcall**[Fu] *function* arg*)
▷ Return values of *function* called with *args*.

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

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

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

(**multiple-value-list**[Fu] *form*)
▷ Return in a list values of *form*.

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

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

(**constantly**[Fu] *foo*)
▷ Return function of any number of arguments returning *foo*.

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

(**function-lambda-expression**[Fu] *function*)
▷ If available, return lambda expression of *function*, $\underline{\texttt{NIL}}_2$ if *function* was defined in an environment without bindings, and $\underline{\text{name}}_3$ of *function*.

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

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

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

multiple-values-limit[co]
▷ 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** $\begin{Bmatrix} var \\ (\begin{Bmatrix} var \\ (macro\text{-}\lambda^*) \end{Bmatrix} [init_{\texttt{NIL}} [supplied\text{-}p]]) \end{Bmatrix}^*$] [*E*]

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

[**&key** $\begin{Bmatrix} var \\ (\begin{Bmatrix} var \\ (\textbf{:}key\ \begin{Bmatrix} var \\ (macro\text{-}\lambda^*) \end{Bmatrix}) \end{Bmatrix} [init_{\texttt{NIL}} [supplied\text{-}p]]) \end{Bmatrix}^*$ [*E*]

[**&allow-other-keys**]] [**&aux** $\begin{Bmatrix} var \\ (var\ [init_{\texttt{NIL}}]) \end{Bmatrix}^*$] [*E*])

Common Lisp Quick Reference

(**pprint-exit-if-list-exhausted**[M])
▷ If *list* is empty, terminate logical block. Return NIL otherwise.

(**pprint-newline**[Fu] $\begin{Bmatrix} \textbf{:linear} \\ \textbf{:fill} \\ \textbf{:miser} \\ \textbf{:mandatory} \end{Bmatrix}$ [$\widetilde{stream}$ $_{\textbf{*standard-output*}}$])
▷ Print a conditional newline if *stream* is a pretty printing stream. Return NIL.

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

print-base[var] $_{10}$ ▷ Radix for printing rationals, from 2 to 36.

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

print-circle[var] $_{\texttt{NIL}}$
▷ If T, avoid indefinite recursion while printing circular structure.

print-escape[var] $_{\texttt{T}}$
▷ If NIL, do not print escape characters and package prefixes.

print-gensym[var] $_{\texttt{T}}$ ▷ If T, print **#:** before uninterned symbols.

print-length[var] $_{\texttt{NIL}}$
print-level[var] $_{\texttt{NIL}}$
print-lines[var] $_{\texttt{NIL}}$
▷ If integer, restrict printing of objects to that number of elements per level/to that depth/to that number of lines.

print-miser-width[var]
▷ Width below which a compact pretty-printing style is used.

print-pretty[var] ▷ If T, print pretty.

print-radix[var] $_{\texttt{NIL}}$ ▷ If T, print rationals with a radix indicator.

print-readably[var] $_{\texttt{NIL}}$
▷ If T, print **read**[Fu]ably or signal error **print-not-readable**.

print-right-margin[var] $_{\texttt{NIL}}$
▷ Right margin width in ems while pretty-printing.

(**set-pprint-dispatch**[Fu] *type function* [*priority*$_0$ [*table*$_{\textbf{*print-pprint-dispatch*}}$]])
▷ 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.

(**pprint-dispatch**[Fu] *foo* [*table*$_{\textbf{*print-pprint-dispatch*}}$])
▷ Return highest priority *function* associated with type of *foo* and $\underline{\texttt{T}}_2$ if there was a matching type specifier in *table*.

(**copy-pprint-dispatch**[Fu] [*table*$_{\textbf{*print-pprint-dispatch*}}$])
▷ Return copy of *table* or, if *table* is NIL, initial value of ***print-pprint-dispatch***[var].

print-pprint-dispatch[var] ▷ Current pretty print dispatch table.

12.5 Format

(**formatter**[M] $\widehat{control}$)
▷ Return function of stream and a **&rest** argument applying **format**[Fu] to stream, *control*, and the **&rest** argument returning NIL or any excess arguments.

(**write-char** *char* [*stream*$_{\text{*standard-output*}}$])
▷ Output $\underline{char}$ to *stream*.

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

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

(**write-sequence** *sequence stream* {|:**start** *start*$_{0}$ |:**end** *end*$_{\text{NIL}}$})
▷ Write elements of $\underline{sequence}$ to *stream*.

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

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

(**pprint-fill** *stream foo* [*parenthesis*$_{\text{T}}$ [*noop*]])
(**pprint-tabular** *stream foo* [*parenthesis*$_{\text{T}}$ [*noop* [*n*$_{16}$]]])
(**pprint-linear** *stream foo* [*parenthesis*$_{\text{T}}$ [*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 $\underline{\text{NIL}}$. Usable with **format** directive ~//.

(**pprint-logical-block** (*stream list* {|{:**prefix** *string*|:**per-line-prefix** *string*} |:**suffix** *string*$_{\text{""}}$})
(**declare** *decl**)* *form*^{P_*})
▷ Evaluate *forms*, which should print *list*, with *stream* locally bound to a pretty printing stream which outputs to the original *stream*. If *list* is in fact not a list, it is printed by **write**. Return $\underline{\text{NIL}}$.

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

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

(**pprint-indent** {:**block**|:**current**} *n* [*stream*$_{\text{*standard-output*}}$])
▷ Specify indentation for innermost logical block relative to leftmost position/to current position. Return $\underline{\text{NIL}}$.

Common Lisp Quick Reference

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

[**&optional** {*var*|({*var*|(*macro-λ**)} [*init*$_{\text{NIL}}$ [*supplied-p*]])}*] [*E*] . *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.

({**defmacro**|**define-compiler-macro**} {*foo*|(**setf** *foo*)} (*macro-λ**) (**declare** *decl**)* [*doc*] *form*^{P_*})
▷ Define macro $\underline{foo}$ which on evaluation as (*foo tree*) applies expanded *forms* to arguments from *tree*, which corresponds to *tree*-shaped *macro-λ*s. *forms* are enclosed in an implicit **block** *foo*.

(**define-symbol-macro** *foo form*)
▷ Define symbol macro $\underline{foo}$ which on evaluation evaluates expanded *form*.

(**macrolet** ((*foo* (*macro-λ**) (**declare** *local-decl**)* [*doc*] *macro-form*^{P_*})*) (**declare** *decl**)* *form*^{P_*})
▷ Evaluate $\underline{forms}$ with locally defined mutually invisible macros *foo* which are enclosed in implicit **block**s of the same name.

(**symbol-macrolet** ((*foo expansion-form*)*) (**declare** *decl**)* *form*^{P_*})
▷ Evaluate $\underline{forms}$ with locally defined symbol macros *foo*.

(**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*$_{\text{NIL}}$ [*supplied-p*]])}*] [**&rest** *var*]
[**&key** {*var*|({*var*|(:*key var*)} [*init*$_{\text{NIL}}$ [*supplied-p*]])}*
[**&allow-other-keys**]] [**&environment** *var*])
▷ Specify how to **setf** a place accessed by $\underline{function}$. **Short form:** (**setf** (*function arg**) *value-form*) is replaced by (*updater arg** *value-form*); the latter must return *value-form*. **Long form:** on invocation of (**setf** (*function arg**) *value-form*), *forms* must expand into code that sets the place accessed where *setf-λ* and *s-var** describe the arguments of *function* and the value(s) to be stored, respectively; and that returns the value(s) of *s-var**. *forms* are enclosed in an implicit **block** named *function*.

(**define-setf-expander** *function* (*macro-λ**) (**declare** *decl**)* [*doc*] *form*^{P_*})
▷ Specify how to **setf** a place accessed by $\underline{function}$. On invocation of (**setf** (*function arg**) *value-form*), *form** must expand into code returning *arg-vars*, *args*, *newval-vars*, *set-form*, and *get-form* as described with **get-setf-expansion** where the elements of macro lambda list *macro-λ** are bound to corresponding *args*. *forms* are enclosed in an implicit **block** *function*.

(**get-setf-expansion** *place* [*environment*$_{\text{NIL}}$])
▷ Return lists of temporary variables $\underline{arg\text{-}vars}$ and of corresponding $\underline{args}_2$ as given with *place*, list $\underline{newval\text{-}vars}_3$ with temporary variables corresponding to the new values, and $\underline{set\text{-}form}_4$ and $\underline{get\text{-}form}_5$ specifying in terms of *arg-vars* and *newval-vars* how to **setf** and how to read *place*.

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

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

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

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

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

&key *var**
▷ Bind *var*s to corresponding keyword arguments.

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

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

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

9.5 Control Flow

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

(**cond** (*test then**~*test*~)*)
▷ Return the values of the first *then** whose *test* returns T; return NIL if all *test*s return NIL.

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

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

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

(**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.

(**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.

(**progn** *form**~NIL~)
▷ Evaluate *form*s sequentially. Return values of last *form*.

(**multiple-value-prog1** *form-r form**)
(**prog1** *form-r form**)
(**prog2** *form-a form-r form**)
▷ Evaluate forms in order. Return values/1st value, respectively, of *form-r*.

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

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

#B; **#O**; **#X**; **#***n***R** ▷ Number of radix 2, 8, 16, or *n*.

#C(*a b***)** ▷ (**complex** *a b*), the complex number *a* + *b*i.

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

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

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

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

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

#P*string* ▷ A pathname.

#:*foo* ▷ Uninterned symbol *foo*.

#.*form* ▷ Read-time value of *form*.

read-eval~T~ ▷ If NIL, a **reader-error** is signalled by **#.**.

#*int***=** *foo* ▷ Give *foo* the label *int*.

#*int***#** ▷ Object labelled *int*.

#< ▷ Have the reader signal **reader-error**.

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

features
▷ List of symbols denoting implementation-dependent features.

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

12.4 Printer

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

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

(**print-object** *object stream*)
▷ Print *object* to *stream*. Called by the Lisp printer.

(**print-unreadable-object** (*foo stream* {|**:type** *bool*~NIL~ |**:identity** *bool*~NIL~}) *form**)
▷ Enclosed in **#<** and **>**, print *foo* by means of *form*s to *stream*. Return NIL.

(**terpri** [*stream*~***standard-output***~])
▷ Output a newline to *stream*. Return NIL.

(**fresh-line**) [*stream*~***standard-output***~]
▷ Output a newline to *stream* and return T unless *stream* is already at the start of a line.

(**read-line** [*stream*~*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.

(**read-sequence** *sequence stream* [**:start** *start*~0~][**:end** *end*~NIL~])
▷ Replace elements of *sequence* between *start* and *end* with elements from *stream*. Return index of *sequence*'s first unmodified element.

(**readtable-case** *readtable*)~:upcase~
▷ Case sensitivity attribute (one of **:upcase**, **:downcase**, **:preserve**, **:invert**) of *readtable*. **setf**able.

(**copy-readtable** [*from-readtable*~*readtable*~ [*to-readtable*~NIL~]])
▷ Return copy of *from-readtable*.

(**set-syntax-from-char** *to-char from-char* [*to-readtable*~*readtable*~ [*from-readtable*~standard readtable~]])
▷ Copy syntax of *from-char* to *to-readtable*. Return T.

readtable ▷ Current readtable.

read-base~10~ ▷ Radix for reading **integer**s and **ratio**s.

read-default-float-format~single-float~
▷ Floating point format to use when not indicated in the number read.

read-suppress~NIL~
▷ If T, reader is syntactically more tolerant.

(**set-macro-character** *char function* [*non-term-p*~NIL~ [*rt*~*readtable*~]])
▷ Make *char* a macro character associated with *function*. Return T.

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

(**make-dispatch-macro-character** *char* [*non-term-p*~NIL~ [*rt*~*readtable*~]])
▷ Make *char* a dispatching macro character. Return T.

(**set-dispatch-macro-character** *char sub-char function* [*rt*~*readtable*~])
▷ Make *function* a dispatch function of *char* followed by *sub-char*. Return T.

(**get-dispatch-macro-character** *char sub-char* [*rt*~*readtable*~])
▷ Dispatch function associated with *char* followed by *sub-char*.

12.3 Macro Characters and Escapes

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

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

(▷ Initiate reading of a list.

" ▷ Begin and end of a string.

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

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

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

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

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

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

(**multiple-value-bind** (*var**) *values-form* (**declare** *decl**)* *body-form**)
▷ Evaluate *body-forms* with *var*s lexically bound to the return values of *values-form*. Return values of *body-forms*.

(**block** *name form**)
▷ Evaluate *forms* in a lexical environment, and return their values unless interrupted by **return-from**.

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

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

(**go** *tag*)
▷ Within the innermost enclosing **tagbody**, jump to a tag **eql** *tag*.

(**catch** *tag form**)
▷ Evaluate *forms* and return their values unless interrupted by **throw**.

(**throw** *tag form*)
▷ Have the nearest dynamically enclosing **catch** with a tag **eq** *tag* return with the values of *form*.

(**sleep** *n*) ▷ Wait *n* seconds, return NIL.

9.6 Iteration

({**do** | **do***} ({*var* | (*var* [*start* [*step*]])}*) (*stop result**) (**declare** *decl**)* {*tag* | *form*}*)
▷ Evaluate **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 **block** named NIL.

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

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

9.7 Loop Facility

(**loop**[M] *form**)

▷ **Simple Loop.** If *forms* do not contain any atomic Loop Facility keywords, evaluate them forever in an implicit **block**[sO] named `NIL`.

(**loop**[M] *clause**)

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

named $n_{\boxed{\texttt{NIL}}}$ ▷ Give **loop**[M]'s implicit **block**[sO] 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**.

{**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 {$step_{\boxed{1}}$|$function_{\boxed{\#'cdr}}$}

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

= *foo* [**then** $bar_{\boxed{foo}}$]

▷ Bind *var* in the first iteration 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**} $package_{\boxed{\textbf{*package*}}}$]

▷ Bind *var* successively to the accessible symbols, or the present symbols, or the external symbols respectively, of *package*.

{**do**|**doing**} *form*⁺

▷ Evaluate *forms* in every iteration.

{**if**|**when**|**unless**} *test i-clause* {**and** *j-clause*}* [**else** *k-clause* {**and** *l-clause*}*] [**end**]

▷ If *test* returns `T`, `T`, or `NIL`, respectively, evaluate *i-clause* and *j-clauses*; otherwise, evaluate *k-clause* and *l-clauses*.

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

return {*form*|**it**}

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

12 Input/Output

12.1 Predicates

(**streamp**[Fu] *foo*)
(**pathnamep**[Fu] *foo*) ▷ `T` if *foo* is of indicated type.
(**readtablep**[Fu] *foo*)

(**input-stream-p**[Fu] *stream*)
(**output-stream-p**[Fu] *stream*)
(**interactive-stream-p**[Fu] *stream*)
(**open-stream-p**[Fu] *stream*)

▷ Return `T` if *stream* is for input, for output, interactive, or open, respectively.

(**pathname-match-p**[Fu] *path wildcard*)

▷ `T` if *path* matches *wildcard*.

(**wild-pathname-p**[Fu] *path* [{**:host**|**:device**|**:directory**|**:name**|**:type**|**:version**|`NIL`}])

▷ Return `T` if indicated component in *path* is wildcard. (`NIL` indicates any component.)

12.2 Reader

({**y-or-n-p**[Fu] | **yes-or-no-p**[Fu]} [*control arg**])

▷ Ask user a question and return `T` or `NIL` depending on their answer. See p. 35, **format**[Fu], for *control* and *args*.

(**with-standard-io-syntax**[M] *form*[P]*)

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

({**read**[Fu] | **read-preserving-whitespace**[Fu]} [$\widetilde{stream}_{\boxed{\textbf{*standard-input*}}}$[var] [$eof\text{-}err_{\boxed{\texttt{T}}}$ [$eof\text{-}val_{\boxed{\texttt{NIL}}}$ [$recursive_{\boxed{\texttt{NIL}}}$]]]])

▷ Read printed representation of object.

(**read-from-string**[Fu] *string* [$eof\text{-}error_{\boxed{\texttt{T}}}$ [$eof\text{-}val_{\boxed{\texttt{NIL}}}$ [{|**:start** $start_{\boxed{0}}$ | **:end** $end_{\boxed{\texttt{NIL}}}$ | **:preserve-whitespace** $bool_{\boxed{\texttt{NIL}}}$}]]])

▷ Return object read from string and zero-indexed position[2] of next character.

(**read-delimited-list**[Fu] *char* [$\widetilde{stream}_{\boxed{\textbf{*standard-input*}}}$[var] [$recursive_{\boxed{\texttt{NIL}}}$]])

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

(**read-char**[Fu] [$\widetilde{stream}_{\boxed{\textbf{*standard-input*}}}$[var] [$eof\text{-}err_{\boxed{\texttt{T}}}$ [$eof\text{-}val_{\boxed{\texttt{NIL}}}$ [$recursive_{\boxed{\texttt{NIL}}}$]]]])

▷ Return next character from *stream*.

(**read-char-no-hang**[Fu] [$\widetilde{stream}_{\boxed{\textbf{*standard-input*}}}$[var] [$eof\text{-}error_{\boxed{\texttt{T}}}$ [$eof\text{-}val_{\boxed{\texttt{NIL}}}$ [$recursive_{\boxed{\texttt{NIL}}}$]]]])

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

(**peek-char**[Fu] [$mode_{\boxed{\texttt{NIL}}}$ [$\widetilde{stream}_{\boxed{\textbf{*standard-input*}}}$[var] [$eof\text{-}error_{\boxed{\texttt{T}}}$ [$eof\text{-}val_{\boxed{\texttt{NIL}}}$ [$recursive_{\boxed{\texttt{NIL}}}$]]]]])

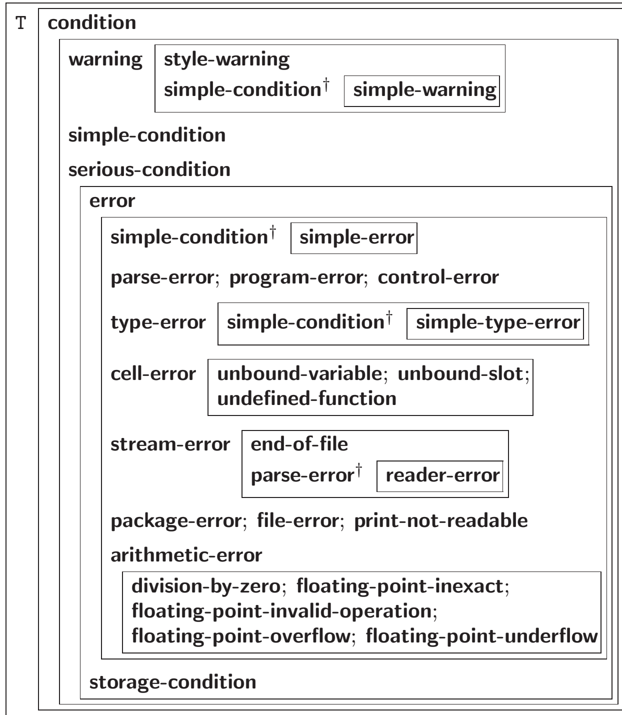
▷ Next, or if *mode* is `T`, next non-whitespace character, or if *mode* is a character, next instance of it, from stream without removing it there.

(**unread-char**[Fu] *character* [$\widetilde{stream}_{\boxed{\textbf{*standard-input*}}}$[var]])

▷ Put last **read-char**[Fu]ed *character* back into *stream*; return `NIL`.

(**read-byte**[Fu] $\widetilde{stream}$ [$eof\text{-}err_{\boxed{\texttt{T}}}$ [$eof\text{-}val_{\boxed{\texttt{NIL}}}$]])

▷ Read next byte from binary *stream*.



†For supertypes of this type look for the instance without a †.

Figure 2: Condition Types.

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

(**cell-error-name** *condition*)[Fu]
▷ Name of cell which caused *condition*.

(**unbound-slot-instance** *condition*)[Fu]
▷ Instance with unbound slot which caused *condition*.

(**print-not-readable-object** *condition*)[Fu]
▷ The object not readably printable under *condition*.

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

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

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

break-on-signals[var] NIL
▷ Condition type debugger is to be invoked on.

debugger-hook[var] NIL
▷ Function of condition and function itself. Called before debugger.

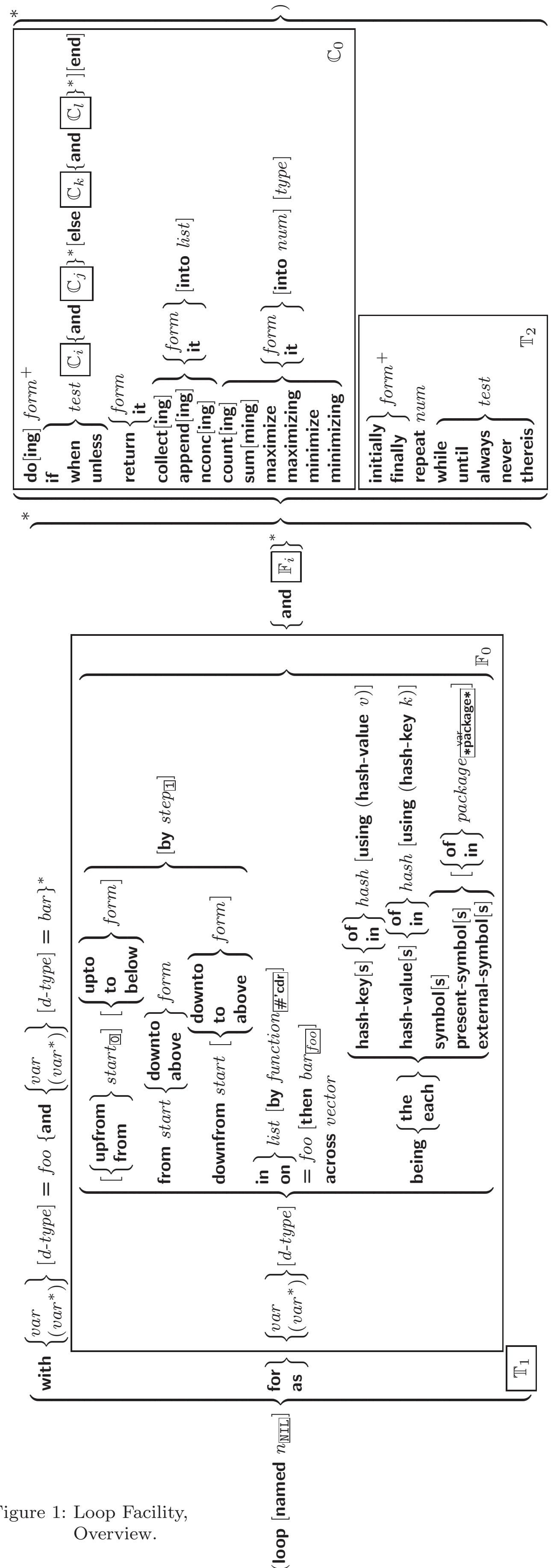


Figure 1: Loop Facility, Overview.

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

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

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

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

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

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

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

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

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

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

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

10 CLOS

10.1 Classes

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

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

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

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

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

(**with-simple-restart** ({*restart* | NIL} *control arg**) *form*^{P*})
▷ Return values of *forms* unless *restart* is called during their evaluation. In this case, describe restart using **format** *control* and *args* (see p. 35) and return NIL and T.

(**restart-case** *form* (*foo* (*ord-λ**) {**:interactive** *arg-function* | **:report** {*report-function* | *string*["*foo*"]} | **:test** *test-function*[T]} (**declare** $\widehat{decl^*}$)* *restart-form*^{P*})*)
▷ Evaluate *form* with dynamically established restarts *foo*. Return values of *form* or, if by (**invoke-restarts** *foo arg**) one restart *foo* is called, use *string* or *report-function* (of a stream) to print a description of restart *foo* and return the values of its *restart-forms*. *arg-function* supplies appropriate *args* if *foo* is called by **invoke-restart-interactively**. If (*test-function condition*) returns T, *foo* is made visible under *condition*. For (*ord-λ**) see p. 17.

(**restart-bind** (($\widehat{\{restart \mid NIL\}}$ *restart-function* {**:interactive-function** *function* | **:report-function** *function* | **:test-function** *function*})*) *form*^{P*})
▷ Return values of *forms* evaluated with *restarts* dynamically bound to *restart-functions*.

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

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

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

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

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

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

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

11 Conditions and Errors

(**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-types*, condition type *foo*. In a new condition, a *slot*'s value defaults to *form* unless set via **:***initarg-name*; it is readable via (*reader i*) or (*accessor i*), and writeable via (*writer i value*) 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.

(**make-condition** *type* {**:***initarg-name value*}*)

▷ Return new condition of *type*.

({**signal** | **warn** | **error**} {*condition* | *type* {**:***initarg-name value*}* | *control arg**})

▷ Unless handled, signal as **condition**, **warning** or **error**, respectively, *condition* or a new condition of *type* or, with **format** *control* and *args* (see p. 35), **simple-condition**, **simple-warning**, or **simple-error**, respectively. From **signal** and **warn**, return `NIL`.

(**cerror** *continue-control* {*condition continue-arg** | *type* {**:***initarg-name value*}* | *control arg**})

▷ Unless handled, signal as correctable **error** *condition* or a new condition of *type* or, with **format** *control* and *args* (see p. 35), **simple-error**. In the debugger, use **format** arguments *continue-control* and *continue-args* to tag the continue option. Return `NIL`.

(**ignore-errors** *form**)

▷ Return values of *forms* or, in case of **error**s, `NIL` and the condition.

(**invoke-debugger** *condition*)

▷ Invoke debugger with *condition*.

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

▷ If *test*, which may depend on *places*, returns `NIL`, signal as correctable **error** *condition* or a new condition of *type* or, with **format** *control* and *args* (see p. 35), **error**. When using the debugger's continue option, *places* can be altered before re-evaluation of *test*. Return `NIL`.

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

(**find-class** *symbol* [*errorp*[T] [*environment*]])

▷ Return class named *symbol*. **setf**able.

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

▷ Make new instance of *class*.

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

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

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

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

▷ Make *slot* in *instance* unbound.

({**with-slots** ({*slot* | (*var slot*)}*) | **with-accessors** ((*var accessor*)*)} *instance* (**declare** *decl**)* *form**)

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

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

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

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

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

(**make-instances-obsolete** *class*) ▷ Update instances of *class*.

({**initialize-instance** (*instance*) | **update-instance-for-different-class** *previous current*} {**:***initarg value*}* *other-keyarg**)

▷ Its primary method sets slots on behalf of **make-instance**/of **change-class** by means of **shared-initialize**.

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

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

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

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

(**shared-initialize** *instance* {*slots* | `T`} {**:***initarg value*}* *other-keyarg**)

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

(**slot-missing** *class object slot* {**setf** | **slot-boundp** | **slot-makunbound** | **slot-value**} [*value*])

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

(**slot-unbound** *class instance slot*)

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

10.2 Generic Functions

([Fu]**next-method-p**) ▷ T if enclosing method has a next method.

([M]**defgeneric** {*foo* | (**setf** *foo*)} (*required-var** [**&optional** {*var* | (*var*)}*] [**&rest** *var*] [**&key** {*var* | (*var* | (:*key var*))}* [**&allow-other-keys**]])
{ | (**:argument-precedence-order** *required-var*+)
| (**declare** (**optimize** *arg**)+)
| (**:documentation** *string*)
| (**:generic-function-class** *class* [standard-generic-function])
| (**:method-class** *class* [standard-method])
| (**:method-combination** *c-type* [standard] *c-arg**)
| (**:method** *defmethod-args*)* })

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

([Fu]**ensure-generic-function** {*foo* | (**setf** *foo*)}
{ | **:argument-precedence-order** *required-var*+
| **:declare** (**optimize** *arg**)+
| **:documentation** *string*
| **:generic-function-class** *class*
| **:method-class** *class*
| **:method-combination** *c-type c-arg**
| **:lambda-list** *lambda-list*
| **:environment** *environment* })

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

([M]**defmethod** {*foo* | (**setf** *foo*)} [{**:before** | **:after** | **:around** | *qualifier**} [primary method]]
({*var* | (*spec-var* {*class* | (**eql** *bar*)})}* [**&optional**
{*var* | (*var* [*init* [*supplied-p*]])}*] [**&rest** *var*] [**&key**
{*var* | ({*var* | (:*key var*)} [*init* [*supplied-p*]])}* [**&allow-other-keys**]]
[**&aux** {*var* | (*var* [*init*])}*]) {| (**declare** *decl**)* | *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 [sO]**block** *foo*. Applicable *qualifier*s depend on the **method-combination** type; see section 10.3.

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

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

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

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

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

({[Fu]**invalid-method-error** *method* | [Fu]**method-combination-error**} *control arg**)
▷ Signal **error** on applicable method with invalid qualifiers, or on method combination. For *control* and *args* see **format**, p. 35.

([gF]**no-next-method** *generic-function method arg**)
▷ Called on invocation of **call-next-method** when there is no next method. Default method signals **error**.

([gF]**function-keywords** *method*)
▷ Return list of keyword parameters of *method* and T (2) if other keys are allowed.

([gF]**method-qualifiers** *method*) ▷ List of qualifiers of *method*.

10.3 Method Combination Types

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

and | **or** | **append** | **list** | **nconc** | **progn** | **max** | **min** | **+**
▷ Simple built-in **method-combination** types; have the same usage as the *c-type*s defined by the short form of [M]**define-method-combination**.

([M]**define-method-combination** *c-type*
{ | **:documentation** *string*
| **:identity-with-one-argument** *bool* [NIL]
| **:operator** *operator* [c-type] })

▷ **Short Form.** Define new **method-combination** *c-type*. In a generic function using *c-type*, evaluate most specific **:around** method supplying the values of the generic function. From within this method, [Fu]**call-next-method** can call less specific **:around** methods if there are any. If not, or if there are no **:around** methods at all, have generic function applied to *gen-arg** return with the values of (*c-type* {*primary-method gen-arg**}*), leftmost *primary-method* being the most specific. In [M]**defmethod**, primary methods are denoted by the *qualifier c-type*.

([M]**define-method-combination** *c-type* (*ord-λ**) ((*group*
{ * | (*qualifier** [*]) | *predicate* }
{ | **:description** *control*
| **:order** {**:most-specific-first** | **:most-specific-last**} [:most-specific-first]
| **:required** *bool* })*)
{ | (**:arguments** *method-combination-λ**)
| (**:generic-function** *symbol*)
| (**declare** *decl**)*
| *doc* } *body*^P*)

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